Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

# Specification with OCL

Jurriaan Hage
Slides adapted from Birgit Demuth, TU Dresden
e-mail: jur@cs.uu.nl
homepage: http://www.cs.uu.nl/people/jur/

Department of Information and Computing Sciences, Universiteit Utrecht

September 16, 2009

# Overview


Introduction

The basics: invariant and pre/post conditions

OCL types

Collections in more detail

Further resources

# 1. Introduction

# Formal specification



- **Formal specification** consists of describing the *system requirements* using *formal notation*
- The system requirements are a more detailed version of the *user requirements*
- The formal notation allows you to write things down in a *precise* and *unambiguous* manner
- Formality enables others (including computers) to help you.
- For example: automated consistency checks
- Formality requires discipline and hard work
- Examples: Z, Object Constraint Language, CASL

# The Object Constraint Language



- OCL is part of the UML.
- We consider version 2.0 (of the UML).
- Pure language: no side-effects.
- Consists of so called OCL expressions.
- Not necessarily executable
  - Code generation within MDA
- Mainly to specify restrictions on a system model.
  - Thereby making the models unambiguous, more precise and well-defined.
- But may also be used to specify *initial values* and *derivation rules*.
- Our focus is on restrictions/constraints, but also consider derivations.

# An aside: UML and MOF



- MOF = Meta-Object Facility (M3).
- Yet another OMG standard (ISO/IEC 19502:2005).
- A language for specifying modelling languages
  - UML (M2)
  - MOF itself
- UML is used to write down M1 Models, e.g., class diagrams.
- Which describe M0, i.e., real world, objects.
- MOF is to metamodelling as EBNF is to grammars.
- OCL is defined to apply to all M2 models expressible in MOF.
- Part of QVT, a languages for querying, viewing and transforming metamodels.

# Constraints



- Many synonyms: predicate, assertion, contract
- In our case, constraints are typically
  - invariants
  - pre-conditions
  - post-conditions
  - guards
- Constraints may occur in various places in the UML, but we focus on constraints for the class diagram.
  - Necessary, since many restrictions cannot be made explicit in the diagram.
- In this case, OCL expressions often capture *business rules*.
  - I.e. domain specific information, not implementation specific information.
- Compare: Enhanced Entity Relationship model with relational algebra.

# A question for you:



$$\exists\, p.\texttt{inclass}(p) \wedge \neg\texttt{knows}(p, \texttt{predicatelogic})\ ?$$

# A question for you:



$$\exists\, p.\mathtt{inclass}(p) \wedge \neg\mathtt{knows}(p, \mathtt{predicatelogic})\ ?$$

In other words:
Is anyone in this classroom not familiar with predicate logic?

Constructively: who is not?

# Context



- OCL constraints are special kinds of predicates
  - attached to a particular UML diagram
  - explicitly refers to a context for evaluating the constraint.

```
context Driver inv:
  age >= 18
```

# A UML reference model (our context for today) 1

Person
-name:String
-age:Integer
-title:String
-gender:Gender

<<enumeration>>
Gender

{male,female}

Date
-today:Date
+isBefore(t:Date):Boolean
+isAfter(t:Date):Boolean
+equal(t:Date):Boolean

utility class

Meeting
-title:String
-day:Date
-start:Integer
-end:Integer
-isConfirmed:Boolean=false
+cancel():void
+confirm():void
+numParticipants():Integer
+shift(d:Integer):void
+duration():Integer
+inConflict(m:Meeting):Boolean

Teammember
-role:String
+numMeetings():Integer
+numConfMeetings():Integer

2..* participants — 0..* meetings
1 moderator — 0..*
2..* members — 1 team

Team
-name:String

1 for — 0..*

Teammeeting

Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

# 2. The basics: invariant and pre/post conditions

# Invariants



General form:

```
context <classifier>
inv [<constraint name>]: <Boolean OCL expression>
```

- Invariants should hold for all objects that satisfy the context classifier, and at all times.
- The object under scrutiny may be explicitly referred to as *self*.
- Constraints may be given a name.

