POLYTECH
PARIS-SUD

2021

Cycle Ingénieur – 2ème année
Département Informatique

Verification and Validation

Part III : Formal Specification with UML/MOAL

Burkhart Wolff
Département Informatique
Université Paris-Saclay / LMF

Plan of the Chapter

Plan of the Chapter

- Syntax & Semantics of our own language

Plan of the Chapter

- Syntax & Semantics of our own language

MOAL

Plan of the Chapter

- Syntax & Semantics of our own language

MOAL

  - mathematical

Plan of the Chapter

- Syntax & Semantics of our own language

MOAL

  - mathematical
  - object-oriented

Plan of the Chapter

- Syntax & Semantics of our own language

MOAL

- mathematical
- object-oriented
- UML-annotation

Plan of the Chapter

- Syntax & Semantics of our own language

MOAL

- mathematical
  - object-oriented
  - UML-annotation
  - language

(conceived as the „essence" of annotation
languages like OCL, JML, Spec#, ACSL, ...)

Plan of the Chapter

Plan of the Chapter

- Concepts of MOAL

Plan of the Chapter

- Concepts of MOAL
  - Basis: Logic and Set-theory

Plan of the Chapter

- Concepts of MOAL
  - Basis: Logic and Set-theory
  - MOAL is a Typed Language

Plan of the Chapter

- Concepts of MOAL
  - Basis: Logic and Set-theory
  - MOAL is a Typed Language
  - Basic Types, Sets, Pairs and Lists

Plan of the Chapter

- Concepts of MOAL
  - Basis: Logic and Set-theory
  - MOAL is a Typed Language
  - Basic Types, Sets, Pairs and Lists
  - Object Types from UML

Plan of the Chapter

- Concepts of MOAL
  - Basis: Logic and Set-theory
  - MOAL is a Typed Language
  - Basic Types, Sets, Pairs and Lists
  - Object Types from UML
  - Navigation along UML attributes and associations

(Idea from OCL and JML)

Plan of the Chapter

- Concepts of MOAL
  - Basis: Logic and Set-theory
  - MOAL is a Typed Language
  - Basic Types, Sets, Pairs and Lists
  - Object Types from UML
  - Navigation along UML attributes and associations

    (Idea from OCL and JML)
- Purpose :

Plan of the Chapter

- Concepts of MOAL
  - Basis: Logic and Set-theory
  - MOAL is a Typed Language
  - Basic Types, Sets, Pairs and Lists
  - Object Types from UML
  - Navigation along UML attributes and associations
    (Idea from OCL and JML)
- Purpose :
  - Class Invariants

Plan of the Chapter

- Concepts of MOAL
  - Basis: Logic and Set-theory
  - MOAL is a Typed Language
  - Basic Types, Sets, Pairs and Lists
  - Object Types from UML
  - Navigation along UML attributes and associations
    (Idea from OCL and JML)
- Purpose :
  - Class Invariants
  - Method Contracts with Pre- and Post-Conditions

Plan of the Chapter

- Concepts of MOAL
  - Basis: Logic and Set-theory
  - MOAL is a Typed Language
  - Basic Types, Sets, Pairs and Lists
  - Object Types from UML
  - Navigation along UML attributes and associations (Idea from OCL and JML)
- Purpose :
  - Class Invariants
  - Method Contracts with Pre- and Post-Conditions
  - Annotated Sequence Diagrams for Scenarios, . . .

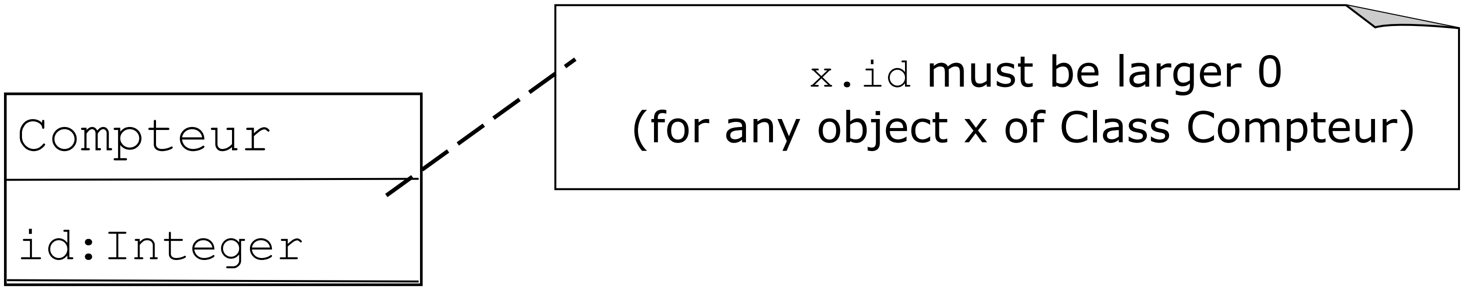
Motivation: Why Logical Annotations

Motivation: Why Logical Annotations

- More precision needed
  (like JML, VCC) that constrains an underlying state σ

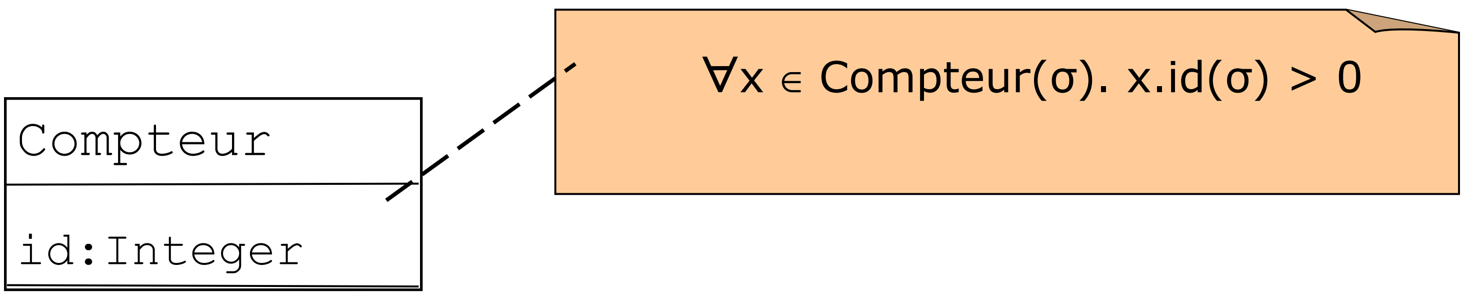
Motivation: Why Logical Annotations

- More precision needed
  (like JML, VCC) that constrains an underlying state σ



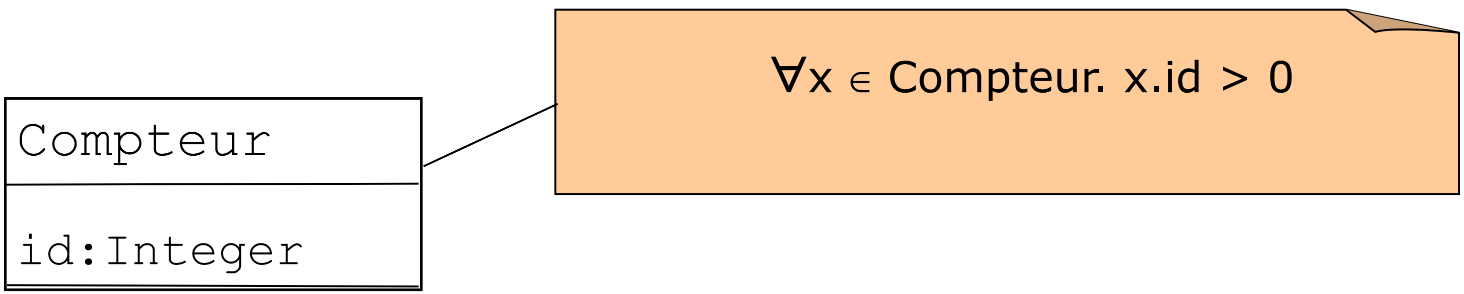
Motivation: Why Logical Annotations

- More precision needed
  (like JML, VCC) that constrains an underlying state σ



Motivation: Why Logical Annotations

- More precision needed
  (like JML, VCC) that constrains an underlying state σ



... by abbreviation convention if no confusion arises.

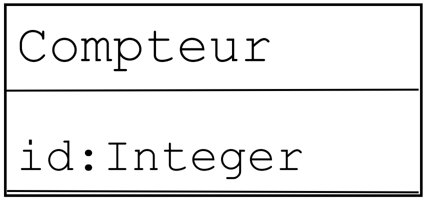
Motivation: Why Logical Annotations

Motivation: Why Logical Annotations

- More precision needed
  (like JML, VCC) that constrains an underlying state σ

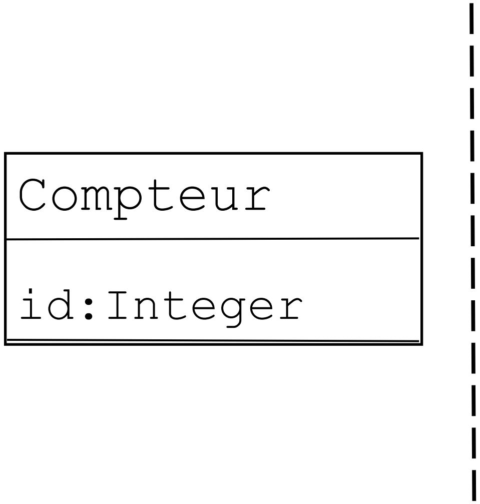
Motivation: Why Logical Annotations

- More precision needed
  (like JML, VCC) that constrains an underlying state σ



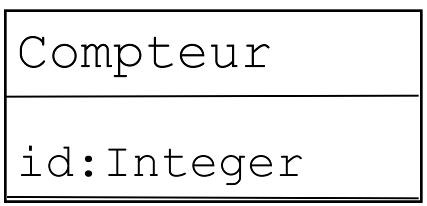
Motivation: Why Logical Annotations

- More precision needed
  (like JML, VCC) that constrains an underlying state σ



Motivation: Why Logical Annotations

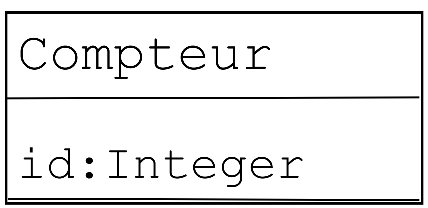
- More precision needed
  (like JML, VCC) that constrains an underlying state σ



definition $inv_{Compteur}(\sigma) \equiv \forall x \in Compteur(\sigma).$
$x.id(\sigma) > 0$

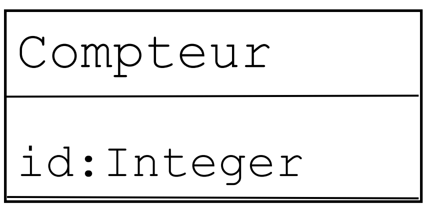
Motivation: Why Logical Annotations

- More precision needed
  (like JML, VCC) that constrains an underlying state σ



Motivation: Why Logical Annotations

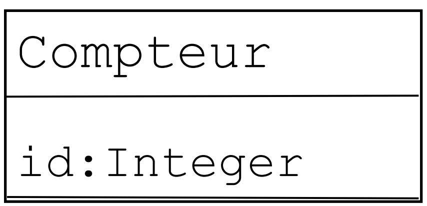
- More precision needed
  (like JML, VCC) that constrains an underlying state σ



... or by convention

Motivation: Why Logical Annotations

- More precision needed
  (like JML, VCC) that constrains an underlying state σ

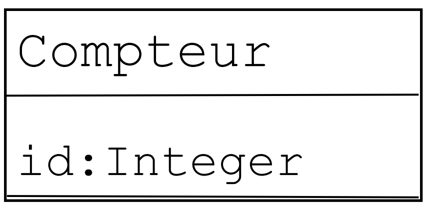


... or by convention

definition $inv_{Compteur} \equiv \forall x \in Compteur.\ x.id > 0$

Motivation: Why Logical Annotations

- More precision needed
  (like JML, VCC) that constrains an underlying state σ



... or by convention

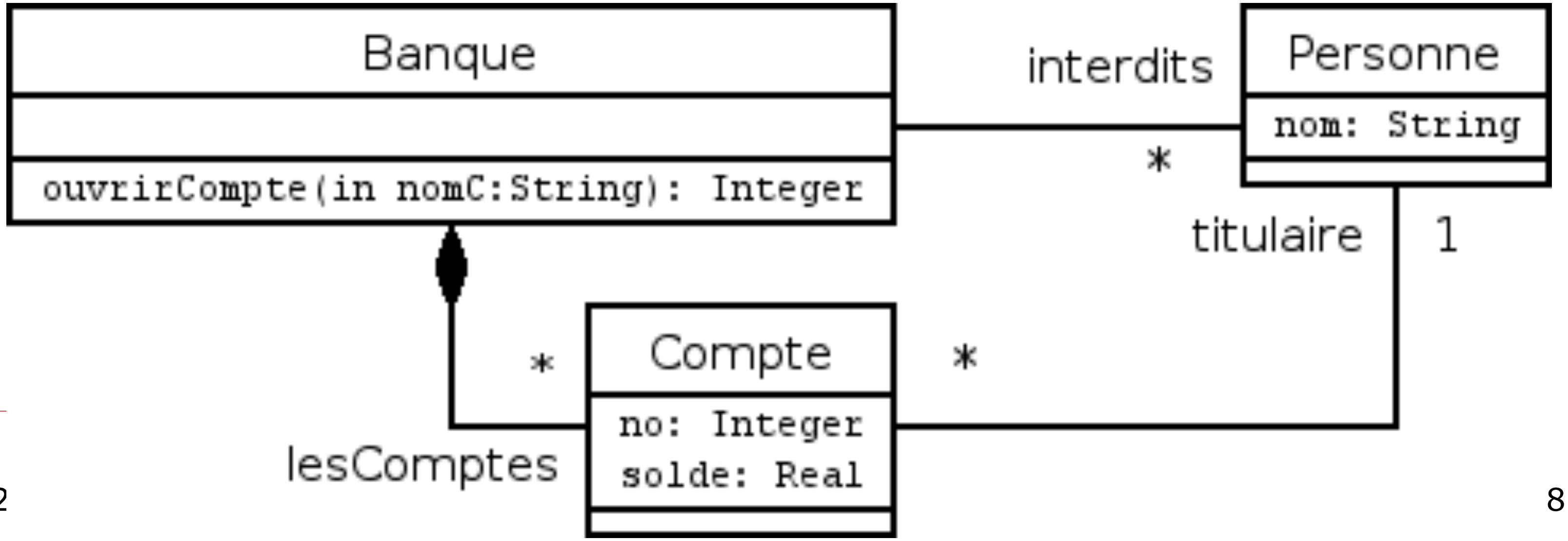
definition $inv_{Compteur} \equiv \forall x \in Compteur.\ x.id > 0$

... or as mathematical definition in a separate document

A first Glance to an Example: Bank

Opening a bank account. Constraints:

- there is a blacklist
- no more overdraft than 200 EUR
- there is a present of 15 euros in the initial account
- account numbers must be distinct.



A first Glance to an Example: Bank (2)

definition unique ≡ isUnique(.no)(Compte)

definition noOverdraft ≡ $\forall$c ∈ Compte. c.id ≥ -200

definition $pre_{ouvrirCompte}$(b:Banque, nomC:String)≡
$\forall$p ∈ Personne. p.nom ≠ nomC

definition $post_{ouvrirCompte}$(b:Banque,nomC:String,r::Int)≡
|{p ∈ Personne | p.nom = nomC ∧ isNew(p)}| = 1
∧ |{c∈Compte | c.titulaire.nom = nomC}| = 1
∧ $\forall$c∈Compte. c.titulaire.nom = nomC
⟶ c.solde = 15 ∧ isNew(c)

MOAL: a specification langage?

- In the following, we will discuss the MOAL Language in more detail ...

Syntax and Semantics of MOAL

Syntax and Semantics of MOAL

- The usual logical language:

Syntax and Semantics of MOAL

- The usual logical language:
  - `True, False`

Syntax and Semantics of MOAL

- The usual logical language:
  - True, False
  - negation : ¬ E,

Syntax and Semantics of MOAL

- The usual logical language:
  - True, False
  - negation : $\neg E$,
  - or: $E \vee E'$, and: $E \wedge E'$, implies: $E \longrightarrow E'$

Syntax and Semantics of MOAL

- The usual logical language:
  - True, False
  - negation : $\neg E$,
  - or: $E \vee E'$, and: $E \wedge E'$, implies: $E \longrightarrow E'$
  - $E = E'$, $E \neq E'$,

Syntax and Semantics of MOAL

- The usual logical language:
  - True, False
  - negation : $\neg\ E$,
  - or: $E \lor E'$, and: $E \land E'$, implies: $E \longrightarrow E'$
  - $E = E'$, $E \neq E'$,
  - if C then E else E' endif

Syntax and Semantics of MOAL

- The usual logical language:
  - True, False
  - negation : $\neg E$,
  - or: $E \vee E'$, and: $E \wedge E'$, implies: $E \longrightarrow E'$
  - $E = E'$, $E \neq E'$,
  - if C then E else E' endif
  - let x = E in E′

Syntax and Semantics of MOAL

- The usual logical language:
  - True, False
  - negation : $\neg E$,
  - or: $E \vee E'$, and: $E \wedge E'$, implies: $E \longrightarrow E'$
  - $E = E'$, $E \neq E'$,
  - if C then E else E' endif
  - let x = E in E'

  - Quantifiers on sets and lists:

$\forall x \in Set.\ P(x)$ $\exists x \in Set.\ P(x)$

Syntax and Semantics of MOAL

Syntax and Semantics of MOAL

- MOAL is (like OCL or JML) a typed language.

