

L3-info

Genie Logiciel

Introduction to UML/OCL

Burkhart Wolff

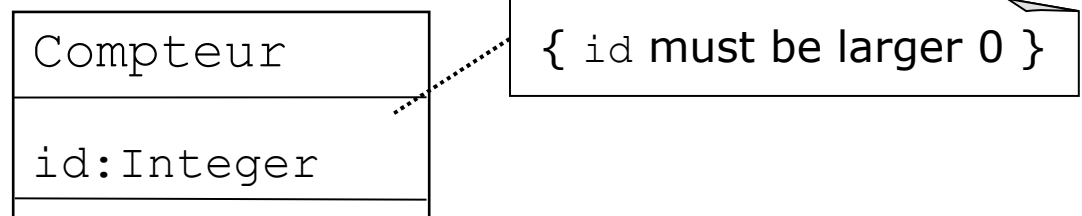
Département Informatique
Université Paris-Sud / Orsay

Plan of the Chapter

- ❑ Syntax & Semantics of OCL Constraint Expressions
 - Logic
 - Connection to UML
 - Basic Data-Types
 - Collection Types
- ❑ Semantics & Semantics of OCL Constraints
 - Class Invariants
 - Pre- and Post-Conditions
- ❑ Ultimate Goal:
Specify system components for test and verification

Motivation: What is OCL

- ❑ Acronym for **O**bject **C**onstraint **L**anguage
- ❑ Semantically: ... is a typed « Annotation language » (like JML, VCC) that constrains an underlying **state**
- ❑ Can be used (in principle) in all diagram types, where annotations were used ...



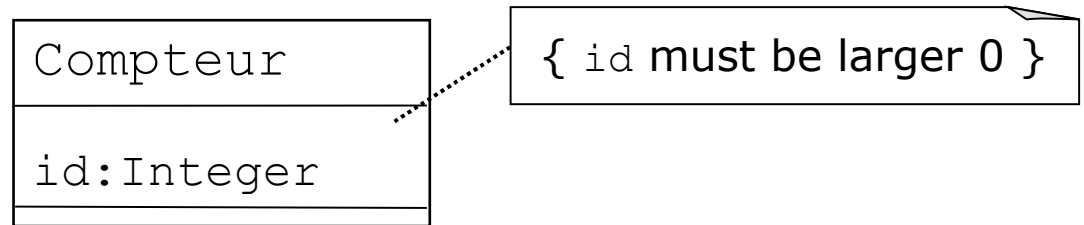
- ❑ ...

Motivation: What is OCL

- ❑ ...
- ❑ ... exists since UML 1.4, and is heavily used to define the UML Meta-Model itself
- ❑ ... is most comprehensively described in *J. Warmer, A. Kleppe : « The Object Constraint Language (Second edition) », Addison-Wesley, 2003*
- ❑ ... has a formal semantics, at least in some versions (OMG document ptc/03-10-14 , see also HOL-OCL)
- ❑ ... is usually not directly supported in CASE TOOLS, but quite a few PlugIns are available.

Why OCL ?

Well, not everything can be expressed in diagrams ...



Informal annotations should be expressed formally, too.

In particular, for:

- Classes and Associations (invariants)
- Operations (pre/post-conditions)
- Transition conditions in State Models (guards)

OCL: a specification language?

- ❑ OCL is not an imperative language:
no side-effects, no sequencing.
- ❑ However, OCL is similar to functional languages like `Ocaml`, `SML`, `Haskell`, ... and has a quite operational flavor: it is executable.
(No unbounded quantifiers in standard OCL !!!)
 - `let x1 : T1 = E1, ..., xn : Tn = En in E`
 - fold-like operators (called iterators):
`Collection->iterate(elem; acc:Integer=0
| acc + 1)`

OCL: a specification language?

- ❑ OCL has a special exceptional element:

`OclUndefined`

which can be result of illegal computations like
`1 / 0` or illegal accesses to objects in a state.

OCL: a specification language?

- All operations in OCL are strict, i.e. satisfy:

```
f(oclUndefined) = oclUndefined  
g(x, oclUndefined) = oclUndefined ...
```

except the operation `_.oclIsUndefined()` :

```
oclUndefined.oclIsUndefined() = true
```

Semantically, it is adequate to consider
`oclUndefined` as a raised exception and
`oclIsUndefined()` as catching it...

Syntax and Semantics of OCL

- ❑ OCL is a typed language. The underlying types of an OCL expression can be:
 - **Basic Types:**
Boolean, Integer, Real, String
 - **the hierarchy `Collection` with sub-types:**
Set, Ordered Set, Bag, Sequence
 - **the class-types of an associated class model**
 - **the enumeration type of an associated class model**
 - **special types:**
OclAny, OclVoid, Tuples

Syntax and Semantics of OCL

- The logical core language: expressions of type Boolean:
 - `not E, E or E', E and E', E implies E'`
 - `E = E', E <> E',`
 - `if C then E else E' endif`

 - Quantifiers are handled on the base of (finite) collections (see later):
 - `Collection->forall(x:Type| E(x))`
 - `Collection->exists(x:Type| E(x))`

Syntax and Semantics of OCL

- The arithmetic core language.
expressions of type `Integer` or `Real`:
 - $- E, E + E',$
 - $E * E', E / E',$
 - $E.abs(), E.div(E), \dots$

Syntax and Semantics of OCL

□ The expressions of type `String`:

- `E.concat(E')` -- « E ++ E' »
- `E.size()` -- « |E| »
- `E.substring(i, j)`
- `E.toInteger()`
- `E.toReal()`
- `'Hello'`

Syntax and Semantics of OCL Collections

- The power of the language comes from the expressions over `Collection's`, i.e. Set, Ordered Set, Bag, Sequence.
- Operations on Collections `C` are:
 - `C->iterate(elem; acc:T=init | op(elem, acc))`

- ... and its special cases:

<code>C->forall(x:Type E(x))</code>	--	<code>«$\forall x \in C \mid P(x)$»</code>
<code>C->exists(x:Type E(x))</code>	--	<code>«$\exists x \in C \mid P(x)$»</code>
<code>C->select(x:Type P(x))</code>	--	<code>«$\{x \in C \mid P(x)\}$»</code>
<code>C->reject(x:Type P(x))</code>	--	<code>«$\{x \in C \mid \neg P(x)\}$»</code>
<code>C->any(x:Type P(x))</code>	--	<code>«$\varepsilon x \in C \mid P(x)$»</code> , some x satisfying P
<code>C->one(x:Type P(x))</code>	--	<code>«$\{x \in C \mid P(x)\} = 1$»</code>

Syntax and Semantics of OCL Collections

- ...
- `C->size()`
- `C->count(e)`
- `C->isUnique(e)` -- `e` has different value for each element
- `C=C'`, `C<>C'` -- different semantics according type
- `C->isEmpty()`, `C->notEmpty()`
- `C->sum()` -- for a collection of numbers
- `C->includes(e)`, -- « $e \in C$ »
- `C->includesAll(C')` -- « $C' \subseteq C$ »
- `C->excludes(e)`, -- « $e \notin C$ »
- `C->excludesAll(C')` -- « $\forall x \in C' \mid x \notin C$ »
- ...