# Example: meetings end later than they start



```
context Meeting
inv: end > start
```

or if you prefer more explicitly

```
context Meeting
inv: self.end > self.start
```

or to name the constraint

```
context Meeting
inv startEndConstraint: end > start
```

# Preconditions and postconditions



- Methods can be specified by pre- and postconditions.
- As in axiomatic approach to program correctness.

$$\{x >= 0.0\}\ \ y = \mathrm{sqrt}(x);\ \ \{y = x * x \wedge y >= 0.0 \wedge x >= 0.0\}$$

- If the precondition holds, then after execution of the statement, the postcondition is guaranteed to hold.
- A checker for OCL should then verify:
  - The preconditions are satisfied when a method call is executed.
  - Afterwards, the postcondition holds.
- The caller is responsible for satisfying the precondition, the method for satisfying the postcondition.

# Example precondition



```
context Meeting::shift(d: Integer)
pre: self.isConfirmed = false

context Meeting::shift(d: Integer)
pre: d>0
```

or combine them as follows:

```
context Meeting::shift(d: Integer)
pre: self.isConfirmed = false and d>0
```

# Precondition general form



```
context classifier::operation (<parameters>)
pre <constraint name>: Boolean OCL expression
```

# Example postcondition



```
context Meeting::duration():Integer
post: result = self.end - self.start

context Meeting::confirm()
post: self.isConfirmed = true
```

– keyword *result* refers to the values returned by *duration()*.

# Postcondition general form



```
context classifier::operation(<parameters>)
post <constraint name>: Boolean OCL expression
```

# A more complicated postcondition



```
context Meeting::shift(d:Integer)
post: start = start@pre + d and end = end@pre + d
```

- For destructive operations, e.g., shift, we must be able to refer to object values as they were before and after execution.
- *@pre* is a modifier to refer to the before/old values.
- It only makes sense to use *@pre* in postconditions.

# Behaviour specifications



- pre and post are about input and outputbehaviour only.
- How do you you specify behaviour of the operation?
- Example: the *administrator* must be sent a *notify* message whenever a meeting is shifted.
- You may also specify arguments to the messages.
- Similar to Mock Objects (see later in the course)

```
context Meeting::shift(d:Integer)
post: administrator^notify("Meeting shifted")
```

# 3. OCL types

# OCL types



- Built-in types:
  - Boolean, Integer, Real, String, OCLAny (=typevar)
  - Collection types (parametric polymorphic/generic) such as Collection, Set, OrderedSet, Bag, Sequence
- User-defined types (OCLType):
  - class/model type with attributes (*self.value*), operations, class attributes (*Class::default*), class operations and *association ends/roles*
  - enumerations: *Gender*, *Gender::male*

# OCL type checking



- OCL is strongly/statically typed
- Subtyping relation, e.g., *Integer* and *OCLType* are subtypes of *OCLAny*.
- Extended to conformance relation:
  *Integer* conforms to *OCLAny* and *Real*
- *Set<Integer>* conforms to *Set<OCLAny>*, but also *Collection<OCLAny>)*.
- OCL expressions are only correct if all types conform.
- Uses Liskov's principle: $T1$ conforms to $T2$ if any instance of $T1$ may occur wherever an instance of $T2$ is expect.
- Why are collection classes not invariant as in Java?

# Navigation expressions



- Essential and special for OCL.
- Use names of association ends to navigate from one object to another through a relation.
- The type of a navigation expression depends on the cardinality of the relation
  - And type checking really helps there

# Example navigation expression



```
context Meeting
inv: self.moderator.age > 35
```

# Example navigation expression



```
context Meeting
inv: self.moderator.age > 35
```

```
context Meeting
inv: self->collect(participants)->size() >= 2
```

or in shorthand notation

```
context Meeting
inv: self.participants->size() >= 2
```

# Navigation to associated class (attributes)



- Given a multi-relation between *Course* and *Student*
- And a class *Grade* associated with the relation, in which the grade is stored in the attribute *value*.
- Specify that the average should be at least 5.5.
- Note the lower case letter to start *grade*!

```
context Course
inv: self.grade.value->sum()
                  / self.grade->size() >= 5.5
```

# Undefinedness



- What happens when you navigate from an object that does not have that role?
- You get *OclVoid*.
- Whenever part of an expression is undefined then the whole expression is undefined
- Except when shortcircuiting is used, e.g.,
  *False implies undefined* equals *True*
- You can check explicitly for undefinedness with *oclIsUndefined() : Boolean*.