Syntax and Semantics of MOAL

- MOAL is (like OCL or JML) a typed language.
  - Basic Types:

    Boolean, Integer, Real, String

Syntax and Semantics of MOAL

- MOAL is (like OCL or JML) a typed language.
  - Basic Types:

    Boolean, Integer, Real, String

  - Pairs:

    $X \times Y$

Syntax and Semantics of MOAL

- MOAL is (like OCL or JML) a typed language.
  - Basic Types:

    Boolean, Integer, Real, String
  - Pairs:

    $X \times Y$
  - Lists:

    List(X)

Syntax and Semantics of MOAL

- MOAL is (like OCL or JML) a typed language.
  - Basic Types:

    Boolean, Integer, Real, String
  - Pairs:

    $X \times Y$
  - Lists:

    List(X)
  - Sets:

    Set(X)

Syntax and Semantics of MOAL

Syntax and Semantics of MOAL

- The arithmetic core language.
  expressions of type `Integer` or `Real`:

Syntax and Semantics of MOAL

- The arithmetic core language.
  expressions of type `Integer` or `Real`:
  - `1,2,3 ...   resp. 1.0, 2.3, pi.`

Syntax and Semantics of MOAL

- The arithmetic core language.
  expressions of type `Integer` or `Real`:
  - `1,2,3 ...    resp. 1.0, 2.3, pi.`
  - `-` E, E `+` E',

Syntax and Semantics of MOAL

- The arithmetic core language.
  expressions of type `Integer` or `Real`:
  - `1,2,3 ...   resp. 1.0, 2.3, pi.`
  - `-` *E*, *E* `+` *E*',
  - *E* `*` *E*', *E* `/` *E*',

Syntax and Semantics of MOAL

- The arithmetic core language.
  expressions of type `Integer` or `Real`:
  - `1,2,3 ...   resp. 1.0, 2.3, pi.`
  - `- E, E + E',`
  - `E * E', E / E',`
  - `abs(E), E div E', E mod E'...`

Syntax and Semantics of MOAL

Syntax and Semantics of MOAL

- The expressions of type `String`:

Syntax and Semantics of MOAL

- The expressions of type `String`:
  - *S* `concat S'`

Syntax and Semantics of MOAL

- The expressions of type `String`:
  - `S concat S'`
  - `size(S)`

Syntax and Semantics of MOAL

- The expressions of type `String`:
  - *S* `concat S'`
  - *size(S)*
  - `substring(i,j,S)`

Syntax and Semantics of MOAL

- The expressions of type `String`:
  - *S* `concat S'`
  - *size(S)*
  - `substring(i,j,S)`
  - `'Hello'`

Syntax and Semantics of MOAL Sets

Syntax and Semantics of MOAL Sets

- `| S |`   `size as Integer`

Syntax and Semantics of MOAL Sets

- $|\ S\ |$ size as Integer
- isUnique(f)(S) $\equiv \forall x,y \in S.\ f(x)=f(y) \longrightarrow x=y$

Syntax and Semantics of MOAL Sets

- | S | size as Integer
- isUnique(*f*)(S) ≡ ∀x,y ∈ S. f(x)=f(y) ⟶ x=y
- {}, {a,b,c} *empty and finite sets*

Syntax and Semantics of MOAL Sets

- | S | size as Integer
- isUnique(*f*)(S) $\equiv \forall x,y \in S.\ f(x)=f(y) \longrightarrow x=y$
- {}, {a,b,c} *empty and finite sets*
- e∈S, e∉S is element, not element

Syntax and Semantics of MOAL Sets

- | S | size as Integer
- isUnique(*f*)(S) ≡ $\forall x,y \in S.\ f(x)=f(y) \longrightarrow x=y$
- {}, {a,b,c} *empty and finite sets*
- $e \in S$, $e \notin S$ is element, not element
- $S \subseteq S'$ is subset

Syntax and Semantics of MOAL Sets

- $|\ S\ |$ size as Integer
- $\mathrm{isUnique}(f)(S) \equiv \forall x,y \in S.\ f(x)=f(y) \longrightarrow x=y$
- {}, {a,b,c} *empty and finite sets*
- $e \in S$, $e \notin S$ is element, not element
- $S \subseteq S'$ is subset
- $\{x \in S \mid P(S)\}$ filter

Syntax and Semantics of MOAL Sets

- | S | size as Integer
- isUnique(*f*)(S) ≡ ∀x,y ∈ S. f(x)=f(y) ⟶ x=y
- {}, {a,b,c} *empty and finite sets*
- e∈S, e∉S is element, not element
- S⊆S' is subset
- {x ∈ S | P(S)} filter
- S ∪ S',S ∩ S' union , intersect between sets of same type

Syntax and Semantics of MOAL Sets

- | S | — size as Integer
- isUnique(*f*)(S) ≡ ∀x,y ∈ S. f(x)=f(y) ⟶ x=y
- {}, {a,b,c} — *empty and finite sets*
- e∈S, e∉S — is element, not element
- S⊆S' — is subset
- {x ∈ S | P(S)} — filter
- S ∪ S', S ∩ S' — union , intersect between sets of same type
- Integer, Real, String ... are symbols for the set of all Integers, Reals,

Syntax and Semantics of MOAL Pairs

Syntax and Semantics of MOAL Pairs

- `(X,Y)` `pairing`

Syntax and Semantics of MOAL Pairs

- `(X,Y)` pairing
- `fst(X,Y) = X` projection

Syntax and Semantics of MOAL Pairs

- `(X,Y)` pairing
- `fst(X,Y) = X` projection
- `snd(X,Y) = Y` projection

Syntax and Semantics of MOAL Lists

Syntax and Semantics of MOAL Lists

Lists S have the following operations:

Syntax and Semantics of MOAL Lists

Lists S have the following operations:

- `x ∈ L` -- is element (overload!)

Syntax and Semantics of MOAL Lists

Lists S have the following operations:

- x ∈ L -- is element (overload!)
- |S| -- length as Integer

Syntax and Semantics of MOAL Lists

Lists *S* have the following operations:

- `x ∈ L` -- is element (overload!)
- `|S|` -- length as Integer
- `head(L),last(L)`

Syntax and Semantics of MOAL Lists

Lists *S* have the following operations:

- `x ∈ L` -- is element (overload!)
- `|S|` -- length as Integer
- `head(L),last(L)`
- *nth*(L, *i*) -- for `i` between `0` et `|S|-1`

Syntax and Semantics of MOAL Lists

Lists *S* have the following operations:

- `x ∈ L` -- is element (overload!)
- `|S|` -- length as Integer
- `head(L),last(L)`
- *nth*(L, *i*) -- for `i` between `0` et `|S|-1`
- *L@L'* -- concatenate

Syntax and Semantics of MOAL Lists

Lists *S* have the following operations:

- `x ∈ L` -- is element (overload!)
- `|S|` -- length as Integer
- `head(L),last(L)`
- *nth*(L,*i*) -- for `i` between `0` et `|S|-1`
- *L@L'* -- concatenate
- *e#S* -- append at the beginning

Syntax and Semantics of MOAL Lists

Lists *S* have the following operations:

- `x ∈ L` -- is element (overload!)
- `|S|` -- length as Integer
- `head(L),last(L)`
- *nth*`(L,`*i*`)` -- for `i` between `0` et `|S|-1`
- *L@L'* -- concatenate
- *e#S* -- append at the beginning
- $\forall_{x \in List}$. `P(x)` -- quantifiers :

Syntax and Semantics of MOAL Lists

Lists S have the following operations:

- `x ∈ L` -- is element (overload!)
- `|S|` -- length as Integer
- `head(L),last(L)`
- *nth*(L, *i*) -- for `i` between `0` et `|S|-1`
- *L@L'* -- concatenate
- *e#S* -- append at the beginning
- ∀ `x∈List. P(x)` -- quantifiers :
- `[x∈L | P(x)]` -- filter

Syntax and Semantics of MOAL Lists

Lists *S* have the following operations:

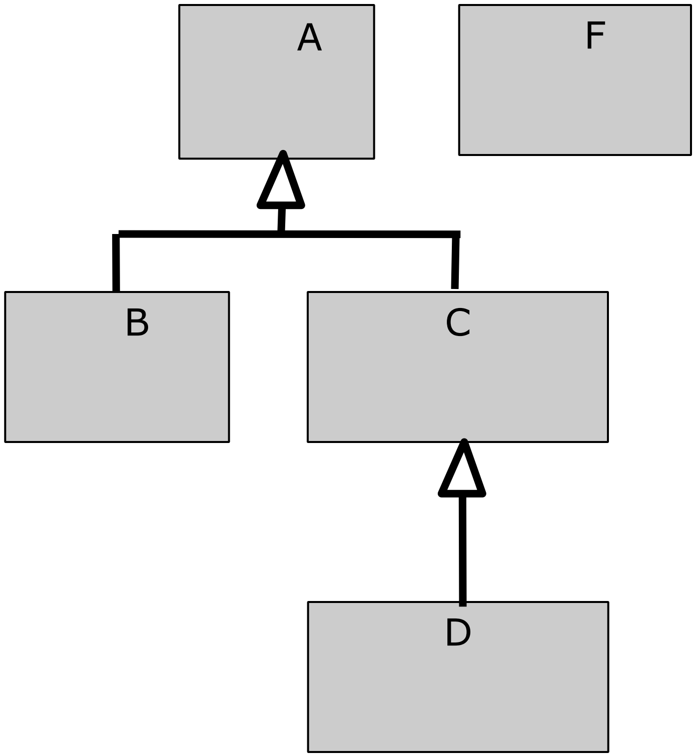
- x ∈ L -- is element (overload!)
- |S| -- length as Integer
- head(L),last(L)
- *nth*(L,*i*) -- for i between 0 et |S|-1
- *L@L'* -- concatenate
- *e#S* -- append at the beginning
- $\forall_{x \in List.}$ P(x) -- quantifiers :
- [x∈L | P(x)] -- filter
- [1,2,3] -- denotations of lists

Syntax and Semantics of Objects

- Objects and Classes follow the semantics of UML
  - inheritance / subtyping
  - casting
  - objects have an id
  - NULL is a possible value in each class-type
  - for any class A, we assume a function:

    $$A(\sigma)$$

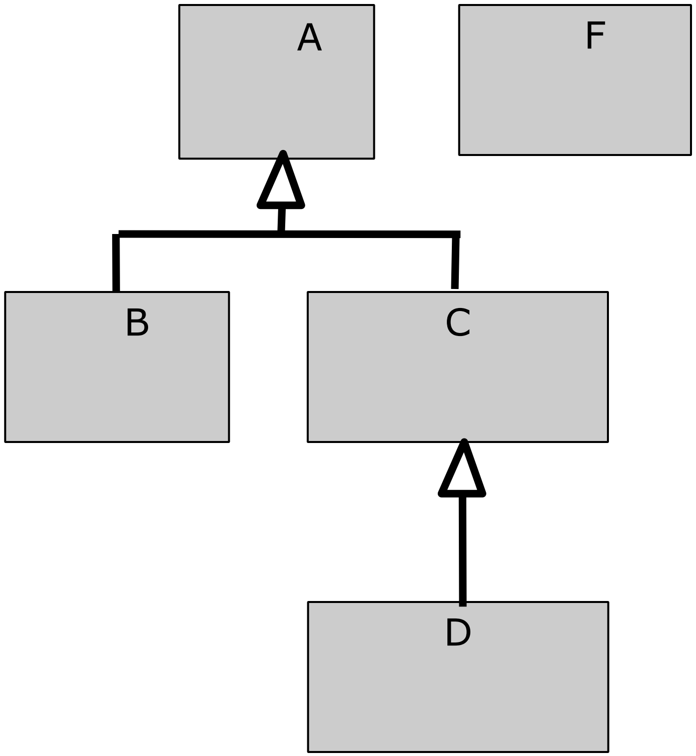
    which returns the set of instances of class A in state σ



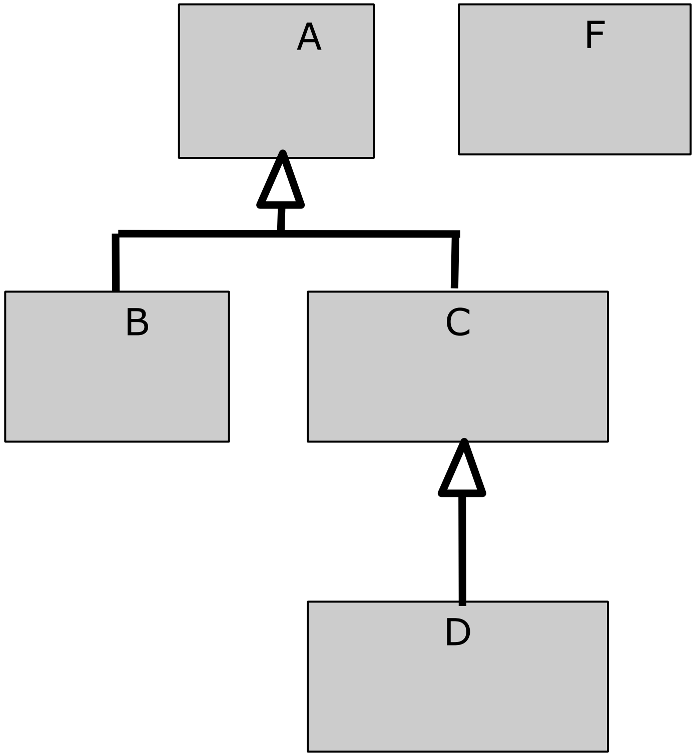
Syntax and Semantics of Objects

- Objects and Classes follow the semantics of UML

  Recall that we will drop the index (σ) whenever it is clear from the context

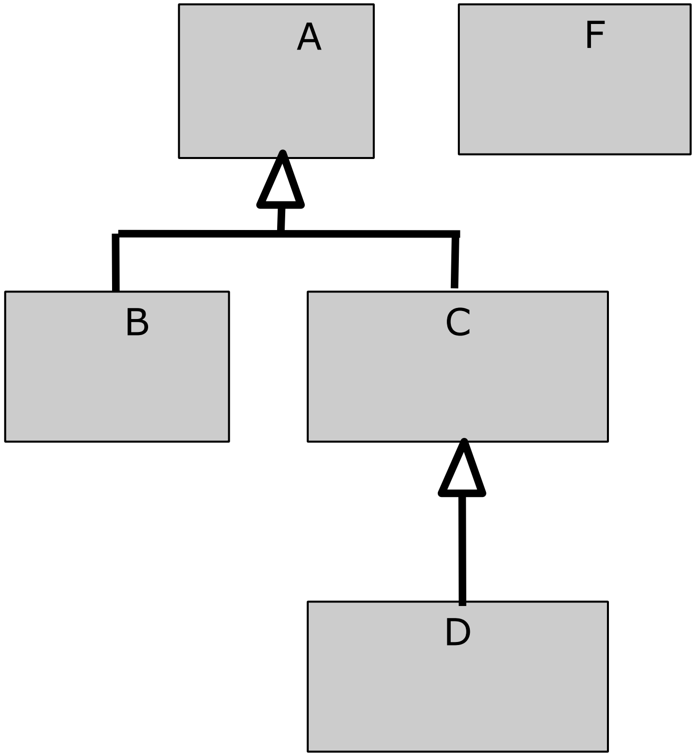


Syntax and Semantics of Objects



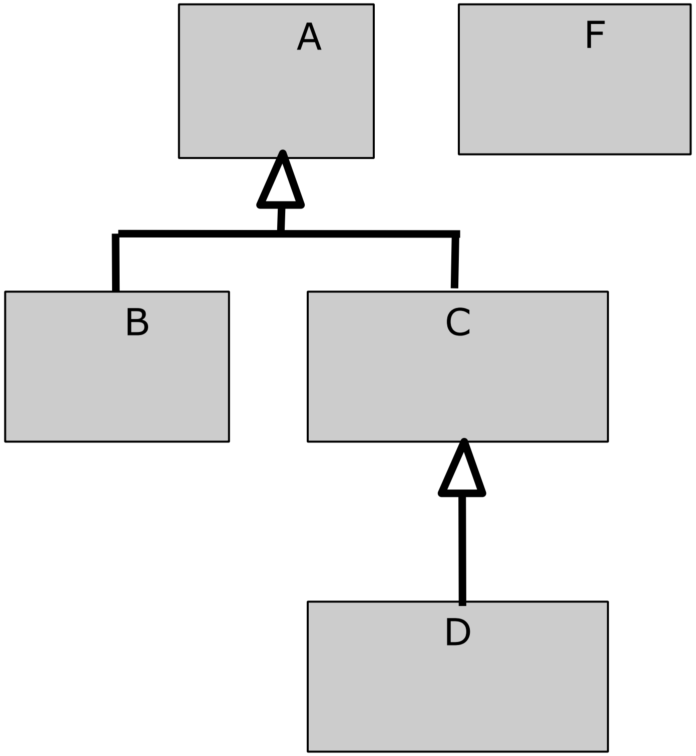
Syntax and Semantics of Objects

- As in all typed object-oriented languages casting allows for converting objects.



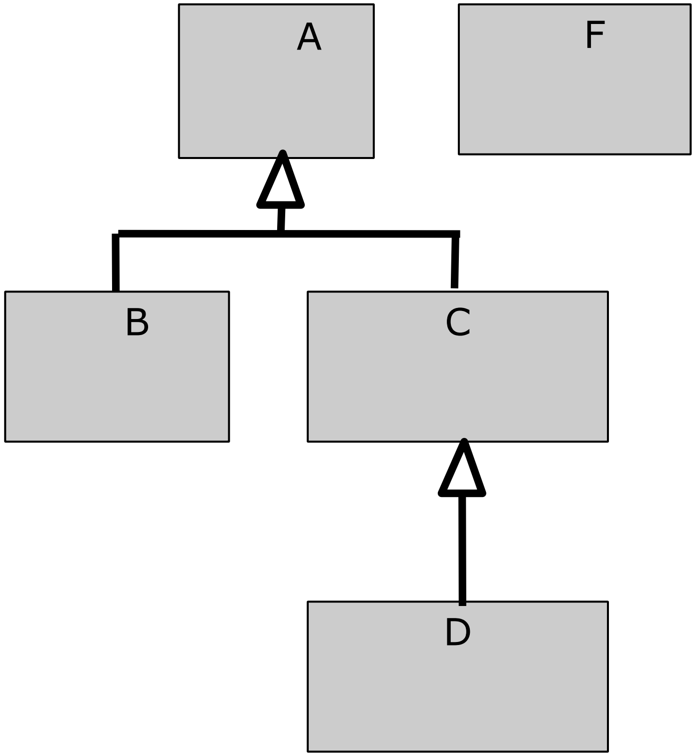
Syntax and Semantics of Objects

- As in all typed object-oriented languages casting allows for converting objects.
- Objects have two types:



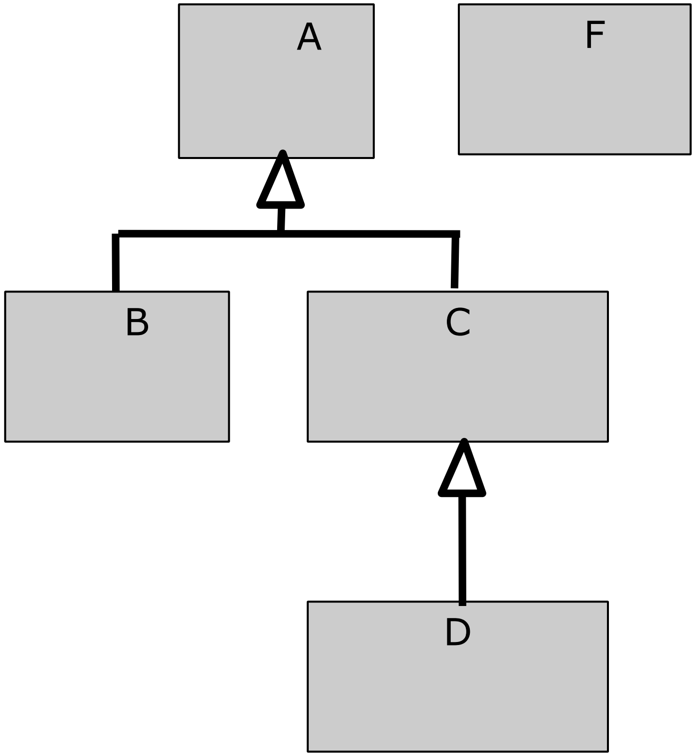
Syntax and Semantics of Objects

- As in all typed object-oriented languages casting allows for converting objects.
- Objects have two types:
  - the « apparent type » (also called static type)



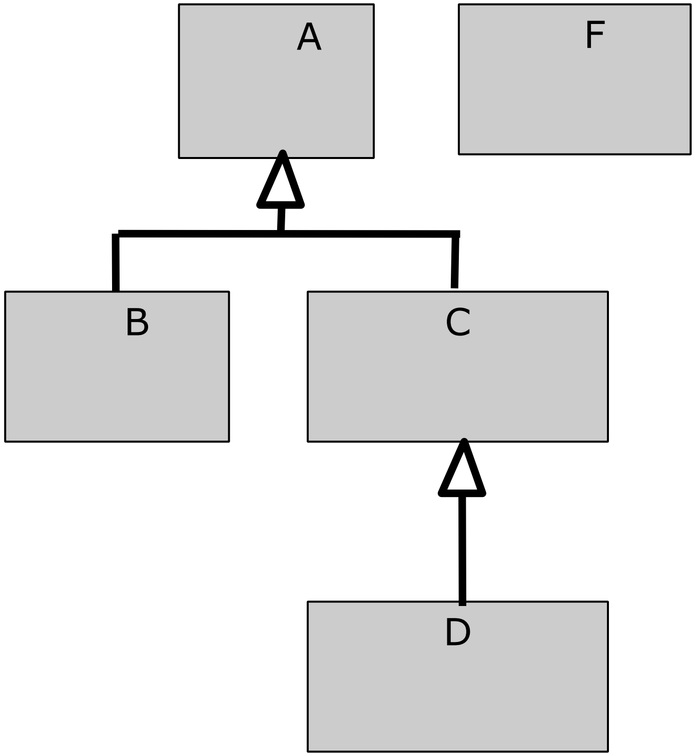
Syntax and Semantics of Objects

- As in all typed object-oriented languages casting allows for converting objects.
- Objects have two types:
  - the « apparent type » (also called static type)
  - the « actual type »