Syntax and Semantics of OCL Sequences

Additionally to the Collection operations,
Sequences S have the following ops:

- $S \rightarrow \text{first}()$
- $S \rightarrow \text{last}()$
- $S \rightarrow \text{at}(i)$ -- for i between 1 et $\text{size}()$
- $S \rightarrow \text{append}(e)$ -- append at the end
- $S \rightarrow \text{prepend}(e)$ -- append at the beginning
-
- ❑ Finally, denotations of collections:
 - $\text{Set}\{1,2,3\}$, $\text{Sequence}\{1,3,2,7,1\}$, ...

Syntax and Semantics of OCL / UML

- ❑ Class Models have OO-features reflected in UML/OCL.
- ❑ Based on a Class Model, families of operations for the conversion and the test of dynamic types (the type at object creation time) were introduced
- ❑ (an OCL expression is therefore only typeable in the CONTEXT of a Class Model)

Syntax and Semantics of OCL / UML

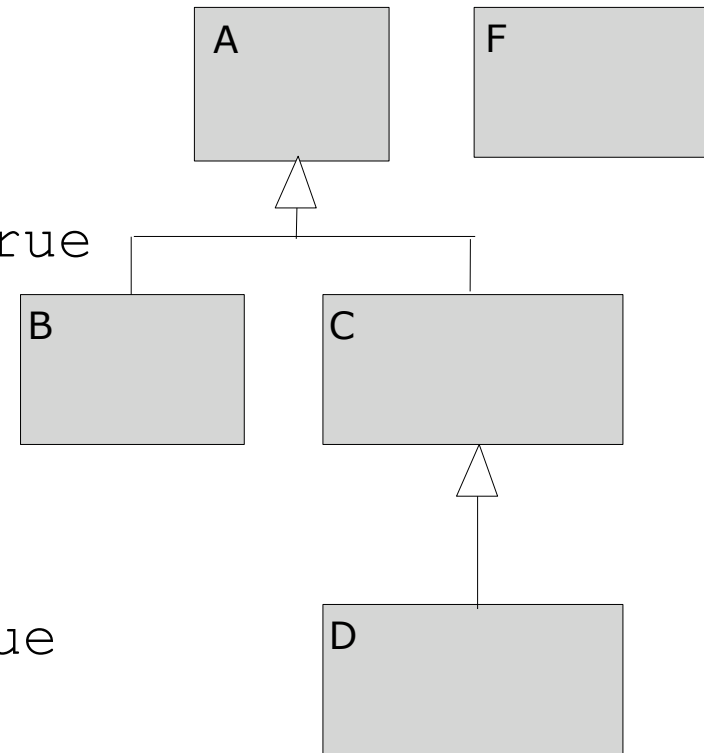
❑ Casting `e.oclAsType(oclType)`

- Create `a::A, b::B, c::C, d::D, f::D`

```
(b.oclAsType(F) .  
  oclIsUndefined()) = true
```

```
b.oclAsType(A) :: A  
d.oclAsType(A) :: A
```

```
(a.oclAsType(B) .  
  oclIsUndefined()) = true
```



Syntax and Semantics of OCL / UML

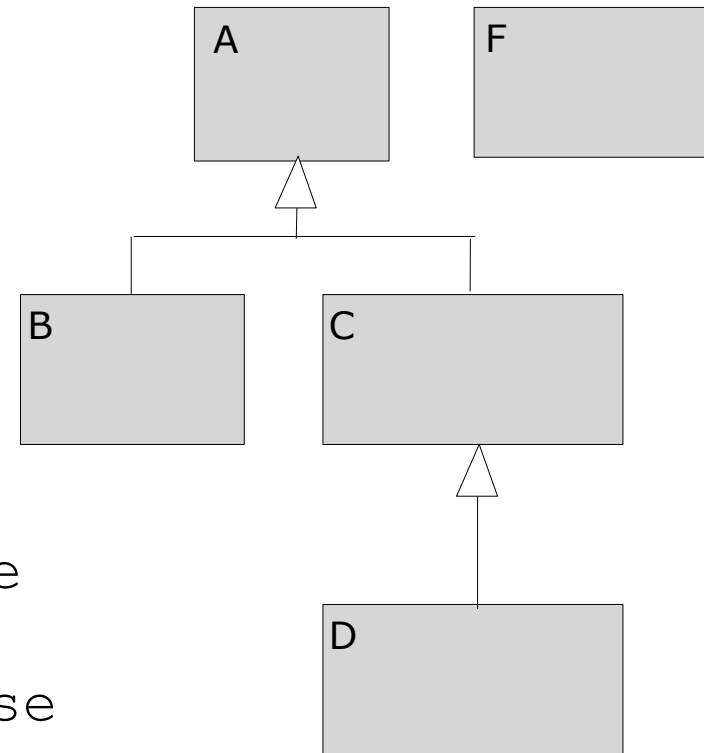
□ Type-Tests `e.oclIsType (oclType)`

- Create `a::A, b::B, c::C, d::D, f::D`

```
a.oclIsTypeOf (A) = true  
b.oclIsTypeOf (A) = false
```

```
(b.oclAsType (A) .  
  .oclIsTypeOf (B) ) = true  
(b.oclAsType (A) .  
  .oclIsTypeOf (A) ) = false
```

(Casts do not change the dynamic type)



Syntax and Semantics of OCL / UML

- A relaxed form of Type-Tests are Kind-Tests (admitting sub-types)