# 4. Collections in more detail

# Dot notation verus arrow notation



- Dot notation is used for non-collection attributes and methods.
- Arrow notation is used for collections.
- Sometimes both are possible `self.manager` can be referred to as a set of Persons or a single Person.
- View dot or arrow as a form of cast/coercion.
- Allows to use set methods like `nonEmpty` on optional arguments:
  `self.license->nonEmpty() implies age >= 18`

```
context Company inv:
self.manager->size() = 1

context Company inv:
self.manager.age > 40
```

# Built-in operators for collections



- Quite a few, many of them overloaded.
- We've seen *nonEmpty()* and *size()*.
- Transformations between collection types: *asBag()*.
- *including(object)*, union, intersection etc.
- *flatten()* removes one level of nestedness and always returns a set.
- Some operations work only on som collection types, e.g., *first()* that need ordering.

# Iterating over collections



- *any, collect, exists, forAll, isUnique, one, select, reject*
- *select* is much like Haskell's *filter*: return the items that fulfill the predicate
- *reject* is like using *select* with the negation of the predicate
- *collect* is like Haskell's *map*: apply something to every element in the collection
- Result of *any* is not a boolean, but gives an item that fulfills the condition.
- *one* versus *isUnique*
- See Section 11.7 of OCL standard for a complete description.

# Iterate



- All operations are expressed in terms of *iterate*, which binds *element* and a *result*.
- Essentially a fold operation (but which?).

```
Collection->iterate (
  element : Type1;
  result  : Type2 = <defaultexpression> |
  <expressionbody>)
```

```
Set {1,2,3}->iterate
   ({i : Integer; sum: Integer=0 | sum + i})
  -- equivalent to sum()
```

# Some more examples



A teammeeting has all team members present (and no others)

```
context Teammeeting
inv: participants->forAll(team=self.for)

-- or

context Meeting inv:
oclIsTypeOf(Teammeeting) implies
  participants->forAll(team=self.for)
```

# Postcondition examples



```
context Teammember :: numMeeting()
post: result=meetings->size()

context Teammember :: numConfMeeting():Integer
post: result=meetings->select(isConfirmed)->size();
```

# Derivation rules (OCL2 only)



Derived attribute:

```
context Team::size : Integer
derive:members->size()
```

Derived association:

```
context Meeting::conflict:Set(Meeting)
derive: select(m | m<>self and self.inConflict(m))
```

# Initial values



- Initial values must satisfy possible invariants, but only at object construction time.

```
context Meeting::isConfirmed : Boolean
init: false

context Teammember::meetings : Set(Meetings)
init: Set()
```

# Local definitions with let



- For controlling the complexity of definitions
- OCL even allows you to define *noConflict* as a new attribute.

```
context Meeting inv:
  let noConflict : Boolean =
     participants.meetings->forAll(m|m <> self and
     m.isConfirmed implies not self.inConflict(m))
  in isConfirmed implies noConflict

context Meeting def:
  noConflict : Boolean = ....
```

# Not considered here

- Packaging constraints
- Using OCL to specify query operations of the eventual system (instead of just properties thereof)
- Using OCL beyond UML class diagrams.

# Limitations of OCL



- No inconsistency checking
- Post conditions specify the changes, leaving implicit that everything else is unchanged.
  - In Z for example, unchangedness is explicit
- Limited recursion (I guess to primitive recursion)
- *allInstances()* problem: works on, e.g., *Person*, but not on the *Integer* type.

# 5. Further resources

# Further resources

- Assignment is on the course website.
- Dresden OCL Portal:
  `http://st.inf.tu-dresden.de/oclportal/`
- Dresden OCL Toolkit:
  `http://dresden-ocl.sourceforge.net/index.php`
- `http://www.omg.org/spec/OCL/2.0/PDF/`, particularly pages 19 – 46.
- Praktisch UML, 4e editie, by Warmer and Kleppe has a few chapters on OCL (in Dutch).
- The Object Constraint Language, Second Edition, by Warmer and Kleppe (but could not find the book easily).