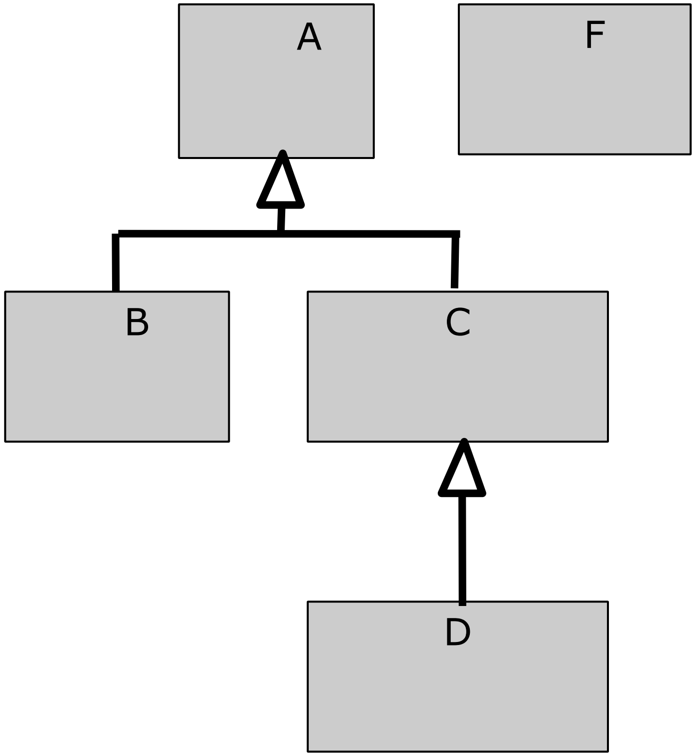
Syntax and Semantics of Objects

- As in all typed object-oriented languages casting allows for converting objects.
- Objects have two types:
  - the « apparent type » (also called static type)
  - the « actual type » (the type at creation)

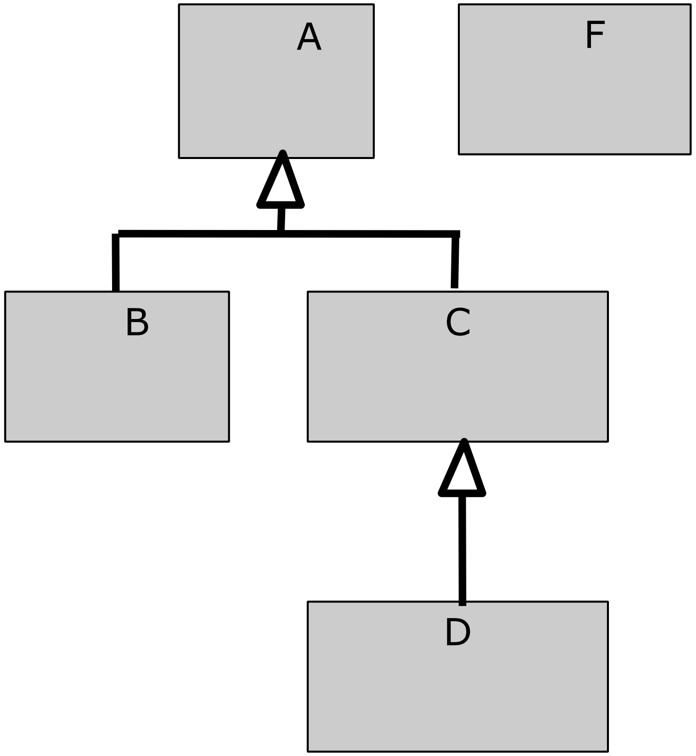


Syntax and Semantics of Objects

- As in all typed object-oriented languages casting allows for converting objects.
- Objects have two types:
  - the « apparent type » (also called static type)
  - the « actual type » (the type at creation)
  - casting changes the apparent type along the class hierarchy, but not the actual type



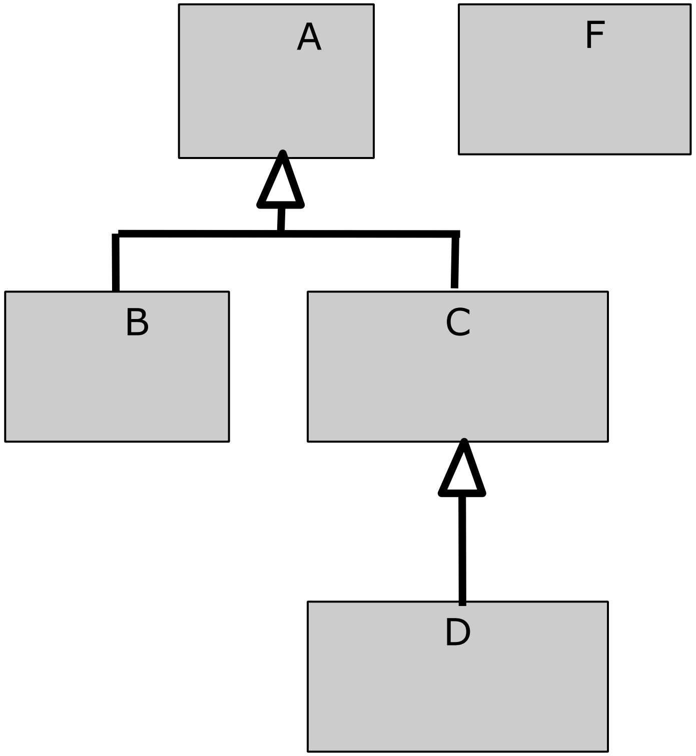
Syntax and Semantics of Objects



Syntax and Semantics of Objects

- Assume the creation of objects

  `a in class A,b in class B,`
  `c in class C,d in class D,`



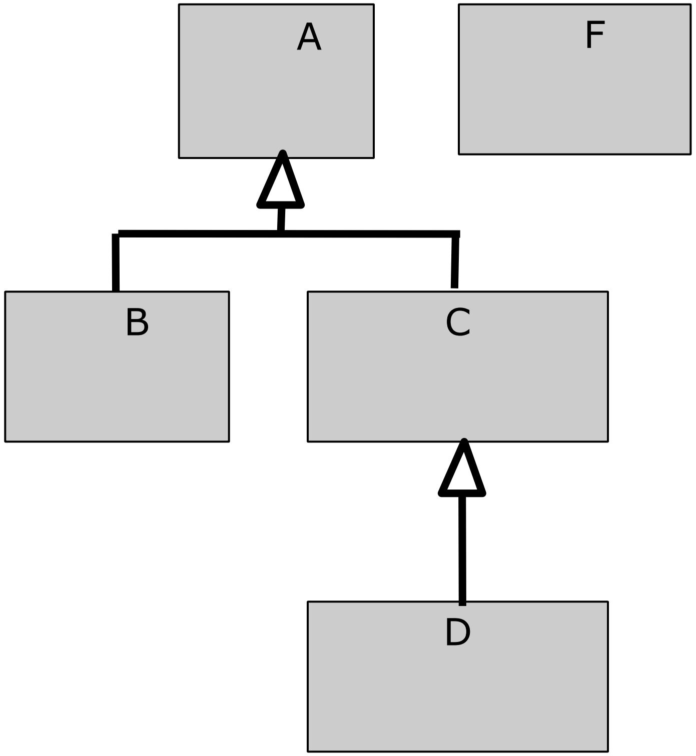
Syntax and Semantics of Objects

- Assume the creation of objects

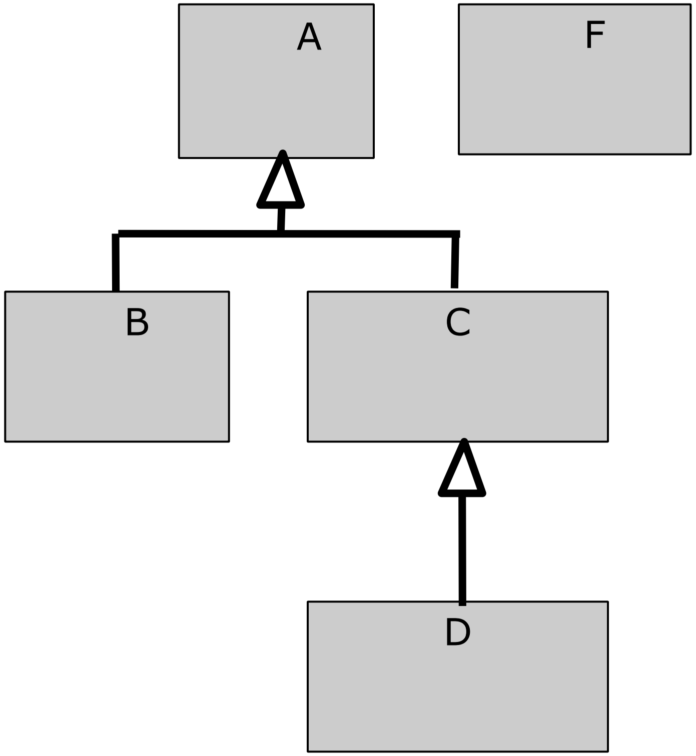
  `a in class A,b in class B,`
  `c in class C,d in class D,`

- Then casting:

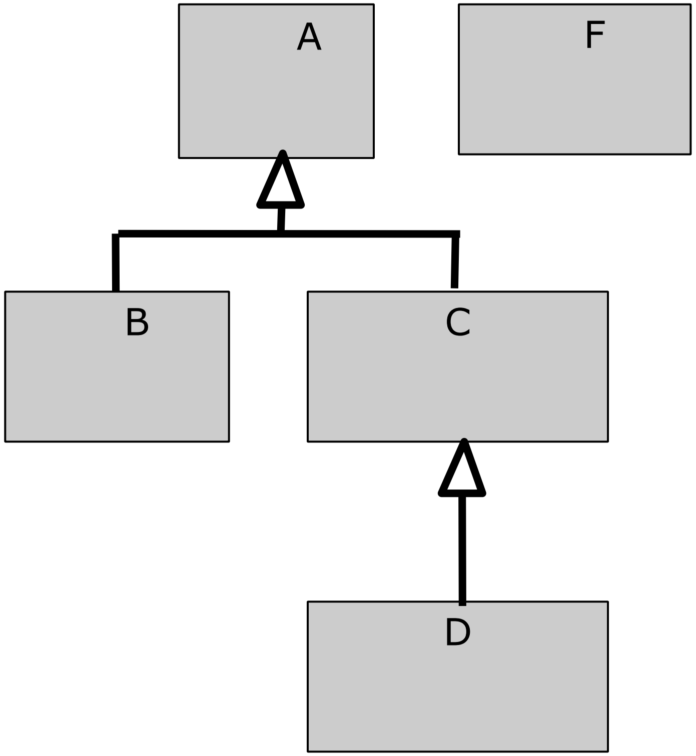
```
⟨F⟩b is illtyped

⟨A⟩b has apparent type A,
    but actual type B

⟨A⟩d has apparent type A,
    but actual type D
```



Syntax and Semantics of OCL / UML

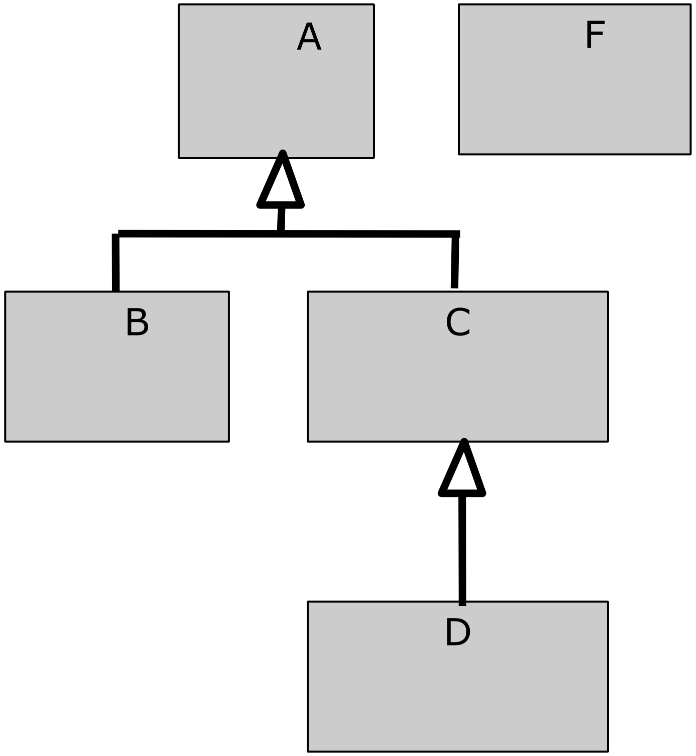


Syntax and Semantics of OCL / UML



Syntax and Semantics of OCL / UML

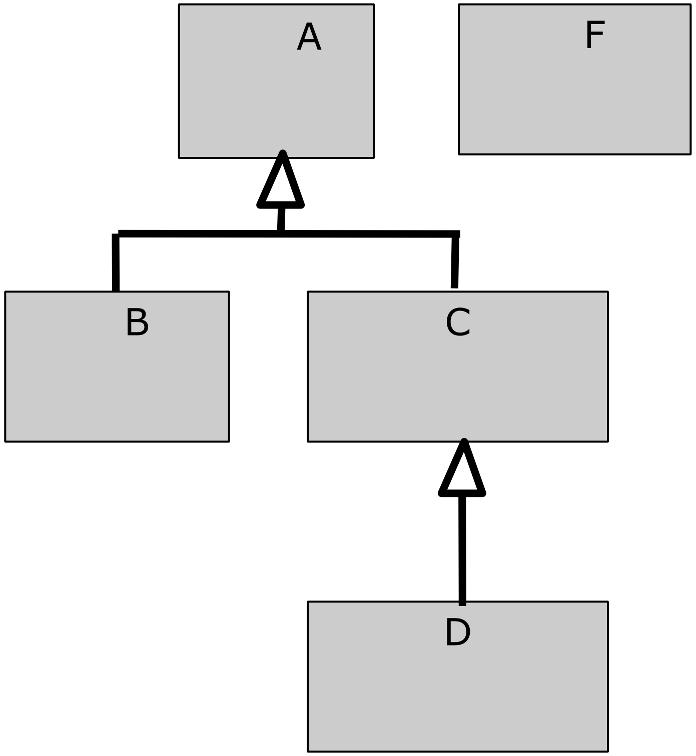
- We will also apply cast-operators to an entire set: So



Syntax and Semantics of OCL / UML

- We will also apply cast-operators to an entire set: So

  ⟨A⟩B(σ) (or just: ⟨A⟩B)

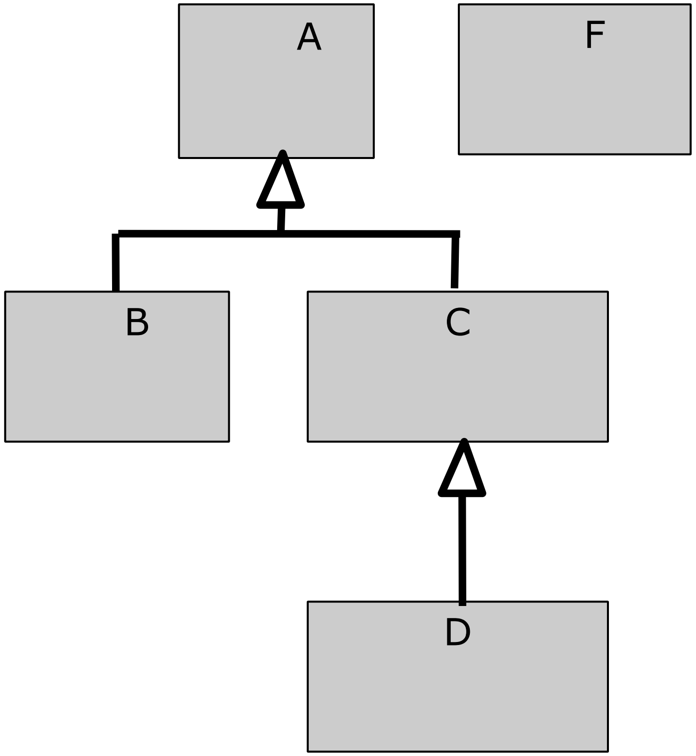


Syntax and Semantics of OCL / UML

- We will also apply cast-operators to an entire set: So

  `⟨A⟩B(σ) (or just: ⟨A⟩B)`

  - is the set of instances of B casted to A.



Syntax and Semantics of OCL / UML

- We will also apply cast-operators to an entire set: So

  ⟨A⟩B(σ) (or just: ⟨A⟩B)

  - is the set of instances of B casted to A.

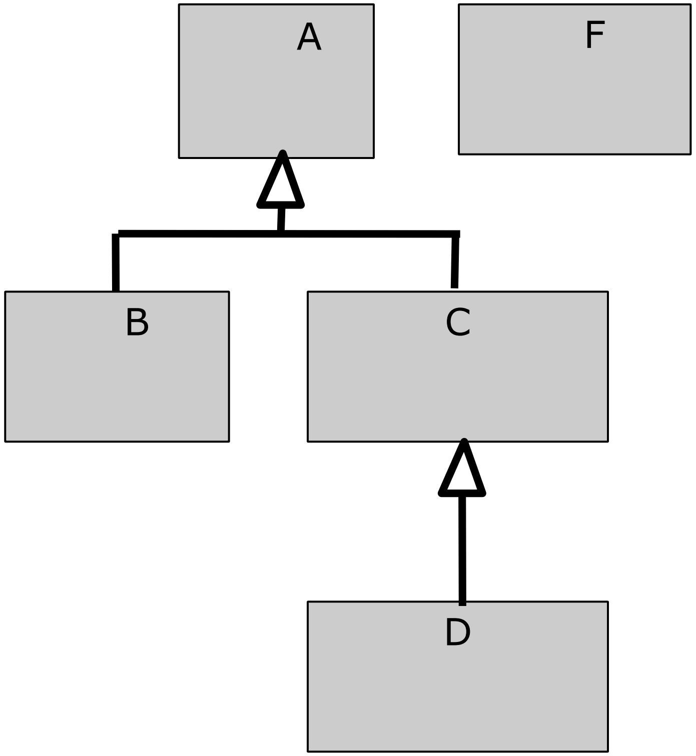
  - We have:

    $\langle A \rangle B \cup \langle A \rangle C \subseteq A$

    but:

    $\langle A \rangle B \cap \langle A \rangle C = \{\}$

    and also: $\langle A \rangle D \subseteq A$ (for all states σ)



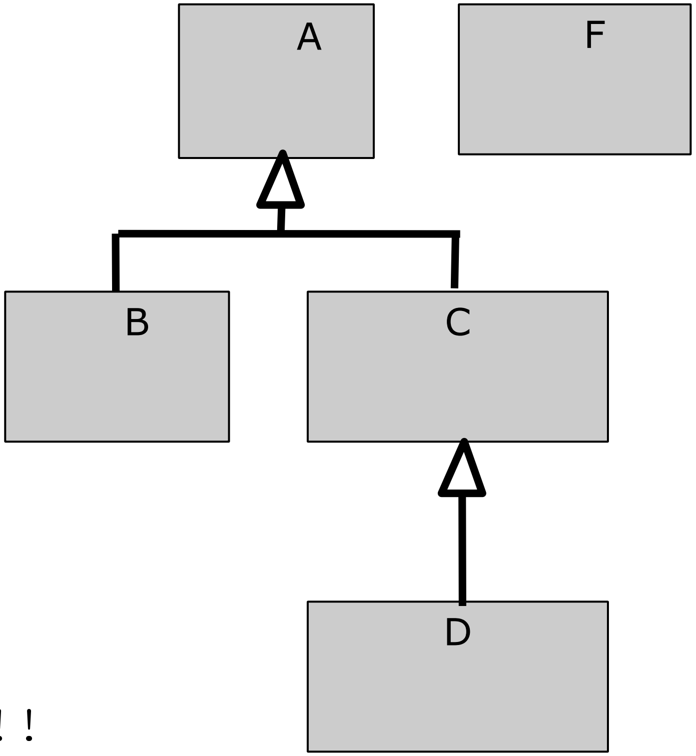
Syntax and Semantics of Objects

- Instance sets can be used to determine the actual type of an object:

  $b \in B$

  corresponds to Java's `instanceof` or `OCL's isKindOf`. Note that casting does NOT change the actual type:

  $\langle A \rangle b \in B$, and $\langle B \rangle \langle A \rangle b = b$!!!