`e.oclIsKind(oclType)`

- Create `a::A, b::B, c::C, d::D, f::D`

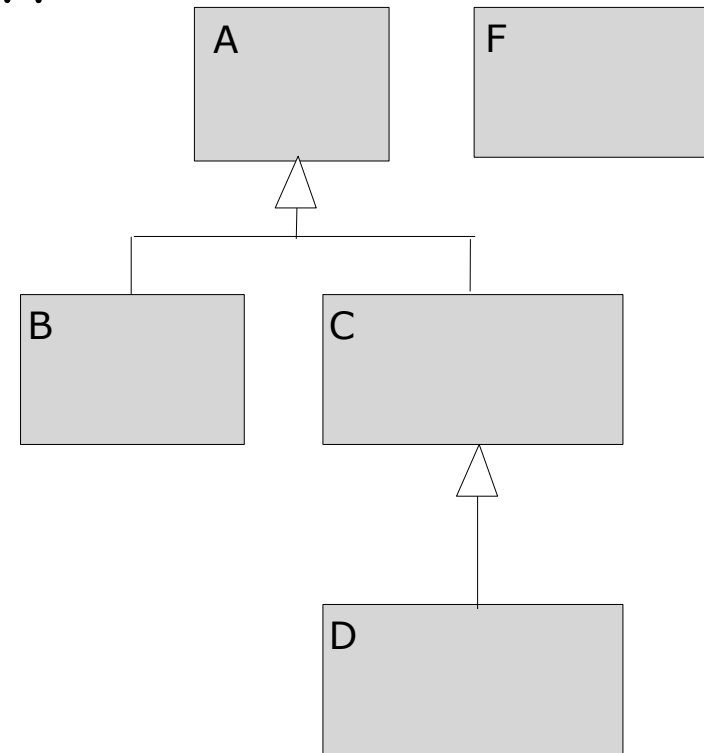
`a.oclIsKindOf(A) = true`

`c.oclIsKindOf(A) = true`

`b.oclIsKindOf(C) = false`

`a.oclIsKindOf(C) = false`

(Again: Casts do not change the dynamic type)



Syntax and Semantics of OCL / UML

- attributes and navigation ends result in « navigation paths » in object models ...

➤ Let $b :: B, c :: C$:



$b.d :: C$

$c.a :: B$

$b.d.a.d.a \dots$

Semantics: accessor functions correspond to dereferentiation in an underlying state. See in the sequel:

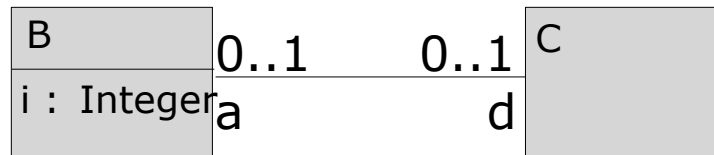
Syntax and Semantics of OCL / UML

- attributes and the pre-state. OCL expressions assume not only an underlying state σ' , but also a pre-state σ .

Syntax and Semantics of OCL / UML

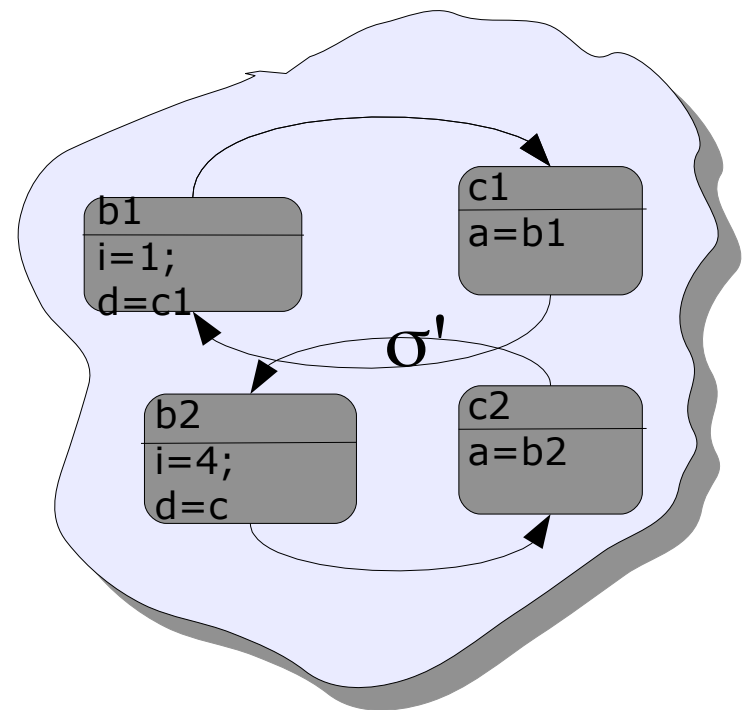
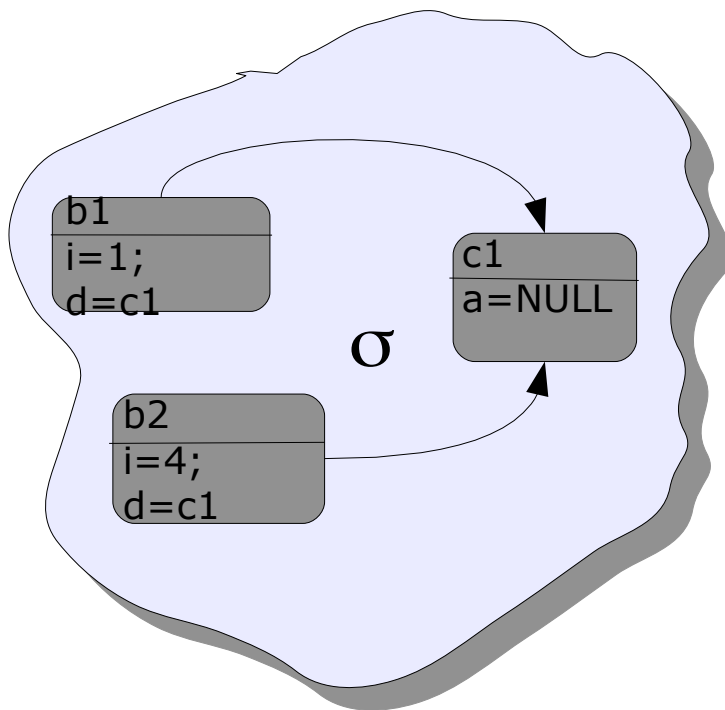
- attributes and the pre-state. OCL expressions assume not only an underlying state σ' , but also a pre-state σ .

