Object Constraint Language (OCL)
(Based on [OMG 2012])

Yih-Kuen Tsay

Dept. of Information Management

National Taiwan University

Outline

- Introduction

- Relation with UML Models

- Values, Types, and Expressions

- Objects and Properties

- Collection Operations

About the OCL

- The Object Constraint Language (OCL) is a **formal** language for writing **expressions** (such as invariants) on UML models.

- It can also be used to specify **queries** over objects.

- OCL expressions are pure specifications **without side effects** (they do not alter the state).

- The OCL is a **typed** first-order language, using a familiar programming language-like syntax.

- The current version OCL 2.3.1 was published in January 2012.

Why OCL

- UML diagrams do not provide all the relevant aspects of a specification.

- Additional constraints expressed by a natural language may be ambiguous.

- Traditional formal languages are precise, but hard to use.

- OCL tries to be formal and yet easy to use.

Note: OCL does not seem to be in wide use, perhaps due to other similar competing specification languages such as JML; however, this exposition shows how UML specifications can be made more precise.

How Can the OCL Be Used

- Queries over objects
- Invariants on classes and types
- Pre and post-conditions on operations
- Guards
- Target sets for messages and actions
- Constraints on operations
- Derivation rules for attributes

Class Diagram Example
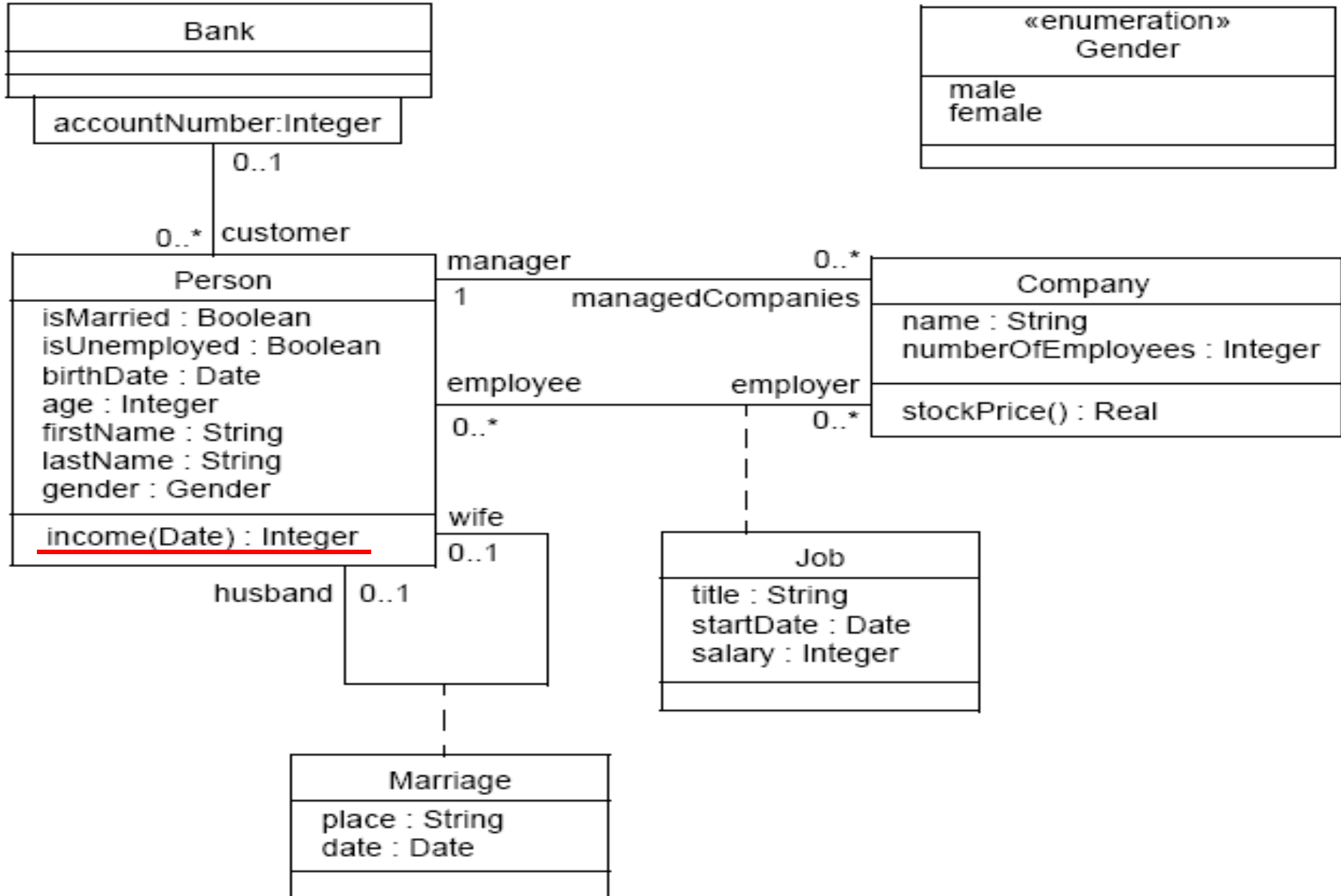
Relation with UML Models: Contexts