Syntax and Semantics of Objects

Syntax and Semantics of Objects

- Summary:

Syntax and Semantics of Objects

- Summary:
  - there is the concept of **actual** and **apparent** type (anywhere outside of Java: **dynamic** and **static** type)

Syntax and Semantics of Objects

- Summary:
  - there is the concept of **actual** and **apparent** type (anywhere outside of Java: **dynamic** and **static** type)
  - type tests check the former

Syntax and Semantics of Objects

- Summary:
  - there is the concept of **actual** and **apparent** type (anywhere outside of Java: **dynamic** and **static** type)
  - type tests check the former
  - type casts influence the latter,

Syntax and Semantics of Objects

- Summary:
  - there is the concept of **actual** and **apparent** type (anywhere outside of Java: **dynamic** and **static** type)
  - type tests check the former
  - type casts influence the latter,

    but not the former

Syntax and Semantics of Objects

- Summary:
  - there is the concept of **actual** and **apparent** type (anywhere outside of Java: **dynamic** and **static** type)
  - type tests check the former
  - type casts influence the latter, but not the former
  - up-casts possible

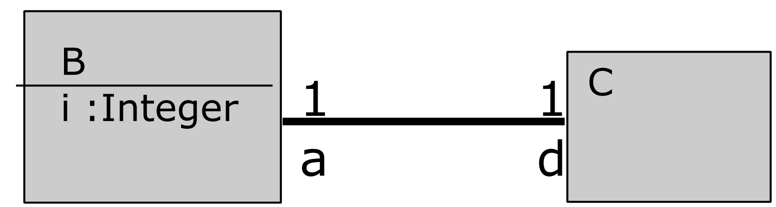
Syntax and Semantics of Objects

- Summary:
  - there is the concept of **actual** and **apparent** type (anywhere outside of Java: **dynamic** and **static** type)
  - type tests check the former
  - type casts influence the latter, but not the former
  - up-casts possible
  - down-casts invalid

Syntax and Semantics of Objects

- Summary:
  - there is the concept of **actual** and **apparent** type
    (anywhere outside of Java: **dynamic** and **static** type)
  - type tests check the former
  - type casts influence the latter,
    but not the former
  - up-casts possible
  - down-casts invalid
  - consequence:
    up-down casts are identities.

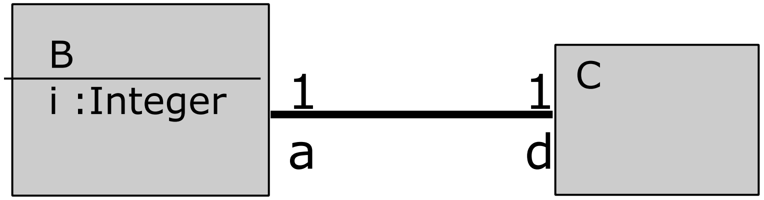
Syntax and Semantics of Object Attributes



Syntax and Semantics of Object Attributes

- Objects represent structured, typed memory in a state σ. They have attributes.

  Attributes can have class types.

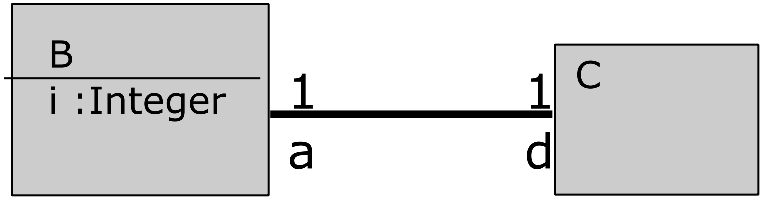


Syntax and Semantics of Object Attributes

- Objects represent structured, typed memory in a state σ. They have attributes.

  Attributes can have class types.

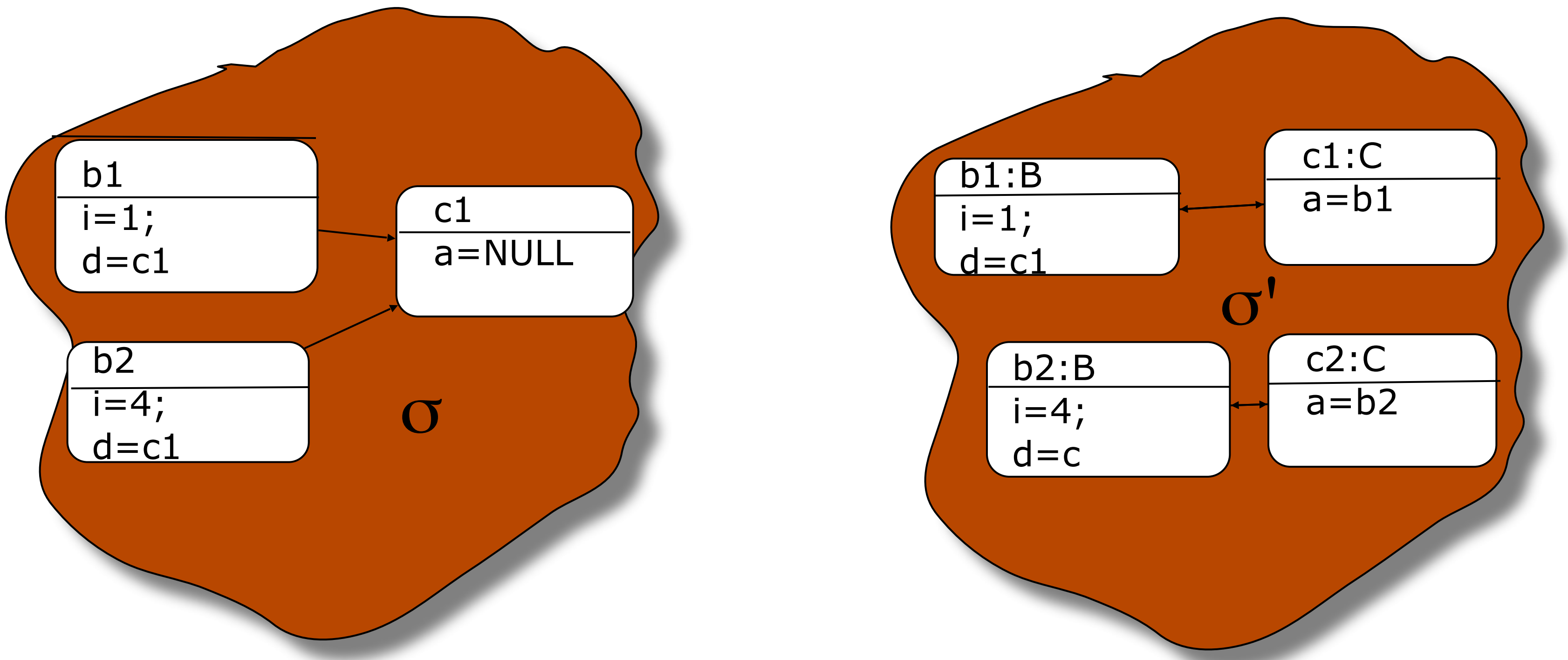
- Reminder: In class diagrams, this situation is represented traditionally by Associations (equivalent)



Syntax and Semantics of Object Attributes

- Example:

attributes of class type in states σ' and σ.



Syntax and Semantics of Object Attributes

Syntax and Semantics of Object Attributes

- each attribute is represented by a function in MOAL. The class diagram right corresponds to delaration of accessor functions:

  .i(σ) :: B -> Integer

  .a(σ) :: C -> B

  .d(σ) :: B -> C

Syntax and Semantics of Object Attributes

- each attribute is represen-
  ted by a function in MOAL.
  The class diagram right
  corresponds to delaration
  of accessor functions:

  .i(σ) :: B -> Integer

  .a(σ) :: C -> B

  .d(σ) :: B -> C

- Applying the σ–convention, this makes navigation expressions possible:

Syntax and Semantics of Object Attributes

- each attribute is represented by a function in MOAL. The class diagram right corresponds to delaration of accessor functions:

  .i(σ) :: B -> Integer

  .a(σ) :: C -> B

  .d(σ) :: B -> C

- Applying the σ–convention, this makes navigation expressions possible:
  - ➢ `b1.d :: C`
    `c1.a :: B` `b1.d.a.d.a ...`

Syntax and Semantics of Object Attributes

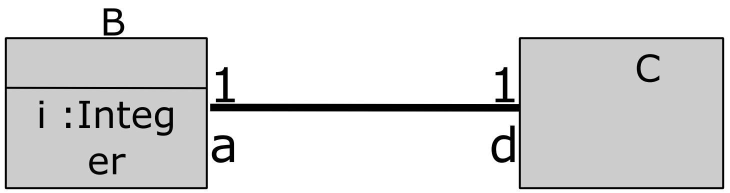
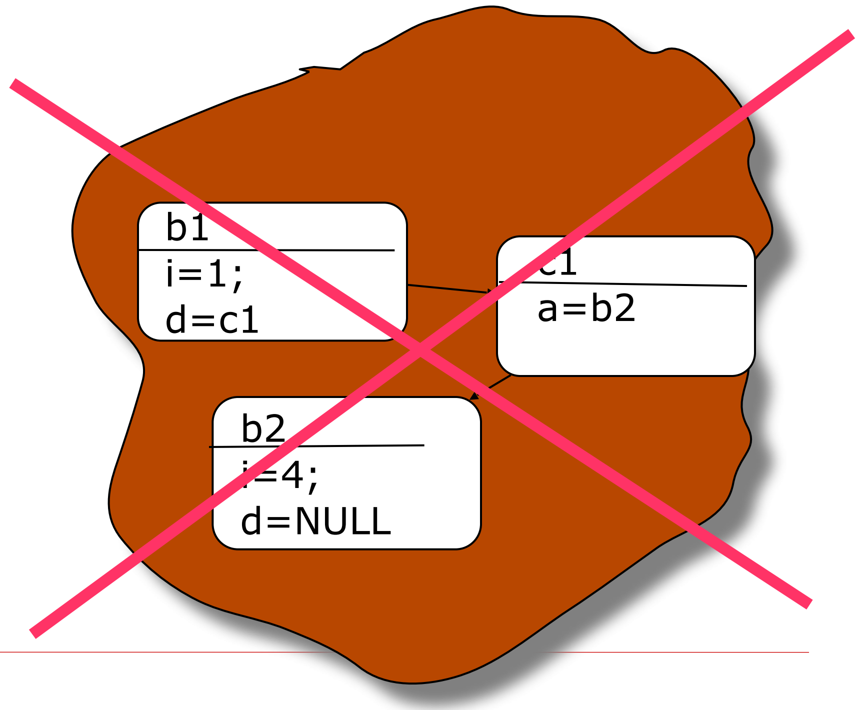
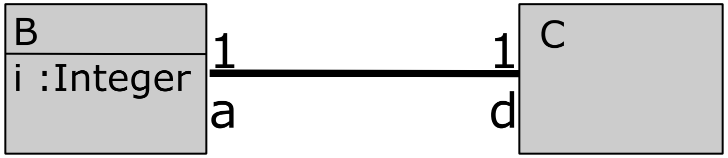
Syntax and Semantics of Object Attributes

- Object assessor functions are „dereferentiations of pointers in a state"

Syntax and Semantics of Object Attributes

- Object assessor functions are „dereferentiations of pointers in a state"
- Accessor functions of class type are strict wrt. NULL.

Syntax and Semantics of Object Attributes

- Object assessor functions are „dereferentiations of pointers in a state"
- Accessor functions of class type are strict wrt. NULL.
  - ```
    NULL.d = NULL
    NULL.a = NULL
    ```
    ```

Syntax and Semantics of Object Attributes

- Object assessor functions are „dereferentiations of pointers in a state"
- Accessor functions of class type are strict wrt. NULL.
  - `NULL.d = NULL`
    `NULL.a = NULL`
  - Note that navigation expressions depend on their underlying state:
    b1.d(σ).a(σ).d(σ).a(σ) = NULL
    b1.d(σ').a(σ').d(σ').a(σ') = b1 !!!

(cf. Object Diagram pp 27)

Syntax and Semantics of Object Attributes



- Note that associations are meant to be « relations » in the mathematical sense.

Thus, states (object-graphs) of this form do not repre-sent the 1:1 association:



Syntax and Semantics of Object Attributes



- This is reflected by 2 « association integrity constraints ».
  For the 1-1-case, they are:

> ➢ definition $ass_{B.d.a} \equiv \forall x \in B.\ x.d.a = x$
>
> ➢ definition $ass_{C.a.d} \equiv \forall x \in C.\ x.a.d = x$

Syntax and Semantics of Object Attributes

Syntax and Semantics of Object Attributes

- Object assessor functions are „dereferentiations of pointers in a state"

Syntax and Semantics of Object Attributes

- Object assessor functions are „dereferentiations of pointers in a state"
- Accessor functions of class type are strict wrt. NULL.

Syntax and Semantics of Object Attributes

- Object assessor functions are „dereferentiations of pointers in a state“
- Accessor functions of class type are strict wrt. NULL.
  - ```
    NULL.d = NULL
    NULL.a = NULL
    ```
    ```

Syntax and Semantics of Object Attributes

- Object assessor functions are „dereferentiations of pointers in a state“
- Accessor functions of class type are strict wrt. NULL.
  - `NULL.d = NULL`
    `NULL.a = NULL`

  - Note that navigation expressions depend on their underlying state:

    b1.d(σ) .a(σ) .d(σ) .a(σ) = NULL

    b1.d(σ’).a(σ’).d(σ’).a(σ’) = b1 !!!

(cf. Object Diagram pp 28)

Syntax and Semantics of Object Attributes

Syntax and Semantics of Object Attributes

- Attibutes can be List or Sets of class types:

Syntax and Semantics of Object Attributes

- Attibutes can be List or Sets of class types:

- Reminder: In class diagrams, this situation is represented traditionally by Associations (equivalent)

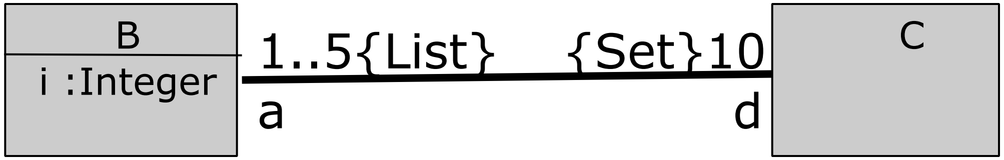
Syntax and Semantics of Object Attributes

- Attibutes can be List or Sets of class types:

- Reminder: In class diagrams, this situation is represented traditionally by Associations (equivalent)

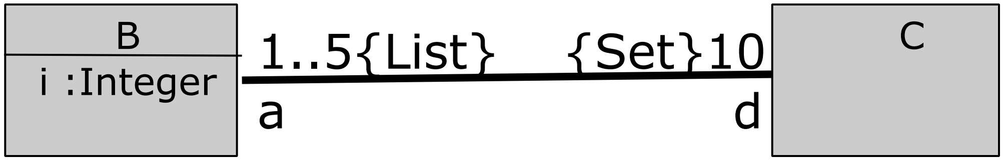
- In analysis-level Class Diagrams, the type information is still omitted; due to overloading of ∀x∈X. P(x) etc. this will not hamper us to specify ...

Syntax and Semantics of Object Attributes



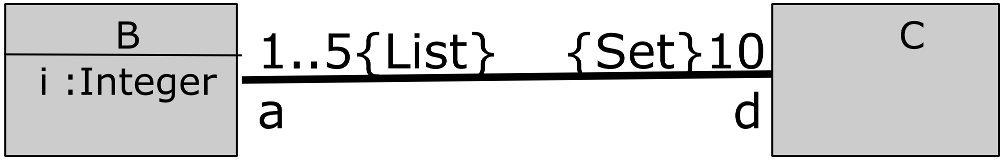
Syntax and Semantics of Object Attributes

- Cardinalities in Associations can be translated canonically into MOCL invariants:



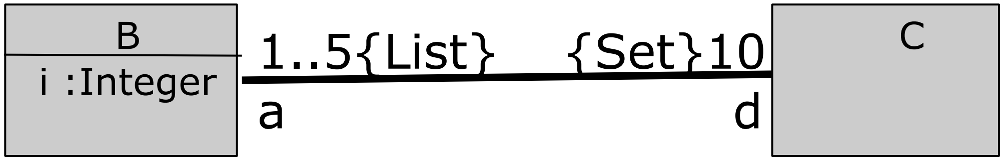
Syntax and Semantics of Object Attributes

- Cardinalities in Associations can be translated canonically into MOCL invariants:



Syntax and Semantics of Object Attributes

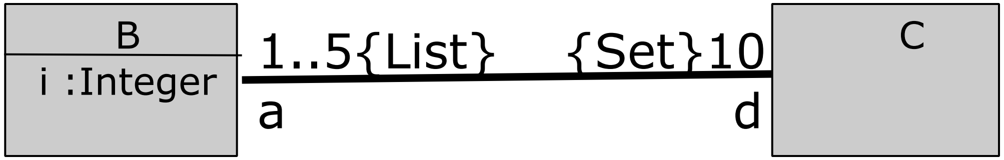
- Cardinalities in Associations can be translated canonically into MOCL invariants:



➢ definition $card_{B.d} \equiv \forall x \in B.\ |x.d| = 10$

Syntax and Semantics of Object Attributes

- Cardinalities in Associations can be translated canonically into MOCL invariants:



> ➢ definition $card_{B.d} \equiv \forall x \in B.\ |x.d| = 10$
>
> ➢ definition $card_{C.a} \equiv \forall x \in C.\ 1 \leq |x.a| \leq 5$

Syntax and Semantics of Object Attributes

Syntax and Semantics of Object Attributes

- Accessor functions are defined as follows for the case of NULL:

Syntax and Semantics of Object Attributes

- Accessor functions are defined as follows for the case of NULL:

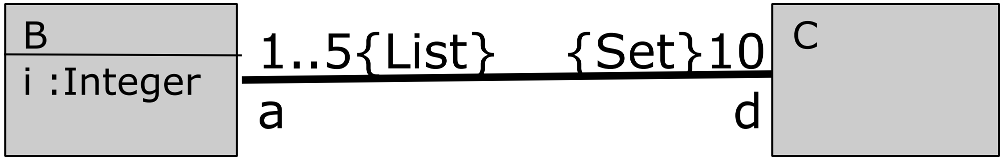
  - NULL.d = {}   -- mapping to the neutral element

Syntax and Semantics of Object Attributes

- Accessor functions are defined as follows for the case of NULL:

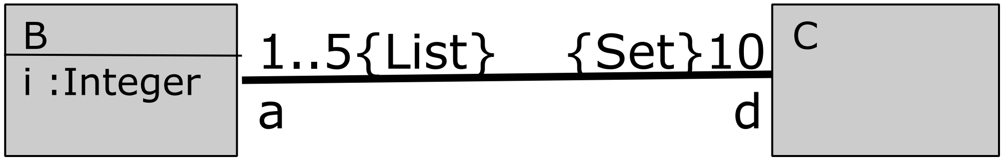
  - NULL.d = {} -- mapping to the neutral element
  - NULL.a = [] -- mapping to the neural element.