In the following, we assume the following class model underlying the shown states:



Syntax and Semantics of OCL / UML

- attributes and the pre-state. OCL expressions assume not only an underlying state σ' , but also a pre-state σ .



Syntax and Semantics of OCL / UML

- attributes and the pre-state. OCL expressions assume not only an underlying state σ' , but also a pre-state σ . (As previous)
Thus, two forms of de-referentiation are possible:
 - Let $b :: B, c :: C$. Typing:

$b.d@pre :: C$ $c.a@pre :: B$

Semantics: accessor functions correspond to de-referentiation in σ' and pre-state σ :

$b.d.a.d.a.i < 5$

$b.d.a@pre.oclIsUndefined() = \text{True}$

$b.d@pre.a = 1$

Syntax and Semantics of OCL / UML

- Assume global state and pre-state **as previous**.
 - Then we have operations to produce all instances of objects in a state or pre-state:

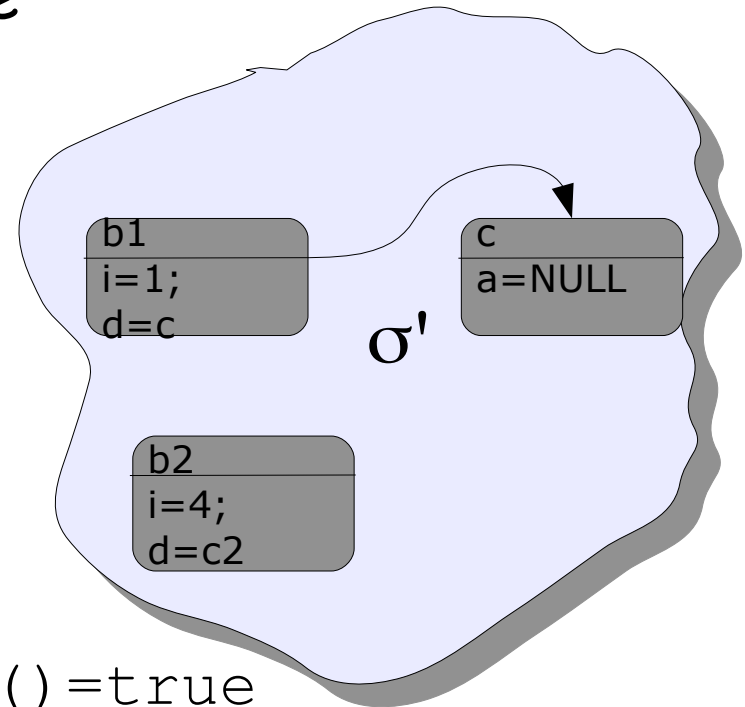
```
C.allinstances() = {c1, c2}
C.allinstances@pre() = {c1}
```

- ... or to check if an object is new:

```
c1.oclIsNew() = False
c2.oclIsNew() = True
```

Syntax and Semantics of OCL / UML

- Note that accessors may be undefined in a concrete object-model !!!



`b1.d.a.oclIsUndefined()=true`

`b2.d.a.oclIsUndefined()=true` -- „Dangling Reference“
& strictness

Syntax and Semantics of OCL / UML

- Automatic Flattening within path expressions !

➤ Let $b :: B, c :: C$:



$b.d :: \text{Set}(C) !!!$

$b.d.a :: \text{Set}(C) !!!$

$B.allInstances().d.a.i < 5$
is a legal statement in OCL !

Details complicated!

$\text{Set}(C)$ vs. $\text{Sequence}(C)$ vs. $\text{Bag}(C)$

Syntax and Semantics of Class Invariants

It is possible to annotate constraints to classes.

This is interpreted as « class invariant ».

```
context C           -- anonyme instance
inv name: P(self) -- where P(self) is a boolean
                        expression possibly using self,
                        but not containing @pre
```

Syntax and Semantics of Class Invariants

It is possible to annotate constraints to classes.

This is interpreted as « class invariant ».

OR

Syntax and Semantics of Class Invariants

It is possible to annotate constraints to classes.

This is interpreted as « class invariant ».