- Each OCL expression is written in the **context** of an instance of a specific type.

- The reserved word *self* is used to refer to the contextual instance.

- The context may be specified by a context declaration.

- An explicit context declaration may be omitted if the OCL expression is properly placed in a diagram.

Context for Invariants

- Inside the class diagram, as part of the constraint stereotype <<invariant>>

  Example:

  self.numberOfEmployees > 50

  specifies that the number of employees (of an object in the class Company) must always exceed 50.

- Alternatively (in a separate file),

  context Company **inv**:

  self.numberOfEmployees > 50

Context for Invariants (cont.)

- The keyword *self* may be omitted.
- Also, a different name may be used for *self* :

  context c : Company **inv**:

  > c.numberOfEmployees > 50

- The invariant itself can also be given a name (after **inv**) for later references:

  context c : Company **inv** enoughEmployees:

  > c.numberOfEmployees > 50

Context for Pre and Post-Conditions

- As part of the <<precondition>> and <<postcondition>> constraint stereotypes associated with an operation

- Here, *self* refers to an instance of the class that owns the operation.

- Basic form:

context Typename::operationName(param1 : Type1, …): ReturnType

    pre: param1 > …

    post: result = … (result is a reserved keyword)

Context for Pre and Post-Conditions (cont.)

- Example:

  context Person::income(d : Date) : Integer

  post: result = 5000

- Names may be given:

  context Typename::operationName(param1 : Type1, ...): ReturnType

  pre parameterOk: param1 > ...

  post resultOk: result = ...

Package Context

- When necessary, the package context can be given.

- Package statement:

package Package::SubPackage

context X **inv**:

     … some invariant …

context X::operationName(..)

     pre: … some precondition …

endpackage

Context for Initial and Derived Values

context Person::income : Integer

init: parents.income->sum() * 1%

    -- pocket allowance

    -- the "income" attribute will be defined later

derive: if underAge

    then parents.income->sum() * 1%

        -- pocket allowance

    else job.salary -- income from regular job

    endif

Basic (Predefined) Values and Types

- **Boolean**: true, false

- **Integer**: 1, -5, 2, 34, 26524, …

- **Real**: 1.5, 3.14, …

- **String**: 'To be or not to be', 'This is a system message', …

- **Others**

  - Collection: Set, Bag, Sequence

  - Tuple

Basic Operations (partial list)

- Integer: *, +, -, /, abs()

- Real: *, +, -, /, floor()

- Boolean: and, or, xor, not, implies, if-then-else

- String: concat(), size(), substring()

- Collection: select, reject, forAll, exists, … (to be described later)

Other Types

- ## Classifiers

  - All classifiers of a UML model are types in its OCL expressions.