Syntax and Semantics of Object Attributes



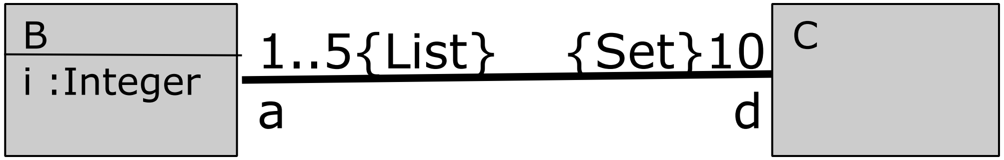
Syntax and Semantics of Object Attributes

- Cardinalities in Associations can be translated canonically into MOCL invariants:



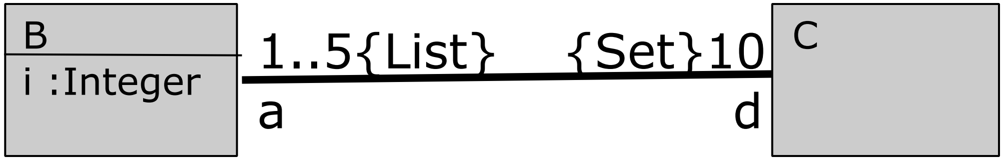
Syntax and Semantics of Object Attributes

- Cardinalities in Associations can be translated canonically into MOCL invariants:



Syntax and Semantics of Object Attributes

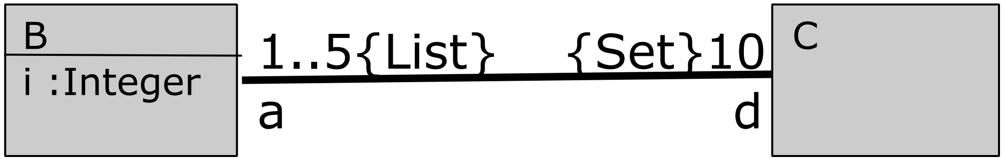
- Cardinalities in Associations can be translated canonically into MOCL invariants:



- definition $card_{B.d} \equiv \forall x \in B.\ |x.d| = 10$

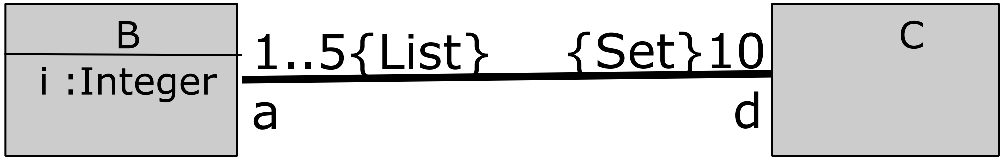
Syntax and Semantics of Object Attributes

- Cardinalities in Associations can be translated canonically into MOCL invariants:



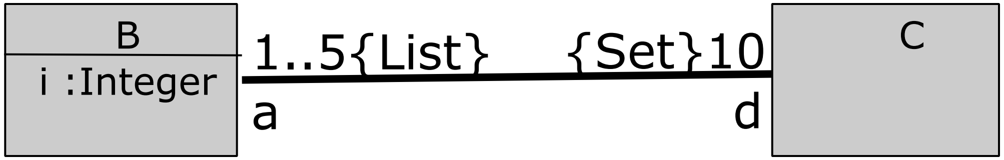
> ➢ definition $card_{B.d} \equiv \forall x \in B.\ |x.d| = 10$
>
> ➢ definition $card_{C.a} \equiv \forall x \in C.\ 1 \leq |x.a| \leq 5$

Syntax and Semantics of Object Attributes



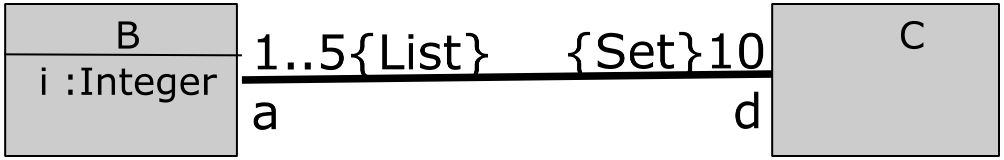
Syntax and Semantics of Object Attributes

- The corresponding association integrity constraints for the *-*-case are:



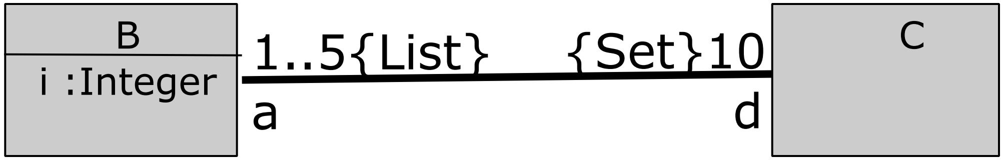
Syntax and Semantics of Object Attributes

- The corresponding association integrity constraints for the *-*-case are:



Syntax and Semantics of Object Attributes

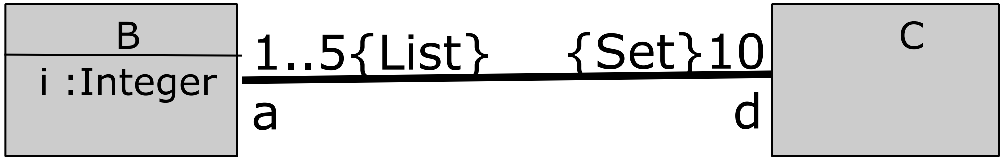
- The corresponding association integrity constraints for the *-*-case are:



> ➢ definition $ass_{B.d.a} \equiv \forall x \in B.\ x \in x.d.a$

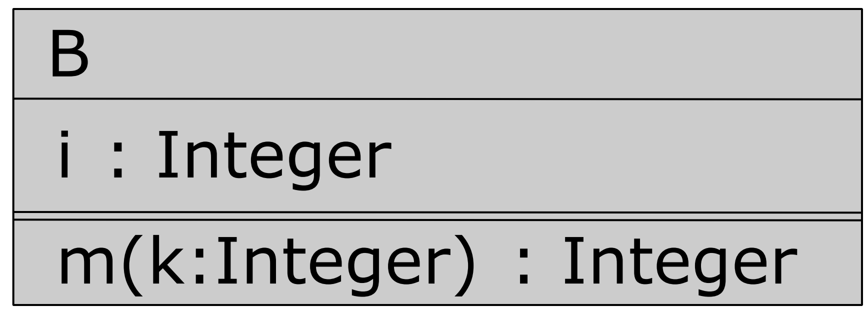
Syntax and Semantics of Object Attributes

- The corresponding association integrity constraints for the *-*-case are:



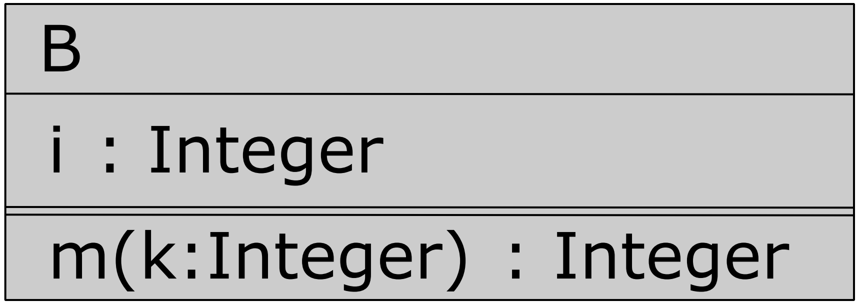
- $\text{definition ass}_{B.d.a} \equiv \forall x \in B.\ x \in x.d.a$
- $\text{definition ass}_{C.a.d} \equiv \forall x \in C.\ x \in x.a.d$

Operations in UML and MOAL



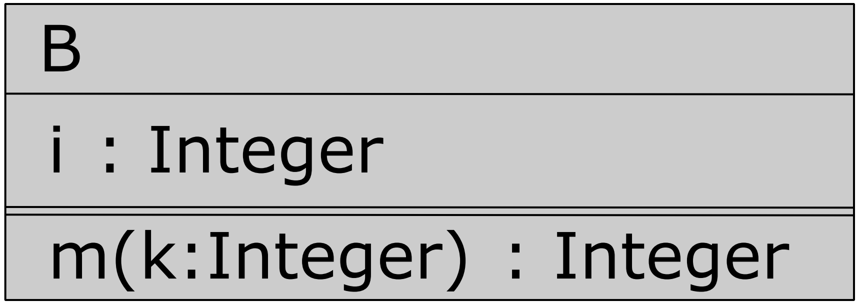
Operations in UML and MOAL

- Many UML diagrams talk over a sequence of states (not just individual global states)



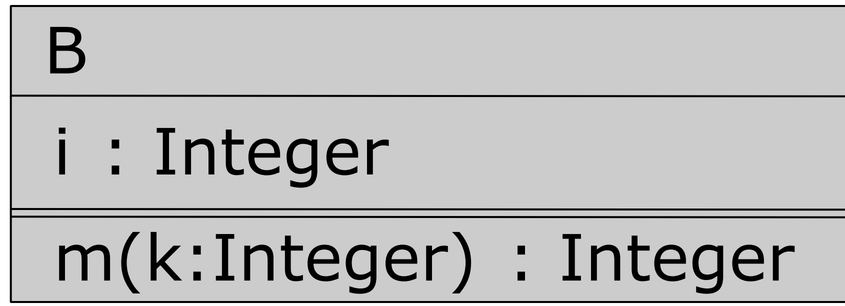
Operations in UML and MOAL

- Many UML diagrams talk over a sequence of states (not just individual global states)
- This appears for the first time in so-called contracts for (Class-model) methods:



Operations in UML and MOAL

- Many UML diagrams talk over a sequence of states (not just individual global states)
- This appears for the first time in so-called contracts for (Class-model) methods:

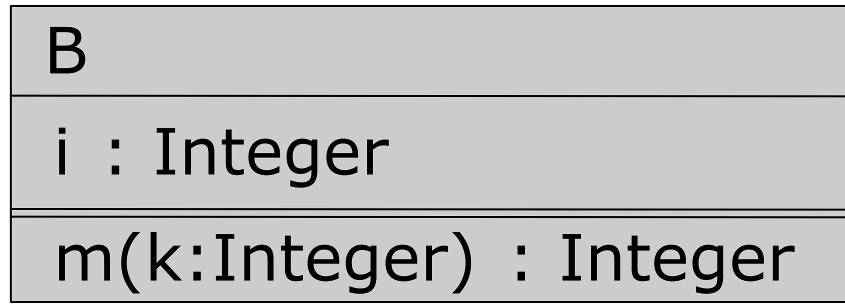


- The « method » m can be seen as a « transaction » of a B object transforming the underlying pre-state σ_{pre} in the state « after » m yielding a post-state σ.

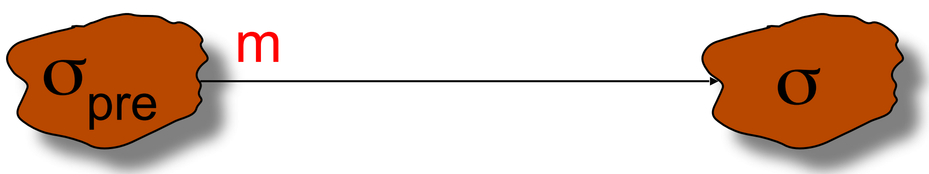
m

Operations in UML and MOAL

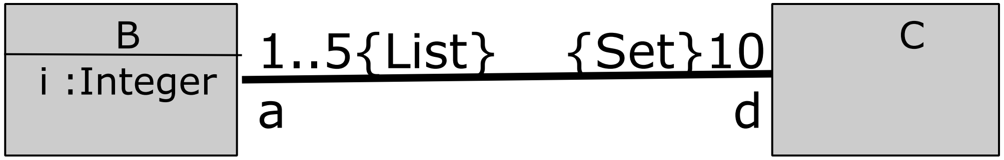
- Many UML diagrams talk over a sequence of states (not just individual global states)
- This appears for the first time in so-called contracts for (Class-model) methods:



- The « method » m can be seen as a « transaction » of a B object transforming the underlying pre-state σ_{pre} in the state « after » m yielding a post-state σ.

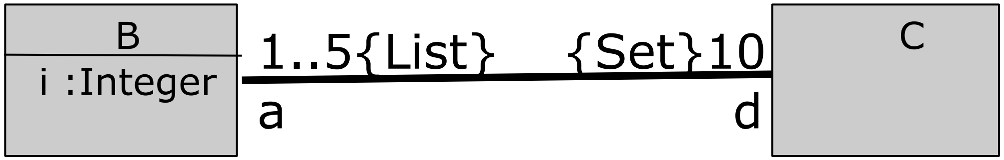


Syntax and Semantics of Object Attributes



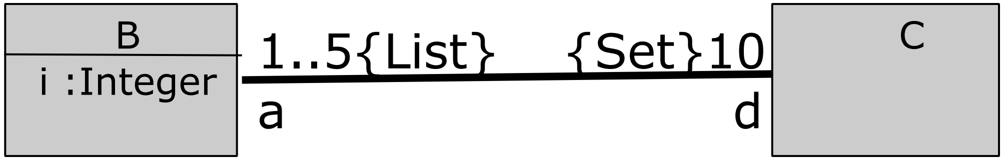
Syntax and Semantics of Object Attributes

- Cardinalities in Associations can be translated canonically into MOCL invariants:



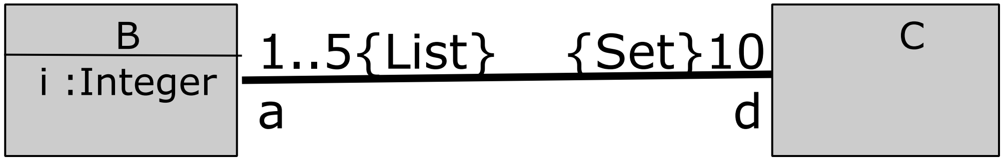
Syntax and Semantics of Object Attributes

- Cardinalities in Associations can be translated canonically into MOCL invariants:



Syntax and Semantics of Object Attributes

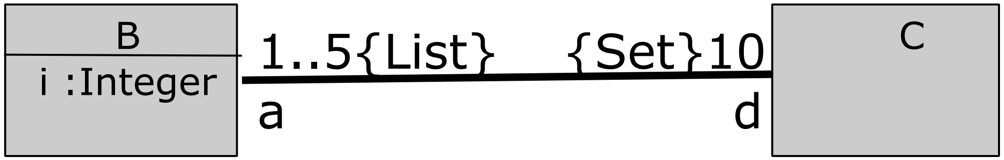
- Cardinalities in Associations can be translated canonically into MOCL invariants:



> ➢ definition $card_{B.d} \equiv \forall x \in B.\ |x.d| = 10$

Syntax and Semantics of Object Attributes

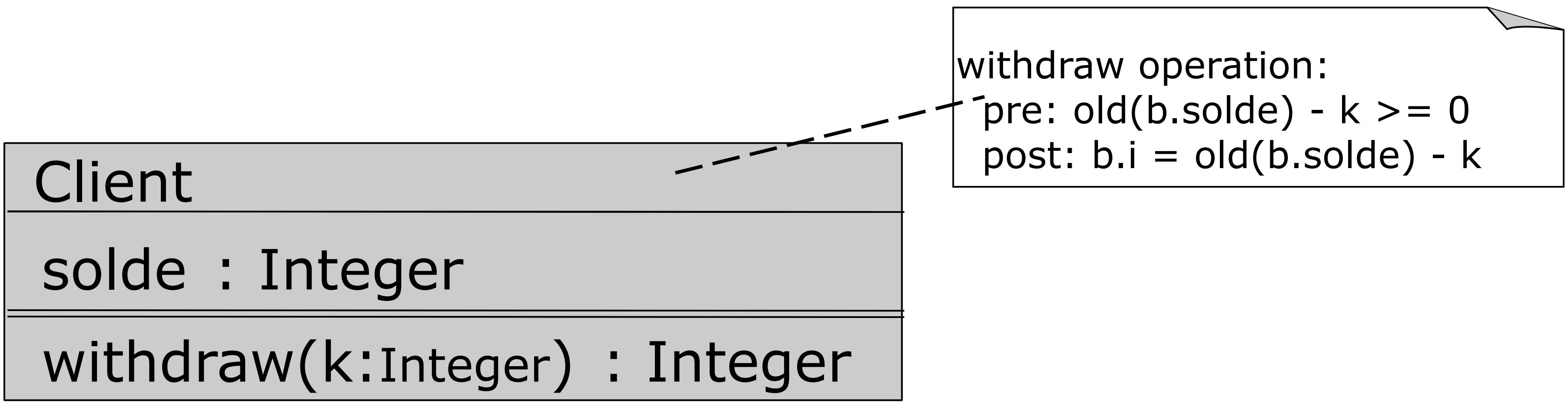
- Cardinalities in Associations can be translated canonically into MOCL invariants:



- definition $card_{B.d} \equiv \forall x \in B.\ |x.d| = 10$
- definition $card_{C.a} \equiv \forall x \in C.\ 1 \leq |x.a| \leq 5$

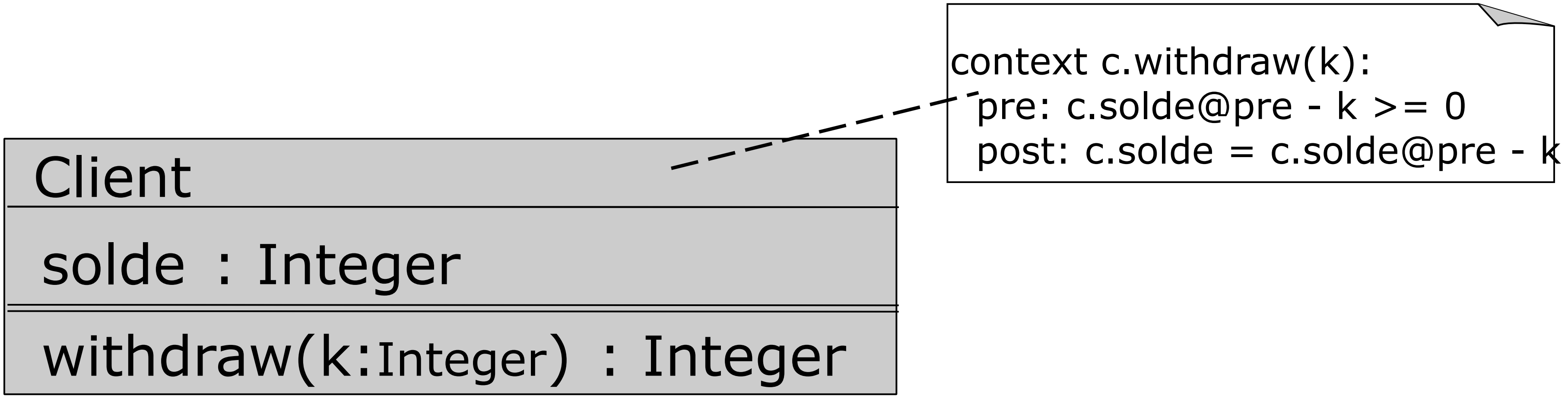
Operations in UML and MOAL

- Syntactically, contracts are annotated like this (JML-ish):



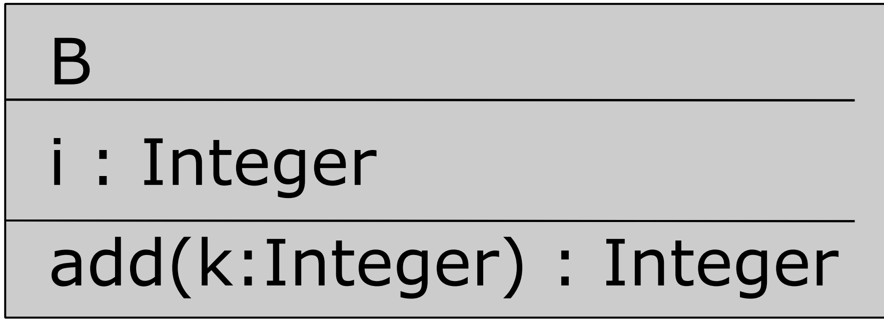
Operations in UML and MOAL

- ... or like this (OCL-ish):



Operations in UML and MOAL Contracts

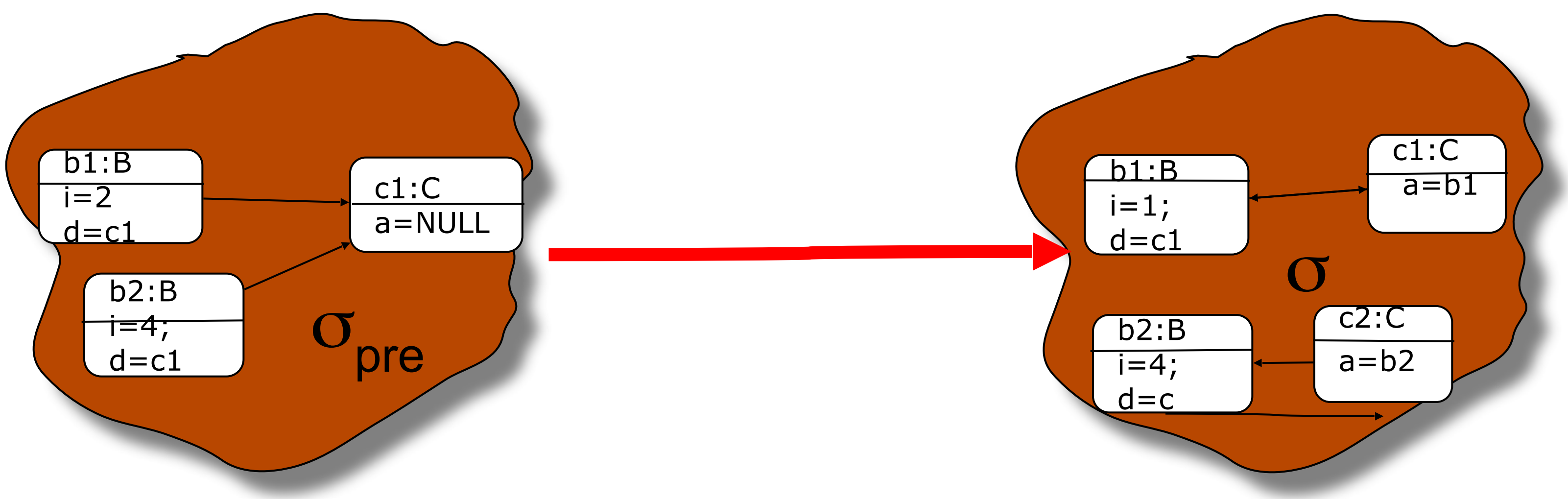
- This appears for the first time in so-called contracts for (Class-model) methods:



- The « method » add can be seen as a « transaction » of a B object transforming the underlying pre-state σ_{pre} in the state « after » add yielding a post-state σ.