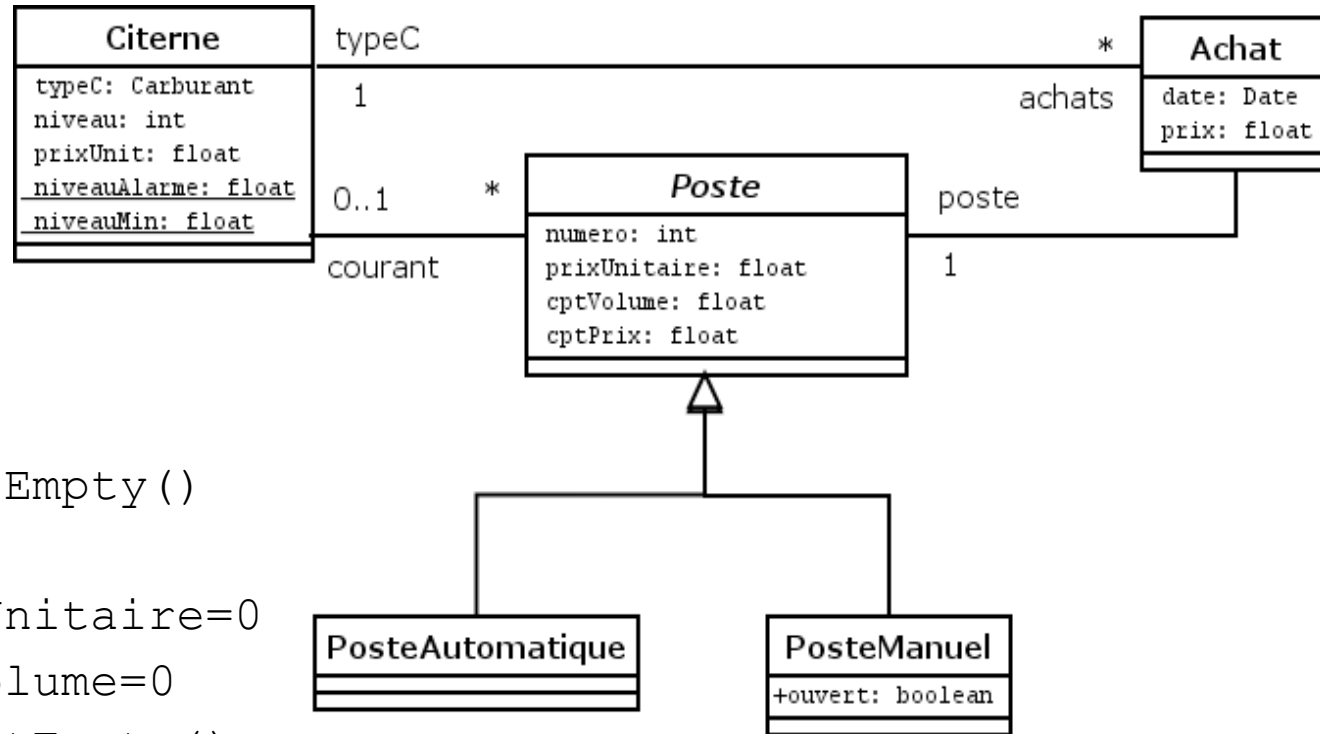
context $c:C$

-- anonyme instance

inv *name*: $P(c)$

-- where $P(c)$ is a boolean expression possibly using c , but not containing @pre

Example: Class Invariants



context p: Poste

inv: p.courant->isEmpty()

implies

p.prixUnitaire=0

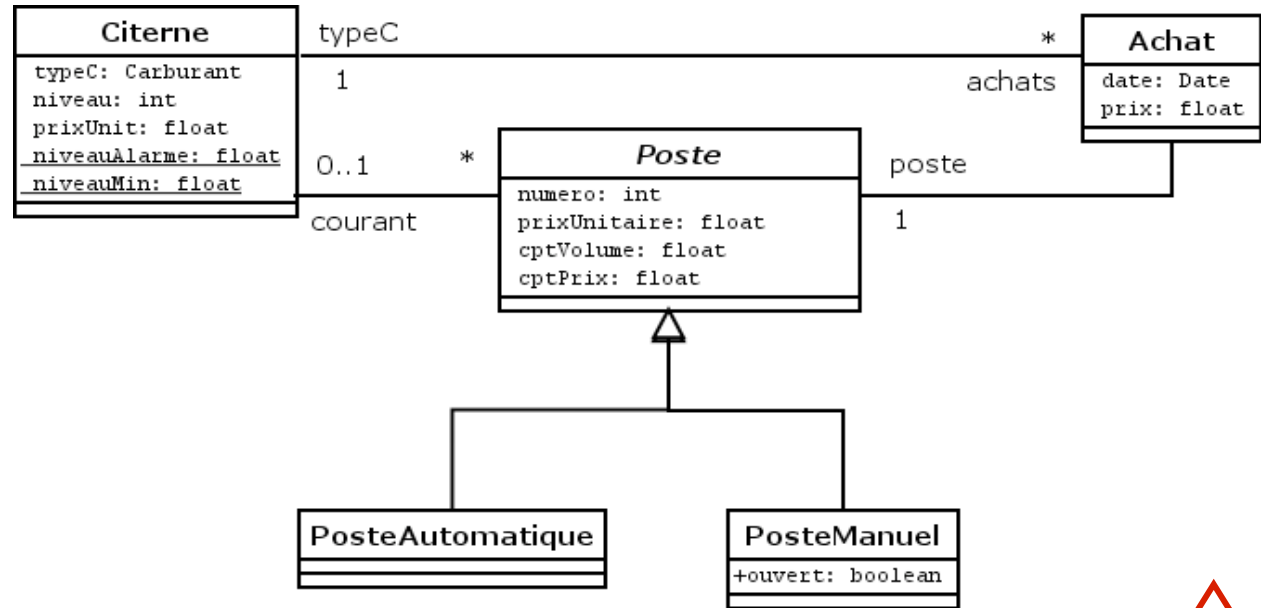
and p.cptVolume=0

inv: p.courant->notEmpty()

implies p.prixUnitaire = p.courant.prixUnit

inv: p.cptPrix = p.cptVolume * p.prixUnitaire

Class Invariants and Cardinalities



```

context a: Achat ... -- a.typeC :: Citerne
                    -- a.typeC.prixUnit :: Real

context c: Citerne... -- c.achats :: Set(Achat) or ???(Achat)

context p: Poste    -- p.courant-->size() <= 1 (abus de langage)
                    -- p.courant.prixUnit is oclUndefined if p.courant-
  
```

>isEmpty()
2008-2009



Semantics of Class Invariants

- ❑ Class Invariants constrain the set of possible state of a system to **valid states**
- ❑ OCL focusses ONLY on **valid states**, i.e. each object must satisfy its invariant:
 - in the initial state
 - after execution of each operation
 - in OCL (in contrast to a Hoare Calculus) invariants must always hold, at any (considered) time point.

Semantics of Class Invariants

- A Class Invariant Specification:

```
context  $c:C$   
inv name:  $P(c)$ 
```

Semantics of Class Invariants

- A Class Invariant Specification:

is an abbreviation for:

Semantics of Class Invariants

- A Class Invariant Specification:

`C.allinstances->forall(c|P(c))`

Consequently, statements like:

`C.allinstances->forall(c|
C.allinstances->forall(c|
P(c)) P(c))`

usually make no sense!

Semantics of Class Invariants

□ Exceptions:

context Commande

inv: `Commande.allInstances->isUnique`
`(c | c.numero)`

context Singleton

inv: `Singleton.allInstances->size() = 1`

Operations in UML

- ❑ **Operation** : interface and contract of a Method in UML (given in JAVA, C, C++, ...)
- ❑ A **methode implements** a method attached to a classe.
 - Between Class/Sub class, a method may re-implement a given operation.
 - One can also have overloading of operations (operations with same name but different parameter types; treated as if having different names).

Operations in UML

□ Syntactic View:

Context $C :: opname(arg_1, \dots, arg_n) : Type$

pre $P(arg_1, \dots, arg_n)$

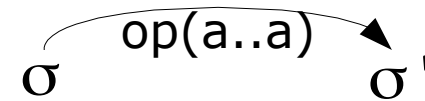
post $Q(arg_1, \dots, arg_n, result)$

where

- P may contain arg_1, \dots, arg_n but no $@pre$, and
- Q may contain arg_1, \dots, arg_n and the result.