- ## Enumerations

Sub-expressions: the Use of *let*

context Person **inv**:

    let income : Integer = self.job.salary->sum() **in**

    if isUnemployed **then**

        income < 100

    else

        income >= 100

    endif

Definition Expressions

- Variables and operations may be introduced for reuse across multiple OCL expressions.

- Example:

context Person

def: income : Integer = self.job.salary->sum()

def: nickname : String = 'Little Red Rooster'

def: hasTitle(t : String) : Boolean

    = self.job->exists(title = t)

Previous Values in Post-Conditions

- **context** Person::birthdayHappens()

  post: age = age@pre + 1

- **context** Company::hireEmployee(p : Person)

  post: employees = employees@pre->including(p)

  and

  stockprice() = stockprice@pre() + 10

Previous Values in Post-Conditions (cont.)

a.b@pre.c

    -- takes the old value of property b of a, say x

    -- and then the new value of c of x.

a.b@pre.c@pre

    -- takes the old value of property b of a, say x

    -- and then the old value of c of x.

More about Types and Operations

- Type conformance (like in an object-oriented language)

- Casting (re-typing)
  - ❑ Syntax: object.oclAsType(OclType)

- Precedence rules

- Infix operators
  - ❑ Example: "a.+(b)" as "a+b"

Properties

- More generally, OCL expressions may talk about things called properties.

- A property is one of the following:
  - An Attribute

    context Person **inv**: self.age > 0

  - An AssociationEnd

  - An Operation with *isQuery* (no side effects)

  - A Method with *isQuery* (no side effects)

- Syntax: object.property

- Multiplicities greater than 1 result in collections.

Properties: AssociationEnds

- Starting from an object, we can navigate an association to refer to other objects.

- Example:
  - **context** Person

    inv: self.manager.isUnemployed = false

    inv: self.employee->notEmpty()

- By default, navigation results in a Set.

- When the multiplicity is 1, the result may be treated as a single object.

Collections

- **OCL Collection Types:**
  - ❑ Set
  - ❑ Bag (may contain duplicates)
  - ❑ Sequence (like a bag, but ordered)
- **Collection literals:**
  - ❑ Set { 1, 2, 5 }
  - ❑ Bag { 1, 3, 4, 3 }
  - ❑ Sequence { 1..10 }
- **The OCL defines many operations on collections.**

Collection Operations

- Select

  context Company **inv**:

  self.employee->select(age > 50)->notEmpty()

- Reject

  context Company **inv**:

  self.employee->reject(isMarried)->isEmpty()

- The select and reject operations always give a sub-collection of the original collection.

Derived Collections

- From a collection, one may also derive a collection of *different* objects.

- Examples:

  self.employee->collect(birthDate)

  self.employee->collect(p|p.birthDate)

  self.employee->collect(p:Person|p.birthDate)

- The result above is a Bag, which may be turned into a Set:

  self.employee->collect(birthDate)->asSet()

Collection Operation: ForAll

context Company

    inv: self.employee->forAll(age <= 65)

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

    inv: self.employee->forAll(p : Person | p.age <= 65)

context Company **inv**:

    self.employee->forAll(e1, e2 : Person |

    e1 <> e2 implies e1.firstName <> e2.firstName)

Collection Operation: Exists

context Company **inv**:

   self.employee->exists(firstName = 'Jack')

context Company **inv**:

   self.employee->exists(p | p.firstName = 'Jack')

context Company **inv**:

   self.employee->exists(p : Person | p.firstName = 'Jack')

The Iterate Operation

- *Reject, Select, forAll, Exists,* and *Collect* can all be described in terms of *iterate*.

- Example:

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

-- is identical to

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

    acc->including(x.property))

-- here x is the *iterator* and acc is the *accumulator*

Features on Classes Themselves

- It is also possible to use features defined on classes themselves.

- A predefined feature is *allInstances*, which gives the set of all instances at the time of evaluation.

- Example:

context Person **inv**:

Person.allInstances()->forAll(p1, p2|

p1 <> p2 **implies** p1.firstName <> p2.firstName)