Syntax and Semantics of MOAL Contracts

- Again: This is the view of a transaction (like in a data-base), it completely abstracts away intermediate states or time. (This possible in other models/calculi, like the Hoare-calculus, though).



Syntax and Semantics of MOAL Contracts

Syntax and Semantics of MOAL Contracts

- Consequence:

Syntax and Semantics of MOAL Contracts

- Consequence:
  - The pre-condition is a formula referring to the σ_{pre} and the method arguments b1, a_1, ..., a_n only.

Syntax and Semantics of MOAL Contracts

- Consequence:
  - The pre-condition is a formula referring to the σ_{pre} and the method arguments b1, a_1, ..., a_n only.
  - the post-condition is only assured if the pre-condition is satisfied

Syntax and Semantics of MOAL Contracts

- Consequence:
  - The pre-condition is a formula referring to the σ_{pre} and the method arguments b1, a_1, ..., a_n only.
  - the post-condition is only assured if the pre-condition is satisfied
  - otherwise the method

Syntax and Semantics of MOAL Contracts

- Consequence:
  - The pre-condition is a formula referring to the σ_{pre} and the method arguments b1, a_1, ..., a_n only.
  - the post-condition is only assured if the pre-condition is satisfied
  - otherwise the method
    - ...may do anything on the state and the result,
      may even behave correctly , may non-terminate !

Syntax and Semantics of MOAL Contracts

- Consequence:
  - The pre-condition is a formula referring to the σ_{pre} and the method arguments b1, a_1, ..., a_n only.
  - the post-condition is only assured if the pre-condition is satisfied
  - otherwise the method
    - ...may do anything on the state and the result,
      may even behave correctly , may non-terminate !
    - raise an exception
      (recommended in Java Programmer Guides
      for public methods to increase robustness)

Syntax and Semantics of MOAL Contracts

Syntax and Semantics of MOAL Contracts

- Consequence:

Syntax and Semantics of MOAL Contracts

- Consequence:
  - The post-condition is a formula referring to both σ_{pre} and σ, the method arguments b1, a_1, ..., a_n and the return value captured by the variable result.

Syntax and Semantics of MOAL Contracts

- Consequence:
  - The post-condition is a formula referring to both σ_{pre} and σ, the method arguments b1, a_1, ..., a_n and the return value captured by the variable result.
  - any transition is permitted that satisfies the post-condition (provided that the pre-condition is true)

Syntax and Semantics of MOAL Contracts

Syntax and Semantics of MOAL Contracts

- Consequence:

Syntax and Semantics of MOAL Contracts

- Consequence:
  - The semantics of a method call:

$$b1.m(a_1, ..., a_n)$$

is thus:

$$pre_m(b1,a_1, ..., a_n)\ (\sigma_{pre})$$

$$\longrightarrow$$

$$post_m(b1,a_1, ..., a_n,result)(\sigma_{pre},\sigma)$$

Syntax and Semantics of MOAL Contracts

- Consequence:
  - The semantics of a method call:

$$b1.m(a_1, \ldots, a_n)$$

    is thus:

$$pre_m(b1,a_1, \ldots, a_n)\ (\sigma_{pre})$$

$$\longrightarrow$$

$$post_m(b1,a_1, \ldots, a_n,result)(\sigma_{pre},\sigma)$$

  - Note that moreover all global class invarants have to be added for both pre-state σ_{pre} and post-state σ !

Syntax and Semantics of MOAL Contracts

- Consequence:
  - The semantics of a method call:

$$b1.m(a_1, ..., a_n)$$

is thus:

$$pre_m(b1,a_1, ..., a_n)(\sigma_{pre})$$

$$\longrightarrow$$

$$post_m(b1,a_1, ..., a_n,result)(\sigma_{pre},\sigma)$$

  - Note that moreover all global class invarants have to be added for both pre-state σ_{pre} and post-state σ !

    For an entire transition, the following must hold:

$$Inv(\sigma_{pre}) \wedge pre_m ... (\sigma_{pre}) \wedge post ... (\sigma_{pre},\sigma) \wedge Inv(\sigma)$$

Syntax and Semantics of MOAL Contracts

Syntax and Semantics of MOAL Contracts

- Example:

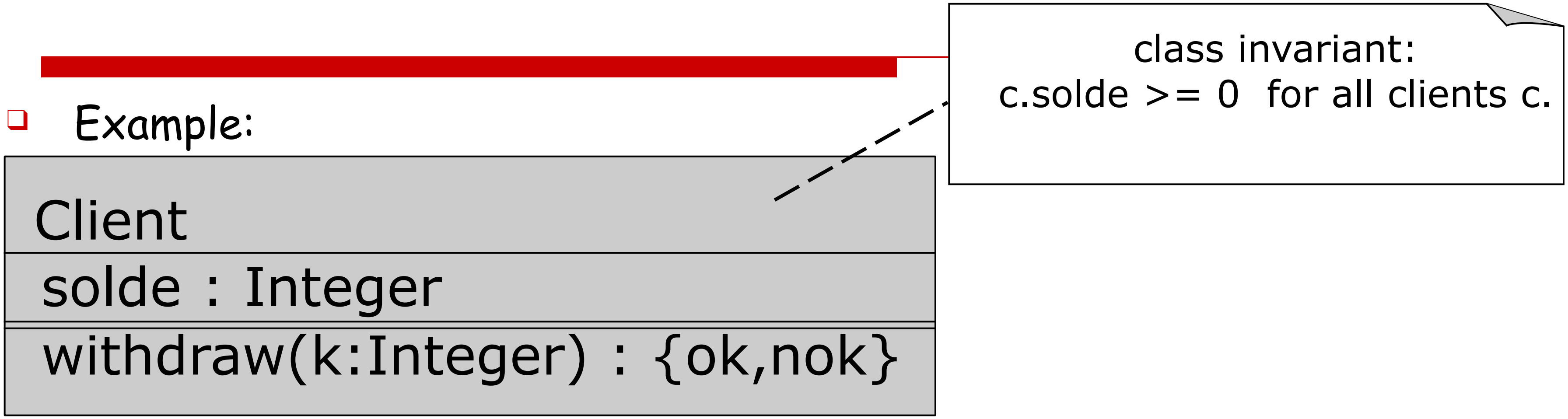
Syntax and Semantics of MOAL Contracts

- Example:

Client
solde : Integer
withdraw(k:Integer) : {ok,nok}

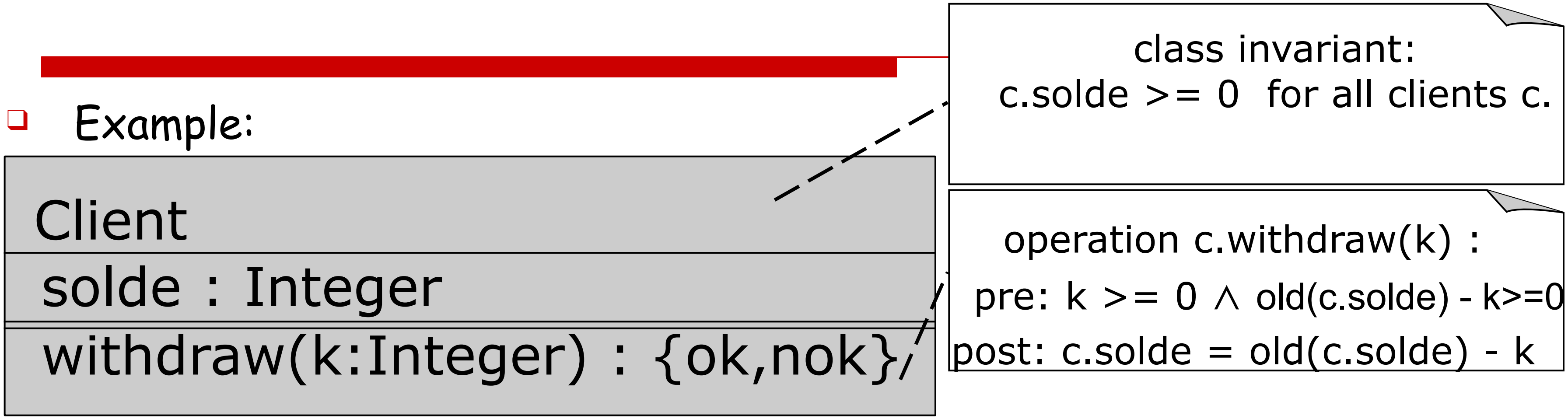
Syntax and Semantics of MOAL Contracts

- Example:



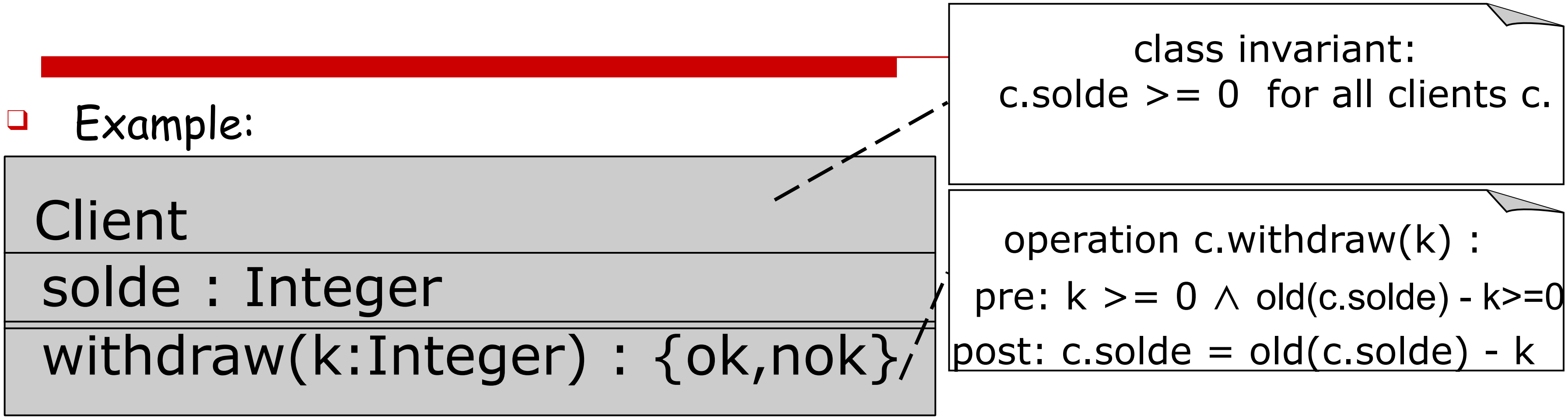
Syntax and Semantics of MOAL Contracts

- Example:



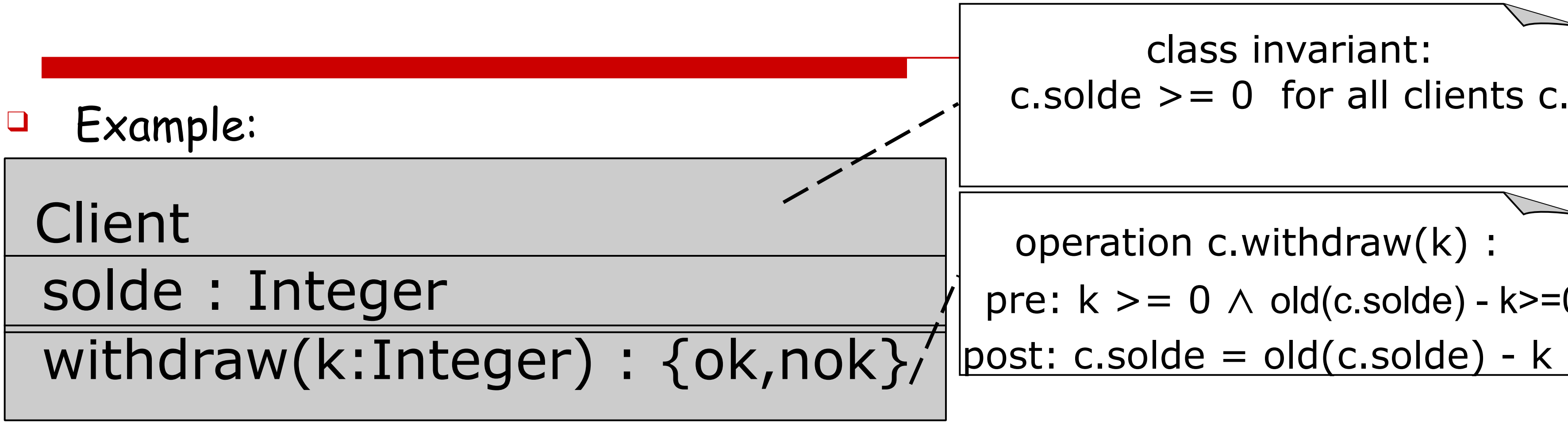
Syntax and Semantics of MOAL Contracts

- Example:



Syntax and Semantics of MOAL Contracts

- Example:



- definition $inv_{Client}(\sigma) \equiv$
  $\forall c \in Client(\sigma).\ 0 \leq c.solde(\sigma)$

Syntax and Semantics of MOAL Contracts

- Example:

Client
solde : Integer
withdraw(k:Integer) : {ok,nok}

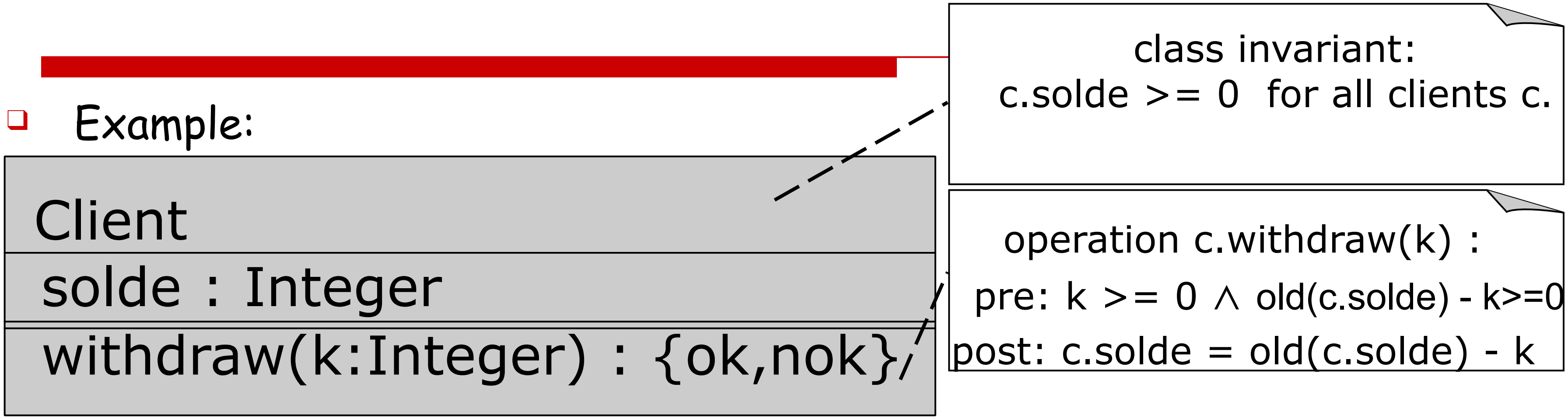
class invariant:
c.solde >= 0 for all clients c.

operation c.withdraw(k) :
pre: k >= 0 ∧ old(c.solde) - k>=0
post: c.solde = old(c.solde) - k

- definition $inv_{Client}(\sigma) \equiv$
  $\forall c \in Client(\sigma).\ 0 \leq c.solde(\sigma)$
- definition $pre_{withdraw}(c, k)(\sigma) \equiv$
  $c \in Client(\sigma) \wedge 0 \leq k \wedge 0 \leq c.solde(\sigma) - k$

Syntax and Semantics of MOAL Contracts

- Example:



- definition $\mathtt{inv}_{\mathtt{Client}}(\sigma) \equiv$

  $\forall c \in \mathtt{Client}(\sigma).\ 0 \leq \mathtt{c.solde}(\sigma)$

- definition $\mathtt{pre}_{\mathtt{withdraw}}(c, k)(\sigma) \equiv$

  $c \in \mathtt{Client}(\sigma) \wedge 0 \leq k \wedge 0 \leq \mathtt{c.solde}(\sigma) - k$

- definition $\mathtt{post}_{\mathtt{withdraw}}(c, k, result)(\sigma_{pre}, \sigma) \equiv$

  $c \in \mathtt{Client}(\sigma_{pre}) \wedge \mathtt{c.solde}(\sigma) = \mathtt{c.solde}(\sigma_{pre}) - k \wedge$

  $\mathtt{result} = \mathtt{ok}$

Syntax and Semantics of MOAL Contracts

Syntax and Semantics of MOAL Contracts

- Notation:

Syntax and Semantics of MOAL Contracts

- Notation:
  - In order to relax notation, we will use for applications to σ_{pre} the old-notation:

Syntax and Semantics of MOAL Contracts

- Notation:
  - In order to relax notation, we will use for applications to σ_{pre} the old-notation:

Client(σ_{pre}) becomes old(Client)

Syntax and Semantics of MOAL Contracts

- Notation:
  - In order to relax notation, we will use for applications to σ_{pre} the old-notation:

Client(σ_{pre}) **becomes** old(Client)

`c.solde`(σ_{pre}) **becomes** old(`c.solde`)

Syntax and Semantics of MOAL Contracts

- Notation:
  - In order to relax notation, we will use for applications to σ_{pre} the old-notation:

Client(σ_{pre}) **becomes** old(Client)

`c.solde`(σ_{pre}) **becomes** old(`c.solde`)

etc.