Operations in UML

- Semantic View:



An operation is called in a valid pre-state σ , and if the arguments satisfy the precondition pre , then the system „jumps“ into the valid post-state σ' and returns a result satisfying the post-condition.

(There may be many solutions for $result$ and the σ' ; i.e the contract part of the OCL makes it to a true specification language.)

Methodology: Invariants and sub-classes

In order to substitute an instance of a super-class by an instance of a sub-class, the following construction must be made:

- ➡ **invariants of a superclass are inherited by a sub-class; they were added to the own invariants.**

Methodology: Invariants and sub-classes

Similarly, if a method overrides a method of the superclass:

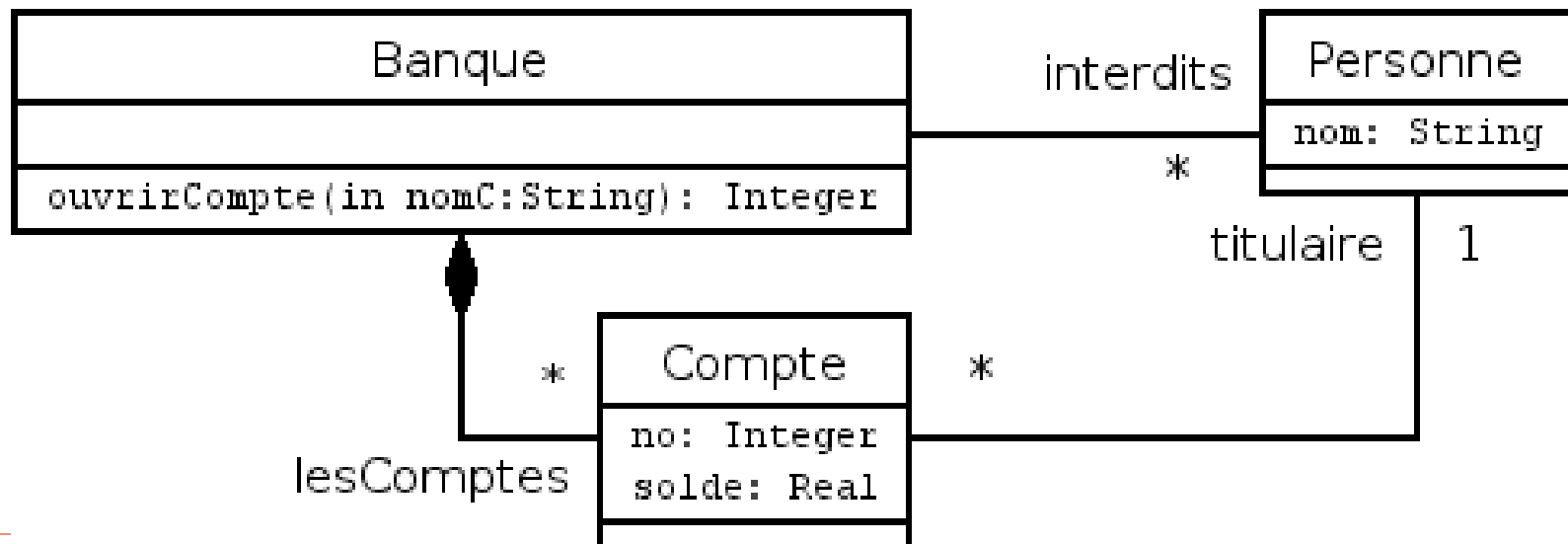
- ➡ the pre-condition can be identical or logically weaker than the precondition of the base method
- ➡ the post-condition can be identical or logically stronger than the postcondition of the base method.

In the Literature, this is often called Liskow's Principle.

Example: Bank

Opening a bank account. Constraints:

- ❑ there is a blacklist
- ❑ there is a present of 15 euros in the initial account
- ❑ account numbers must be distinct.



Example: Bank (2)

```
context Banque::ouvrirCompte(nomC: String) : Integer
```

```
    signal Refus
```

Déraisonnable ;-(

```
pre: Personne.allInstances->one(p | p.nom = nomc)
```

```
post: let p: Personne =
```

```
    Personne.allInstances->any(nom = nomc) in
```

```
    if interdits->includes(p2 | p2.nom = nomc)
```

```
        implies Refus.sent()
```

```
    else c.oclIsNew() and c.oclIsTypeOf(Compte)
```

```
        and lesComptes->forAll(c2 | c2 = c or c2.numero ≠  
        c.numero)
```

```
        and lesComptes = lesComptes@pre->including(c)
```

```
        and result = c.numero and c.solde = 15
```

```
        and c.titulaire = p
```

```
endif
```

one () : existe-t-il un et un seul tq ... ?

any () : l'un quelconque de ceux qui ...

Example: Bank (3)

```
context Banque::ouvrirCompte(nomC: String) : Integer
    signal Refus
let lp: Personne.allInstances->select(p| p.nom = nomc)
    p : lp->any(true)          -- indéfini si lp est vide !
in
pre: lp->size()=1
post:
if interdits->includes(p) implies Refus.sent()
else c.oclIsNew() and c.oclIsTypeOf(Compte)
    and lesComptes->forall(c2| c2 = c or c2.numero ≠ c.numero)
    and lesComptes = lesComptes@pre->including(c)
    and compte.titulaire = p
    and result = c.numero and c.solde = 15
endif
```

Examples with @pre

```
context Banque::ajouter(c: Compte) : Integer
pre:  not lesComptes->includes(c)
post: lesComptes=lesComptes@pre->including(c)
```

```
context Banque::ajouter(c: Compte) : Integer
pre:  not lesComptes->includes(c)
post: lesComptes->includes(c) and
        lesComptes@pre->forAll(c2 |
                                lesComptes->includes(c2))
```

👉 **Attention: These versions are not equivalent !**

Miscellaneous - HOL-OCL/su4sml

- OCL is a “programmer oriented” specification language, and as such very verbose.