Syntax and Semantics of MOAL Contracts

Syntax and Semantics of MOAL Contracts

- Example (revised):

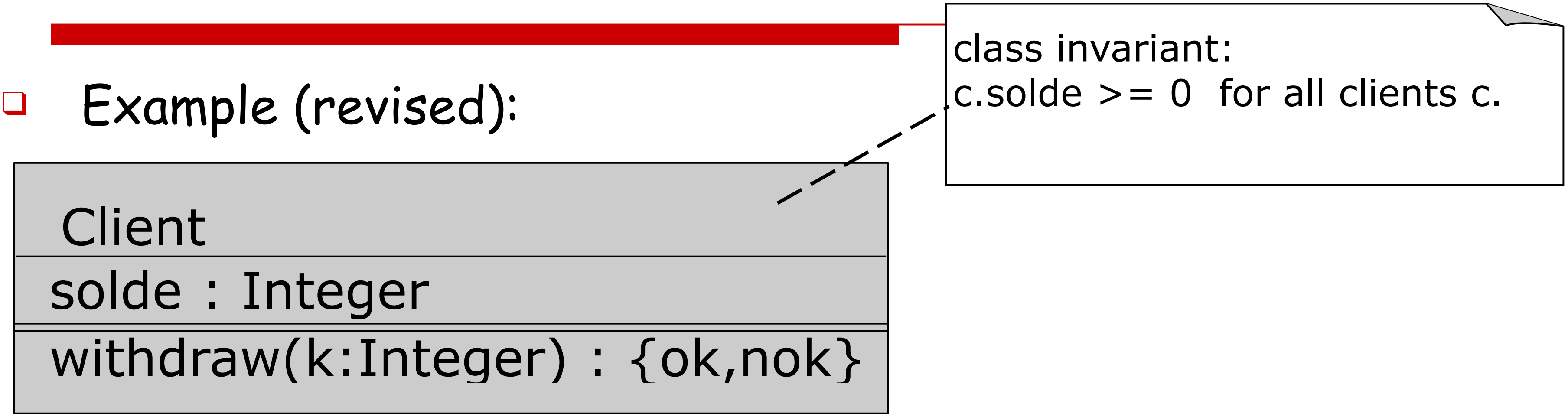
Syntax and Semantics of MOAL Contracts

- Example (revised):

Client
solde : Integer
withdraw(k:Integer) : {ok,nok}

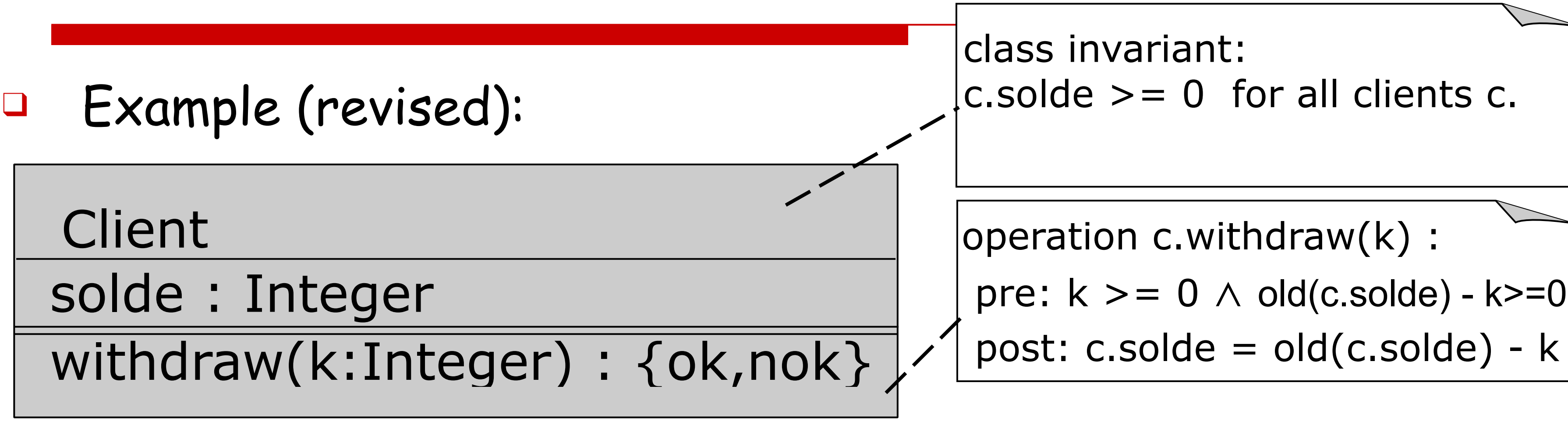
Syntax and Semantics of MOAL Contracts

- Example (revised):



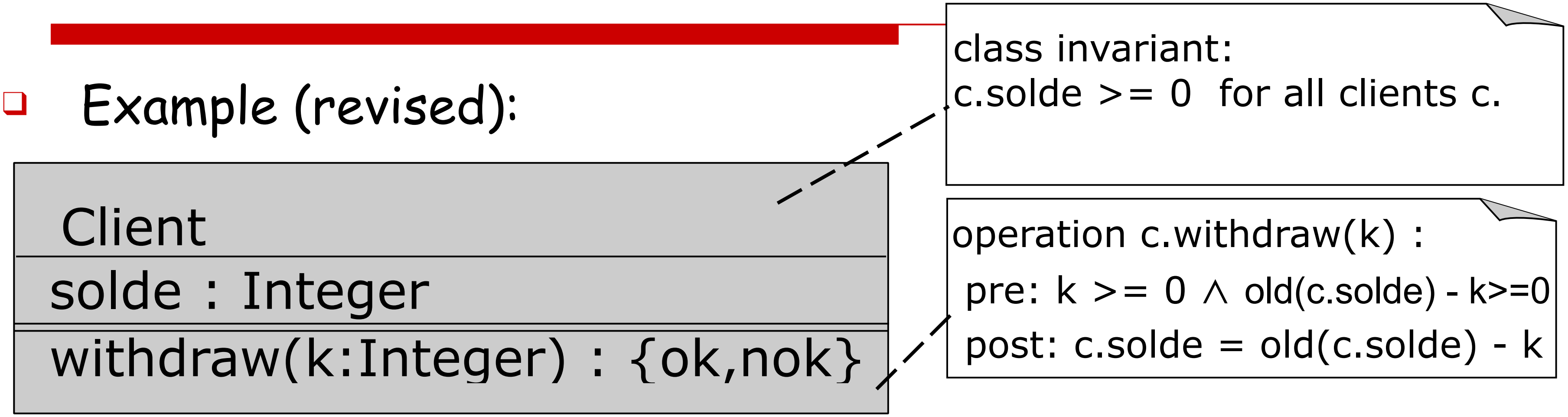
Syntax and Semantics of MOAL Contracts

- Example (revised):



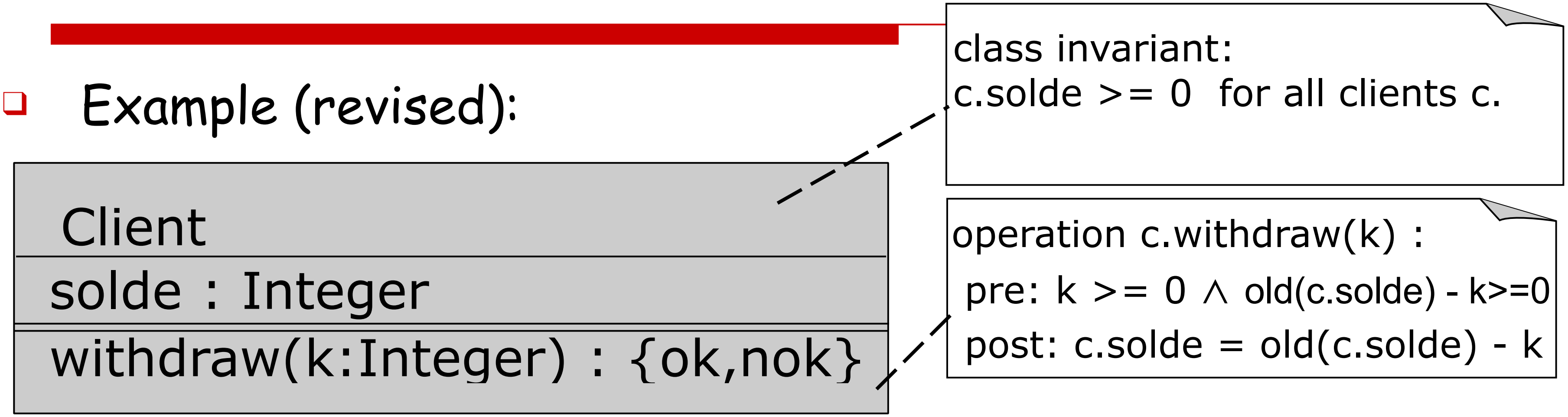
Syntax and Semantics of MOAL Contracts

- Example (revised):



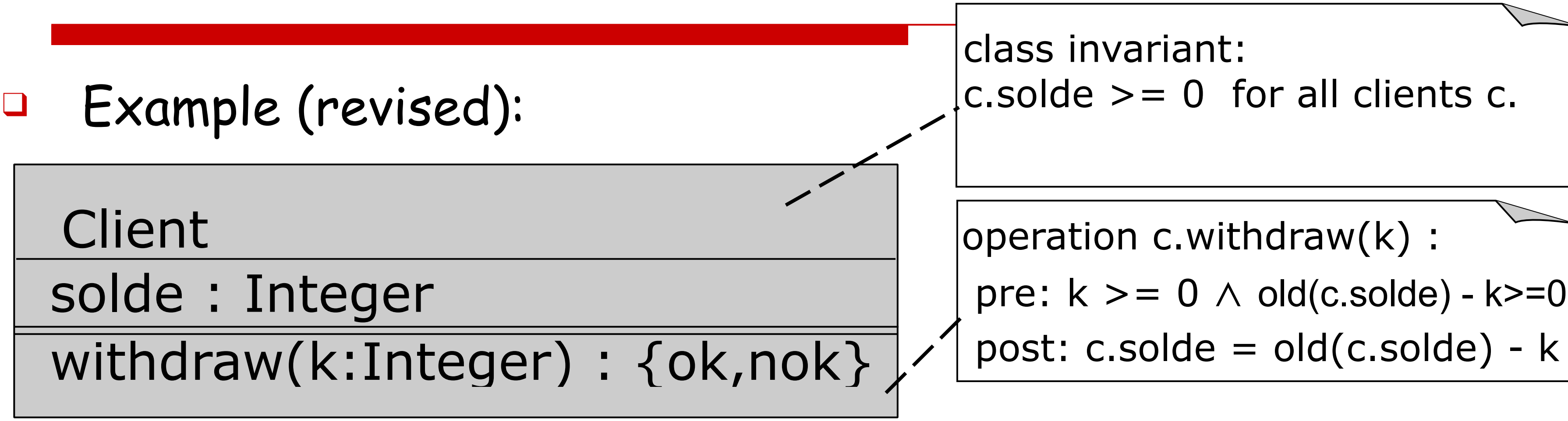
Syntax and Semantics of MOAL Contracts

- Example (revised):



Syntax and Semantics of MOAL Contracts

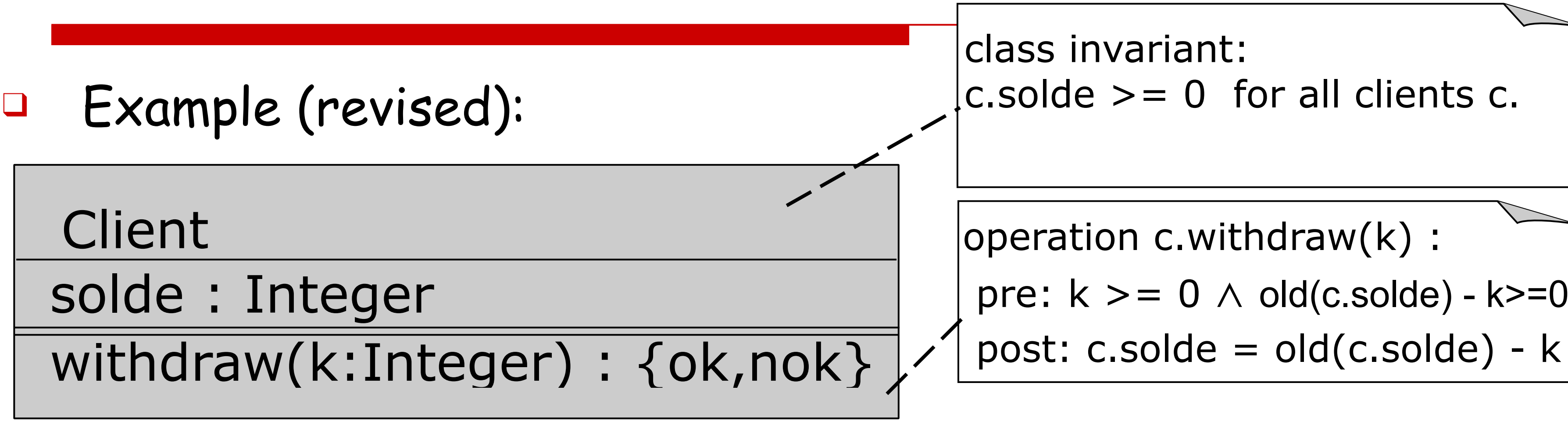
- Example (revised):



- definition $inv_{Client} \equiv \forall c \in Client.\ 0 \leq c.solde$

Syntax and Semantics of MOAL Contracts

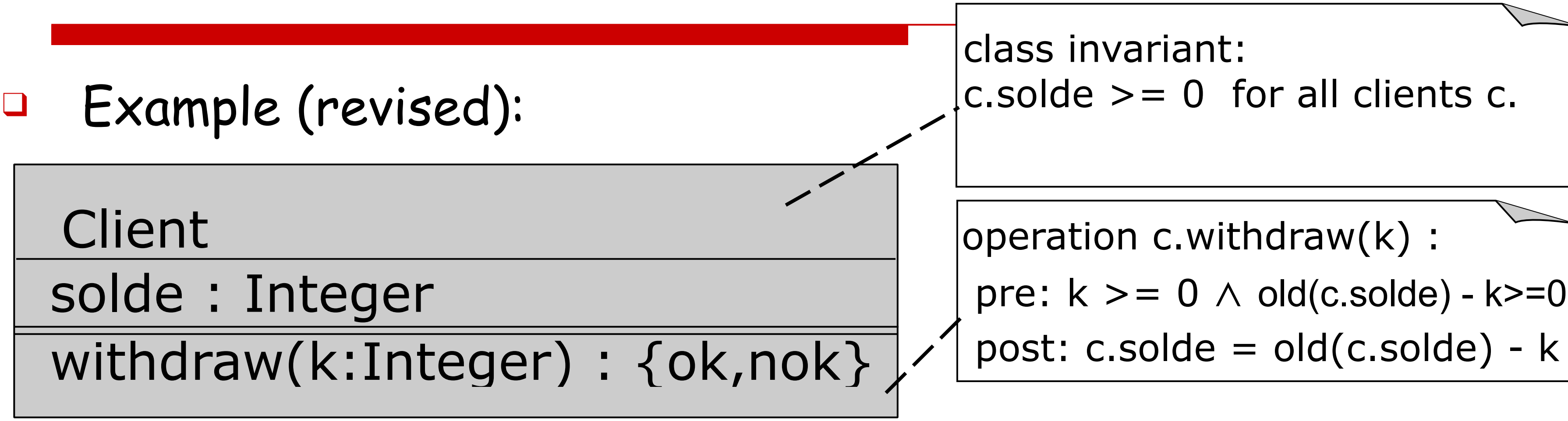
- Example (revised):



- definition $inv_{Client} \equiv \forall c \in Client.\ 0 \le c.solde$
- definition $pre_{withdraw}(c, k) \equiv$
  $c \in Client \wedge 0 \le k \wedge 0 \le c.solde - k$

Syntax and Semantics of MOAL Contracts

- Example (revised):



- definition $inv_{Client} \equiv \forall c \in Client.\ 0 \leq c.solde$
- definition $pre_{withdraw}(c, k) \equiv$

  $c \in Client \wedge 0 \leq k \wedge 0 \leq c.solde - k$
- definition $post_{withdraw}(c, k, result) \equiv$

  $c \in old(Client) \wedge c.solde = old(c.solde) - k \wedge$

  $result = ok$

Semantics of MOAL Contracts

Semantics of MOAL Contracts

- Two predicates are helpful when defining contracts. They exceptionally refer to both (σ_{pre},σ)

Semantics of MOAL Contracts

- Two predicates are helpful when defining contracts. They exceptionally refer to both (σ_{pre},σ)
  - `isNew(p)`(σ_{pre},σ) is true only if object p of class C does not exist in σ_{pre} but exists in σ

Semantics of MOAL Contracts

- Two predicates are helpful when defining contracts. They exceptionally refer to both (σ_{pre}, σ)

  - `isNew(p)` (σ_{pre}, σ) is true only if object p of class C does not exist in σ_{pre} but exists in σ

  - modifiesOnly(S)(σ_{pre}, σ) is only true iff

Semantics of MOAL Contracts

- Two predicates are helpful when defining contracts. They exceptionally refer to both (σ_{pre},σ)
  - `isNew(p)` (σ_{pre},σ) is true only if object p of class C does not exist in σ_{pre} but exists in σ
  - modifiesOnly(S)(σ_{pre},σ) is only true iff
    - all objects in σ_{pre} are except those in S identical in σ

Semantics of MOAL Contracts

- Two predicates are helpful when defining contracts. They exceptionally refer to both (σ_{pre},σ)
  - `isNew(p)` (σ_{pre},σ) is true only if object p of class C does not exist in σ_{pre} but exists in σ
  - modifiesOnly(S)(σ_{pre},σ) is only true iff
    - all objects in σ_{pre} are except those in S identical in σ
    - all objects in σ exist either in are or are contained in S

Semantics of MOAL Contracts

- Two predicates are helpful when defining contracts. They exceptionally refer to both (σ_{pre},σ)
  - `isNew(p)` (σ_{pre},σ) is true only if object p of class C does not exist in σ_{pre} but exists in σ
  - modifiesOnly(S)(σ_{pre},σ) is only true iff
    - all objects in σ_{pre} are except those in S identical in σ
    - all objects in σ exist either in are or are contained in S

With this predicate, one can express : „and nothing else changes". It is also called «framing condition».

Semantics of MOAL Contracts

- Two predicates are helpful when defining contracts. They exceptionally refer to both (σ_{pre},σ)

  - `isNew(p)` (σ_{pre},σ) is true only if object p of class C does not exist in σ_{pre} but exists in σ

  - modifiesOnly(S)(σ_{pre},σ) is only true iff
    - all objects in σ_{pre} are except those in S identical in σ
    - all objects in σ exist either in are or are contained in S

  With this predicate, one can express : „and nothing else changes". It is also called «framing condition».

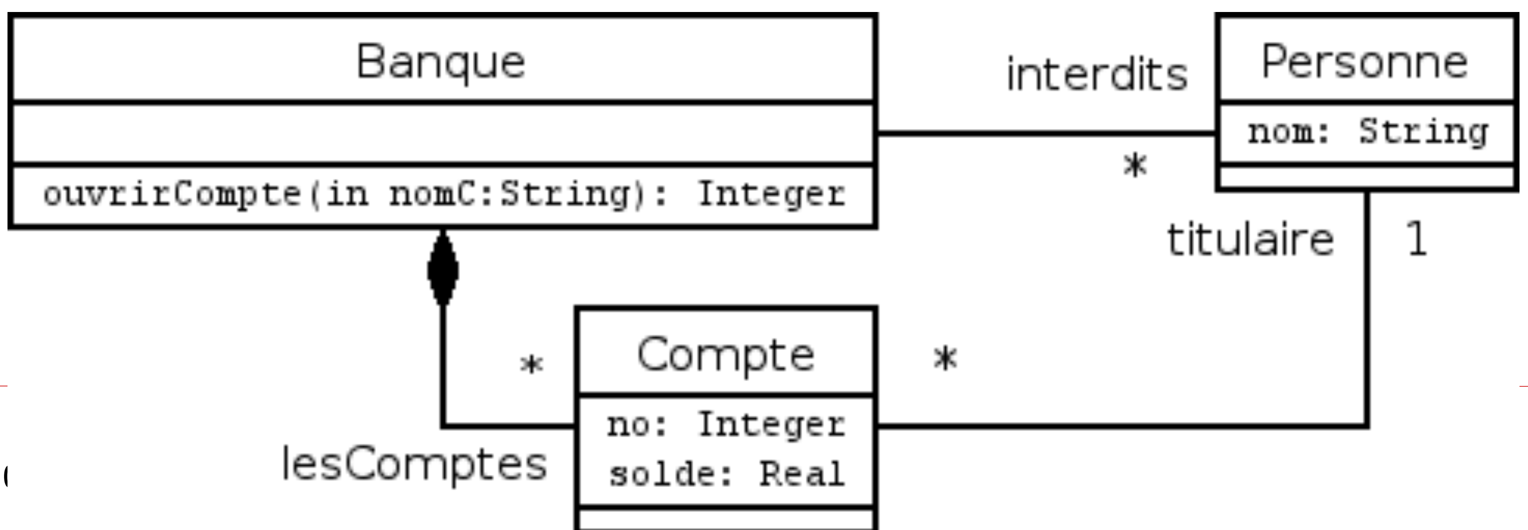
A Revision of the Example: Bank

A Revision of the Example: Bank

Opening a bank account. Constraints:

A Revision of the Example: Bank

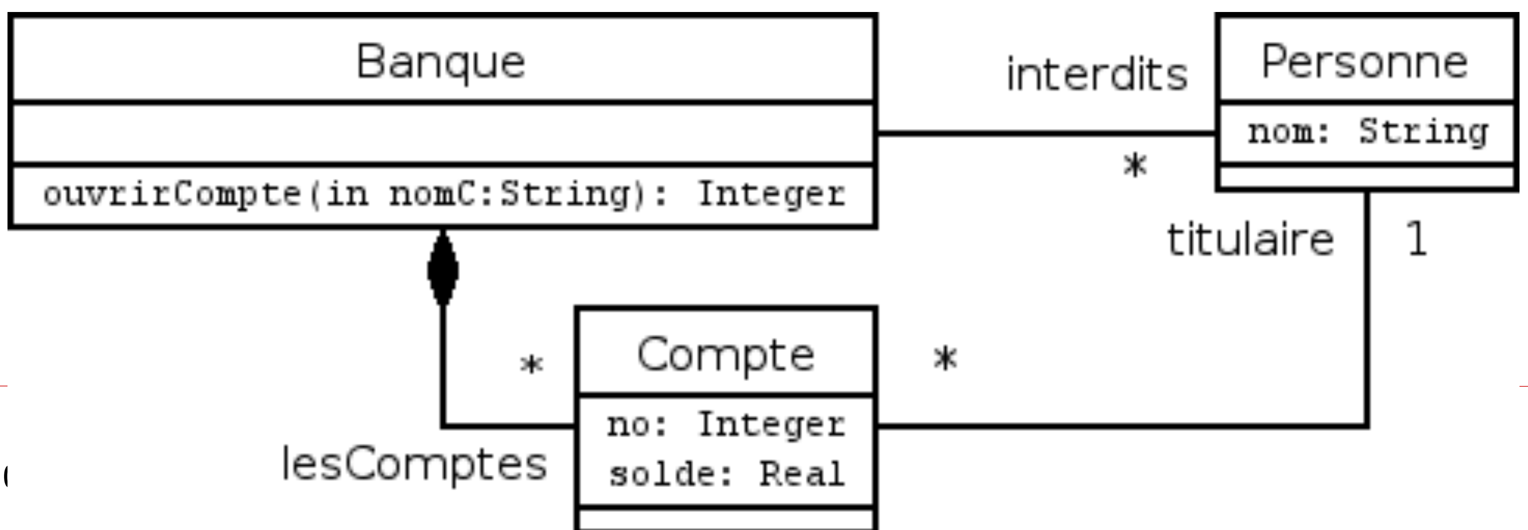
Opening a bank account. Constraints:



A Revision of the Example: Bank

Opening a bank account. Constraints:

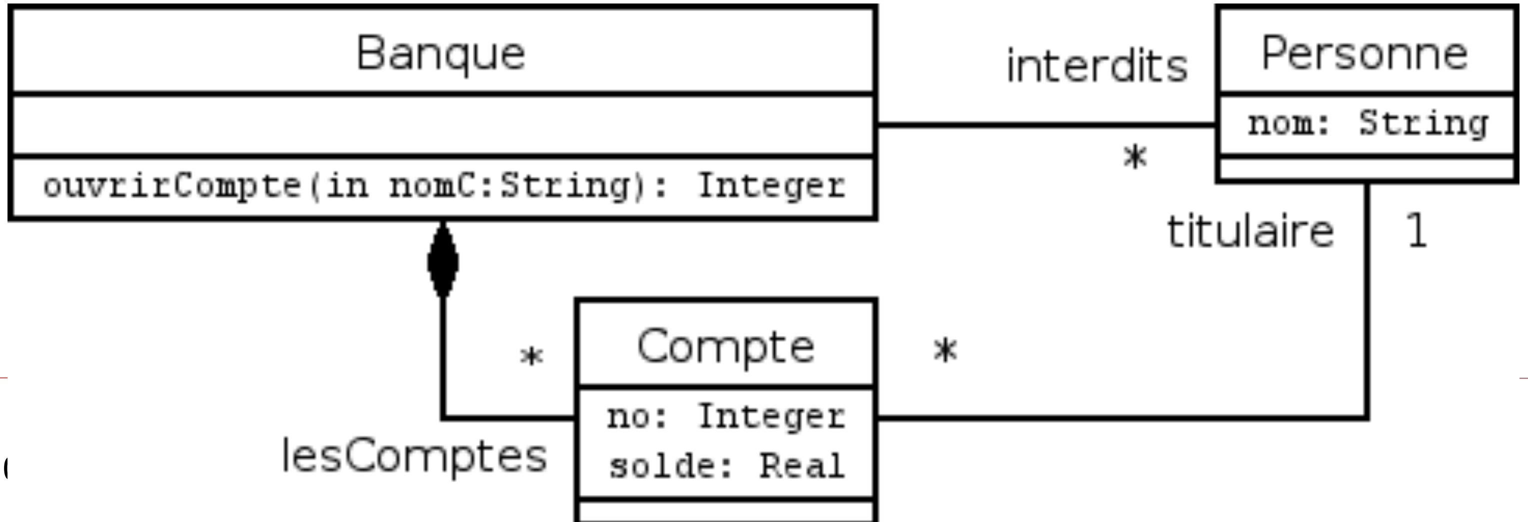
- there is a blacklist



A Revision of the Example: Bank

Opening a bank account. Constraints:

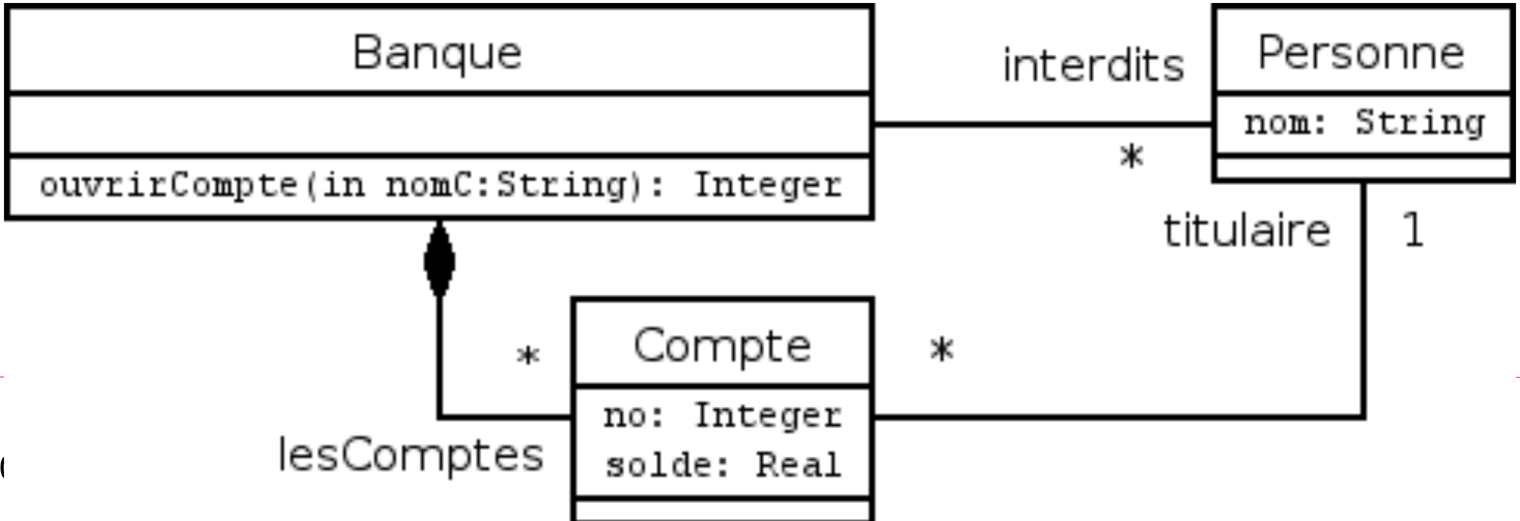
- there is a blacklist
- no more overdraft than 200 EUR



A Revision of the Example: Bank

Opening a bank account. Constraints:

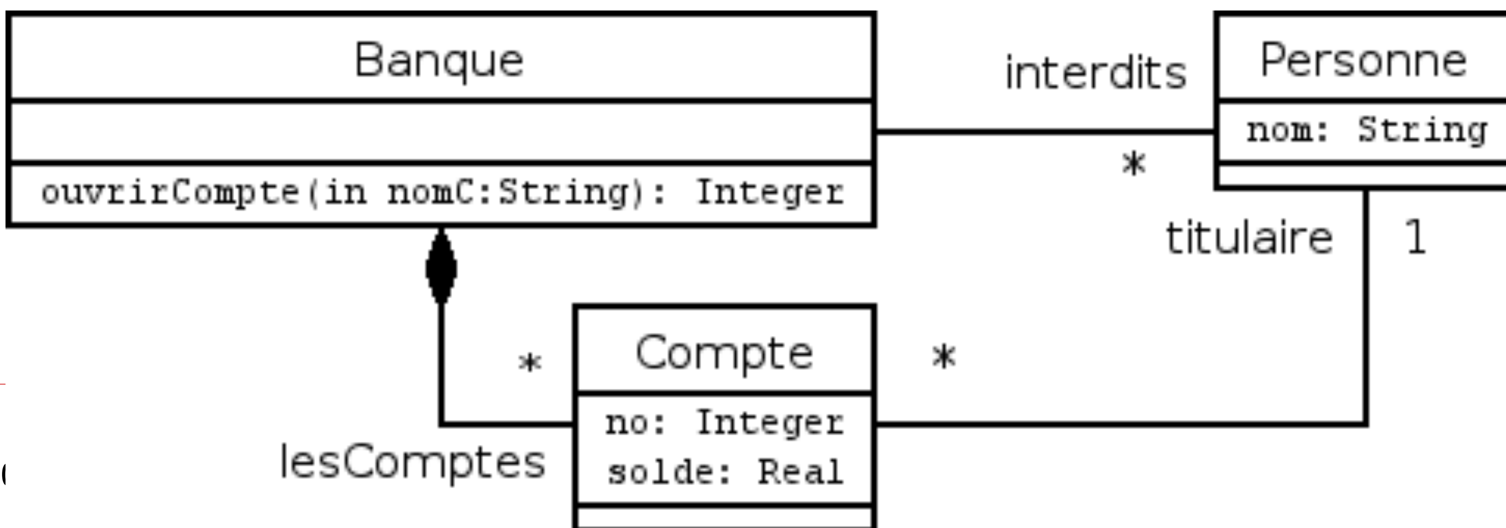
- there is a blacklist
- no more overdraft than 200 EUR
- there is a present of 15 euros in the initial account



A Revision of the Example: Bank

Opening a bank account. Constraints:

- there is a blacklist
- no more overdraft than 200 EUR
- there is a present of 15 euros in the initial account
- account numbers must be distinct.

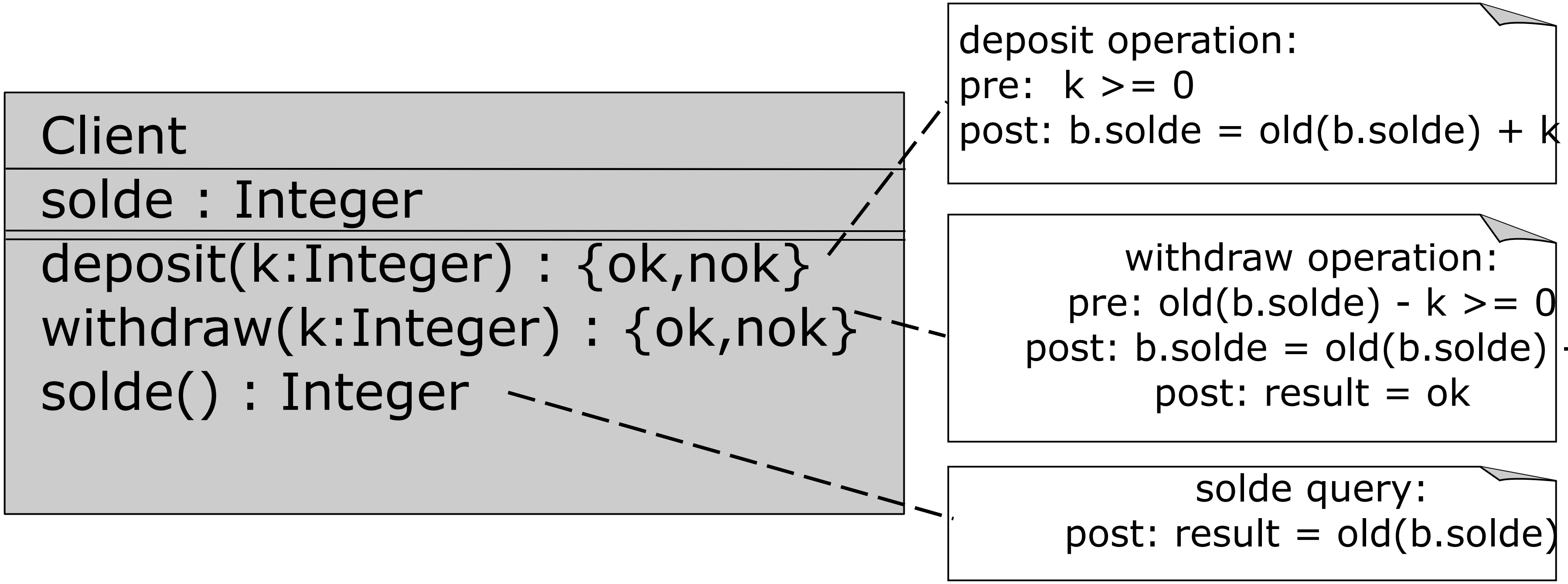


A Revision of the Example: Bank (2)

```
definition pre_ouvrirCompte(b:Banque, nomC:String)≡
                                  ∀p ∈ Personne. p.nom ≠ nomC

definition post_ouvrirCompte(b:Banque,nomC:String,r:Integer)≡
    |{p ∈ Personne | p.nom = nomC| = 1
   ∧  ∀p ∈ Personne. p.nom = nomC ⟶ isNew(p)
        ∧ |{c∈Compte | c.titulaire.nom = nomC}| = 1
     ∧ ∀c∈Compte. c.titulaire.nom = nomC ⟶ c.solde = 15
                                              ∧ isNew(c)
     ∧ b.lesComptes=old(b.lesComptes)∪
                  {c∈Compte | c.titulaire.nom = nomC}
         ∧ b.interdits=old(b.interdits)∪
                  {c∈Compte | c.titulaire.nom = nomC}
     ∧ modifiesOnly({b}∪{c∈Compte c.titulaire.nom = nomC}
                  ∪ {p ∈ Personne | p.nom = nomC})
```

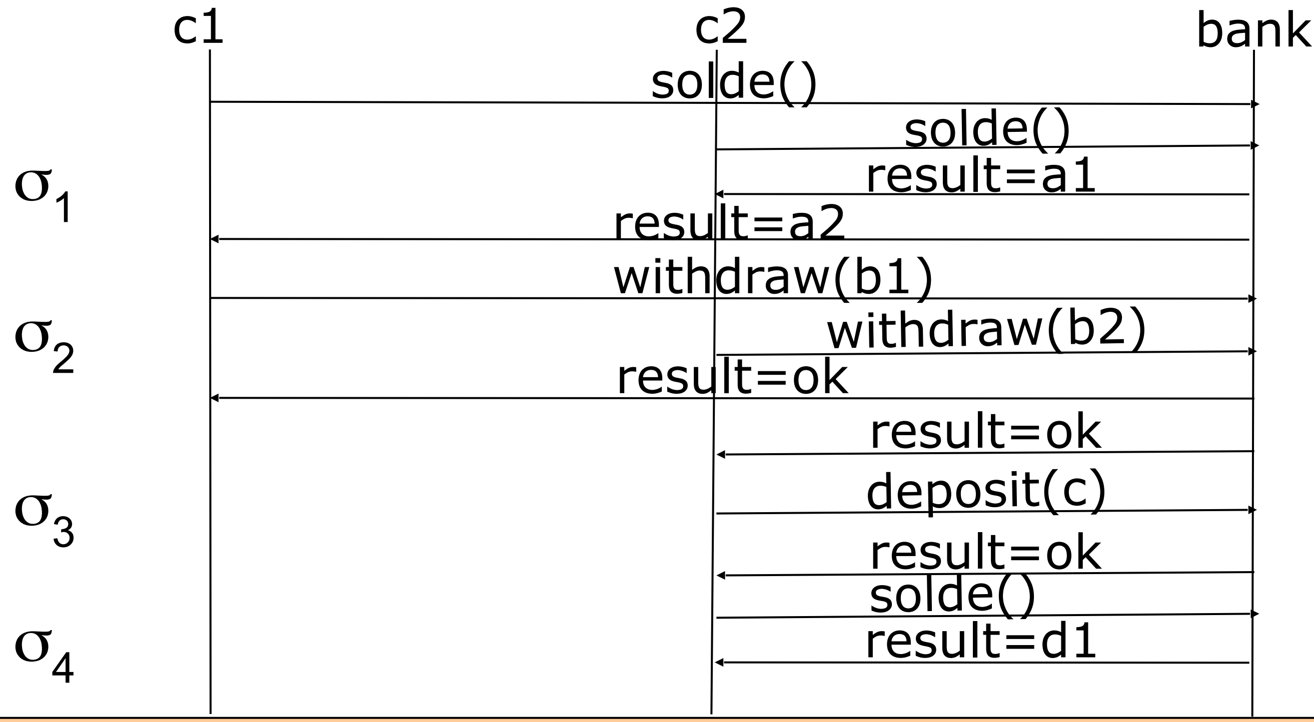
Operations in UML and MOAL

- Example:



Operations in UML and MOAL

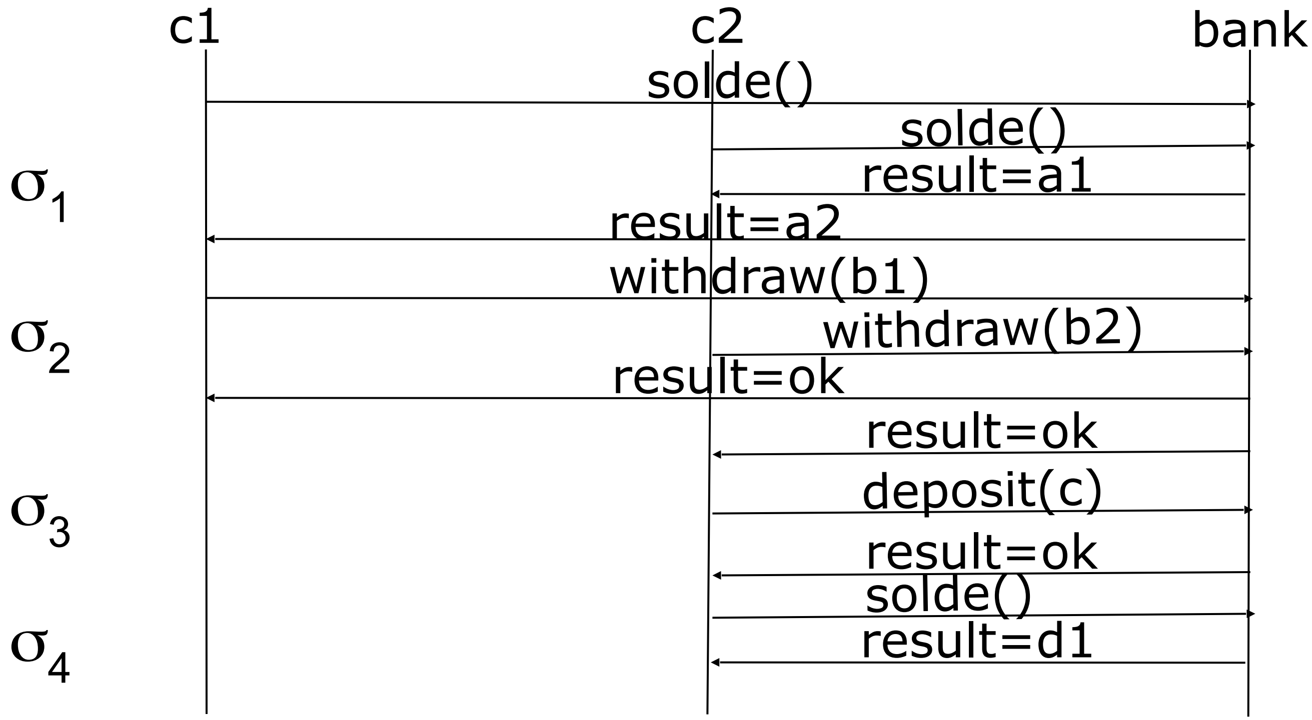
- Abstract Concurrent Test Scenario:



assert c1.solde(σ_4)=a2-b1 $\wedge$ b1 ≥ 0 $\wedge$ a2 ≥ b1

Operations in UML and MOAL

- Abstract Concurrent Test Scenario:



Any instance of b1 and a1 is a test ! This is a „Test Schema" !
Note: b1 can be chosen dynamically during the test !

Summary

Summary

- MOAL makes the UML to a real, formal specification language

Summary

- MOAL makes the UML to a real, formal specification language
- MOAL can be used to annotate Class Models, Sequence Diagrams and State Machines

Summary

- MOAL makes the UML to a real, formal specification language
- MOAL can be used to annotate Class Models, Sequence Diagrams and State Machines
- Working out, making explicit the constraints of these Diagrams is an important technique in the transition from Analysis documents to Designs.