When proving things, we use a mathematical notation (inspired by the HOL-OCL proof system) for derivations (built over strong equality \equiv):

<code>x and y</code>	\equiv	<code>x ^ y</code>
<code>x or y</code>	\equiv	<code>x v y</code>
<code>x implies y</code>	\equiv	<code>x -> y</code>
<code>not x</code>	\equiv	<code>¬x</code>
<code>x.oclIsDefined()</code>	\equiv	<code>∂x</code>
<code>x.oclIsUndeclared()</code>	\equiv	<code>¬∂x</code>

Miscellaneous - HOL-OCL/su4sml

- ❑ OCL is a “evaluation oriented” specification language, and strictly oriented to computable formulas.

When proving things, we use a mathematical concepts (inspired by the HOL-OCL proof system) like:

```
Integer->forall(x, y | x + y = y + x)
String->forall(x | ...)
Real->forall(x | ...)
```

assuming infinite, unbounded characteristic sets for the basic data types comprising „values“ (in contrast to objects)

Miscellaneous - HOL-OCL/su4sml

- OCL 1.4 is small, but well-defined, while OCL 2 is currently only partially defined.

Vaguely following OCL2.0, and forced by the needs of program verification, we use (inspired by HOL-OCL) the constant

`null`

for a polymorphic value (so, in any class type there exists a `null`). It is defined (can thus be passed as argument), but any access on it `null.left` is undefined.

Miscellaneous - HOL-OCL/su4sml

□ Note:

```
x = null implies x.a.oclIsUndefined()
```

but not vice-versa. There are dangling references, i.e. non-null references whose de-referentiation in a state yields undefined !!!

Compare pp. 24 !

Summary

- ❑ OCL makes the UML to a real, formal specification language
- ❑ OCL can be used in *Class Models* and *State Machines*
- ❑ OCL can be used on the conceptual level, on the design level, and is heavily used to define the UML itself.



- ❑ In the *HOL-OCL* extensions, it has the expressive power of *First-Order Logics (FOL)* with arithmetic.

Les Machines à états

- ❑ Elles permettent de donner une autre vision des classes: comportement dynamique d'une instance prototypique...
 - Dans quel état une méthode est-elle appellable ?
 - Quel est son effet, dans quel état laisse-t-elle l'instance ?
 - Quelles sont toutes les méthodes qui peuvent amener une instance dans un état donné ?
- Permet de vérifier qu'on oublie pas de transitions ou d'états !

Les machines à états (2)

- ❑ État: abstraction de la vie d'un objet pendant laquelle il satisfait une condition, exécute certaines activités et répond à des évènements extérieurs d'une certaine façon.
 - Notion d'état complexe (structuré)
 - Activités internes dans les états
 - Activités à l'entrée et à la sortie d'un état
- ❑ Transitions: relation entre deux états, reflétant généralement un changement de comportement.
Décrit par:

2008-2009  les états de départ et d'arrivée
B. Wolff - L3 - Genie Logiciel - Introduction to  l'évènement déclencheur

Les machines à états (3)

- ❑ Les états peuvent rester implicites dans chaque méthode...
- ❑ ... ou au contraire, on peut programmer l'instance comme un automate
 - `switch` à la Java/C, organisé autour d'une variable qui mémorise l'état courant
- ❑ A mi-chemin entre l'analyse et la conception !

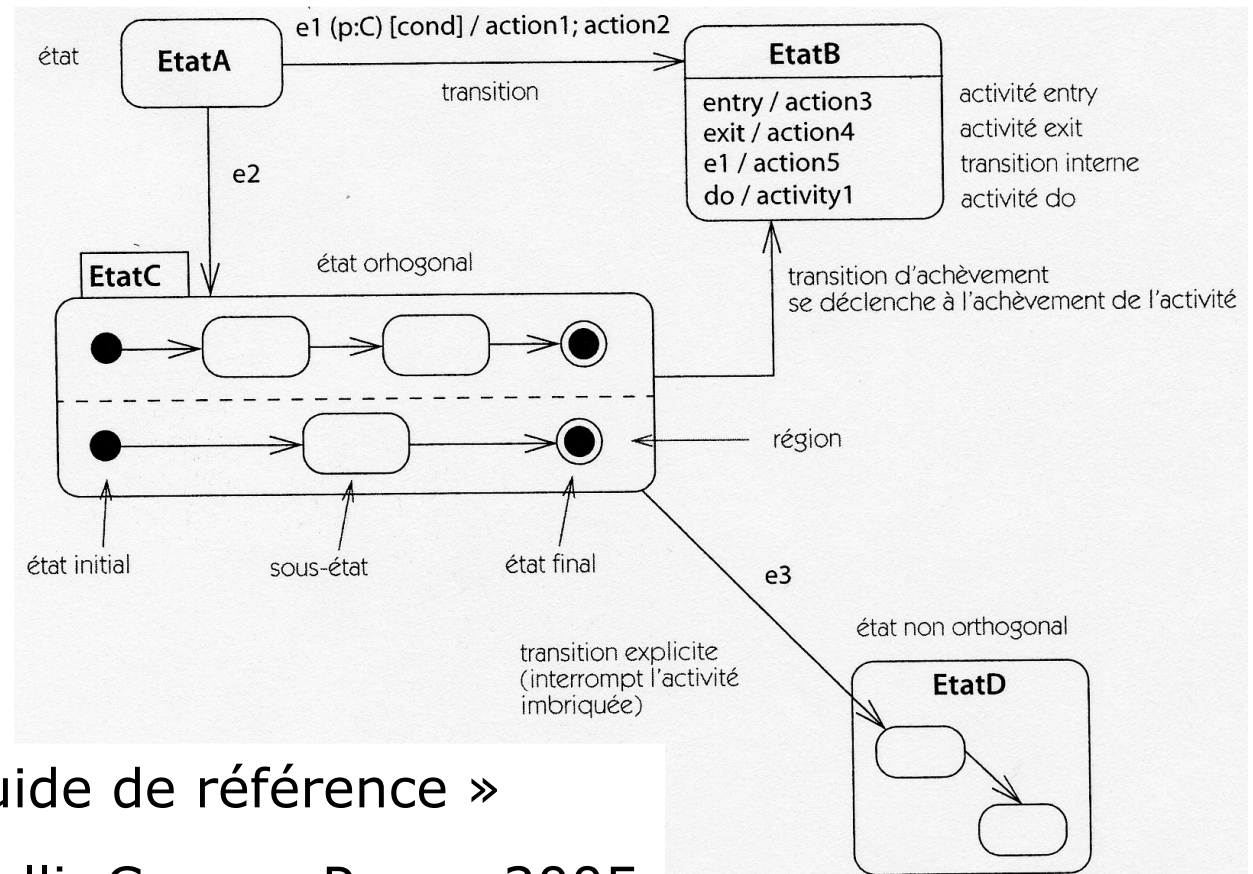
Les machines à états (4)

- ❑ Actions associées à une transition
 - Appel d'une méthode sur une instance
 - Modification d'un attribut
 - Envoi d'un signal à une instance
 - Création ou destruction d'un instance
 - ...

- ❑ Possibilité de « transitions internes »

- ❑ États structurés: on peut décomposer en
 - États orthogonaux (parallèles ↔ facettes indépendantes)
 - États non orthogonaux (séquence de sous-états)

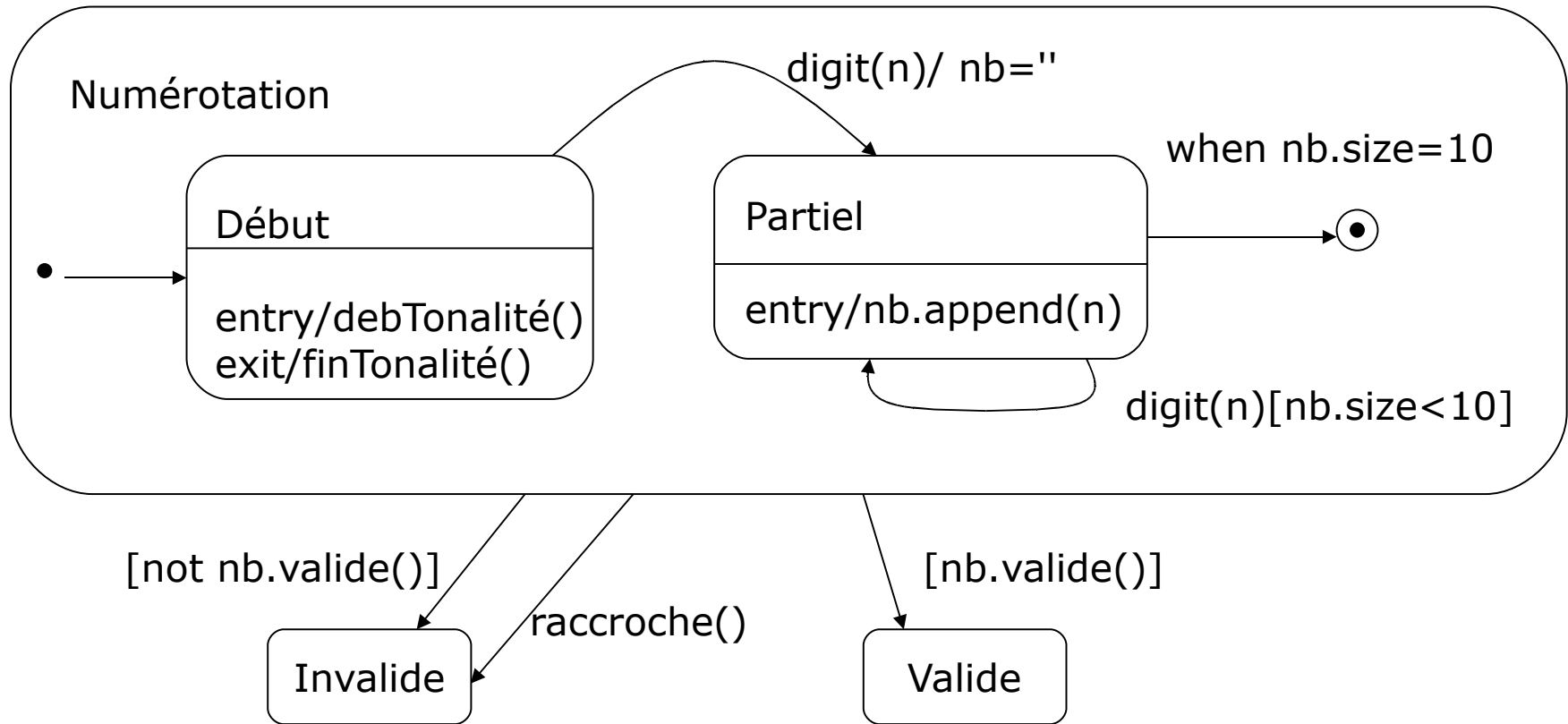
Machines à états : syntaxe étendue



« UML 2.0, Guide de référence »

Rumbaugh & alli, CampusPress, 2005

Machines à états : exemple d'un téléphone



Exercice: modifier la machine à états pour accepter les numéros courts !

Machines à états et OCL

Les états sont traités comme des types énumérés.

`o.oclInState (s)` : retourne vrai si le nom de l'état dans lequel se trouve l'objet `o` est `s`.

Exemple:

Si `M1` est une instance de `Montre` on peut écrire :

```
M1.oclInState (AffichageHeure)
```

OU

```
M1.oclInState (EnModification::EnModificationH)
```

<<énumération>> ETAT-MONTRE

AffichageHeure ModificationHeure ModificationMinute

Contraintes de la montre (1)

context Montre **inv** :

Minutes ≥ 0 and Minutes ≤ 59 **and**

Heures ≥ 0 and Heures ≤ 23

context Montre :: appuyerB ()

pre: true

post : self.oclInState (ModificationHeure) **implies**

Minutes = Minutes@pre and

Heures = Heures@pre + 1 mod 24

and self.oclInState (ModificationMinute) **implies**

Minutes = Minutes@pre + 1 mod 60 and

Heures = Heures@pre

and self.oclInState (AffichageHeure) **implies**

Minutes = Minutes@pre and

Heures = Heures@pre

Contraintes de la montre (2)

```
context Montre :: appuyerA ()  
pre: true  
post :  
    self.oclInState@pre (AffichageHeure) implies  
        self.oclInState (ModificationHeure)  
and  
    self.oclInState@pre (ModificationHeure) implies  
        self.oclInState (ModificationMinutes)  
and  
    self.oclInState@pre (ModificationMinutes) implies  
        self.oclInState (AffichageHeure)
```