Ingredients

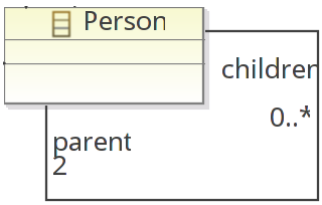
Reading: A standard reading on OCL is [4]. Please read chapter 3, which contains guidelines for writing constraints (see esp. Section 3.10 on tips and tricks).

Current OCL specification can be found at http://www.omg.org/spec/OCL/Current/. As usual do not use it for studying OCL. Learn the language from other materials, but then try to skim and read fragments of OCL spec, to get a feeling of its flavor. Chapter 7 (The OCL Language Description) is certainly worth looking into.

Jackson contrasts OCL with his specification language Alloy, which is more relational, than first-order in flavour. His book [3] and the journal paper on Alloy [2] contain short and interesting critiques of OCL. Short (few pages only, to be found in the final sections), but very useful if your project is about constraints.

1 Introduction

In the last exercise session we looked at a model similar to the following diagram:



However we were unable to guarantee well formedness of this diagram easily. The problem was that we could not easily syntactically guarantee that if A is a parent of B, then the two are different (and that a person cannot be a parent of itself).

Today we look into more expressive ways of specifying such constraints easily, using the Object Constraint Language.

2 General Characterization of OCL

> The Object Constraint Language (OCL) [is] a formal language used to describe expressions on UML models. These expressions typically specify invariant conditions that must hold for the system being modeled or queries over objects described in a model. Note that when the OCL expressions are evaluated, they do not have side effects (i.e., their evaluation cannot alter the state of the corresponding executing system).[OCL specification]

⚠ We are going to describe OCL referring to the knowledge you already have about programming languages. :

OCL is a **declarative** programming language, with well defined syntax and semantics. OCL is **first order predicate logics** given a programmer friendly syntax (no **quantifiers**, or quantifiers are hidden as **collection iterators**).[1]

OCL is declarative (no variables, no state, but no higher order - so this is not your new Haskell). OCL is **strongly typed** (like for instance Java).

Constraints express **invariants** over classes.

They can also be used to express **pre-** and **post- conditions** for operations, but we are not concerned with them in this lecture.

OCL can also be used to specify body of operations independently of programming language; again we are not concerned with that here; and for defining and deriving new associations and attributes, initial values. Standard reference on all aspects of OCL is [4].

In the DSL design context, we are primarily interested in using OCL to constrain class diagrams further, with some well-formedness conditions. Typically the diagram itself is not sufficiently expressive to do this. In principle, one could write the well-formedness rules in natural language, but then we are not able to leverage automatic constraint checks provided by EMF.

In EMF, OCL can be used to write integrity constraints (similar to data integrity constraints, that you know from databases) and derived value expressions. The latter are used to specify default values of attributes, etc.

3 OCL Syntax and Semantics by Example

A usual invariant constraint has the following structure:

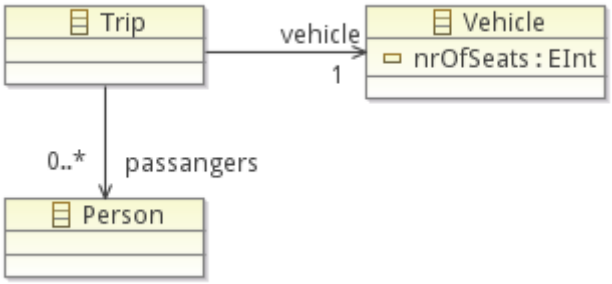
[1] Technically speaking, because OCL allows function definitions and recursion, it is more expressive than first order logics. It can for example express transitive closure. You stay within first-order logics if you do not use function definitions and recursion. Also we do not use operations (functions) in language design, which is the main application of class diagrams and OCL in this course.

```
context ClassName

inv: OCL-expression
```

Such an invariant will apply to all objects of the given class. The expression must be a Boolean expression (a predicate).

For example in the following model we are interested in describing trips that combine a number of persons (passengers) in a vehicle:

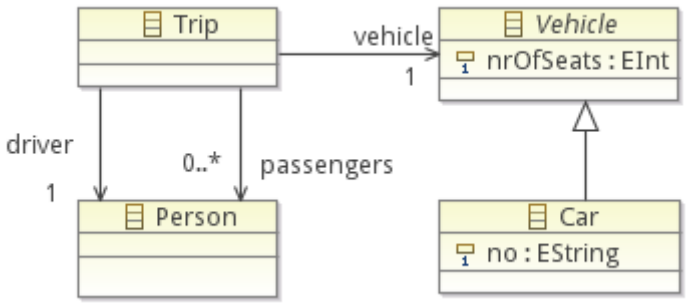


A natural integrity constraint is that the vehicle associated with the trip needs to be large enough to accommodate all the involved passengers. This constraint cannot be expressed directly in the diagrammatic language of class diagrams. It can be stated in OCL as follows:

```
context Trip
inv: passengers->size() <= vehicle.nrOfSeats
```

The type of the entire constraint expression is Boolean. It is written in the context of a Trip, so it applies to all instances of Trip. It contains numeric expressions, and a collection expression (`passengers->size()`). OCL contains a rich expression language that is syntactically similar to expression languages of modern object-oriented programming languages.

We want to add cars, and drivers to our model:



Further, we would like to make sure that a driver is listed on the passenger list. This is enforced using the following constraint:

```
context Trip
inv: passengers->includes(driver)
```

Like in Java, names of attributes and references are resolved locally in the context. So the above constraint is equivalent to:

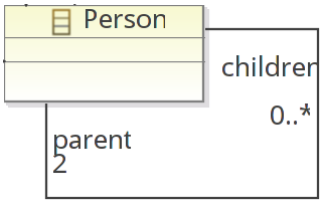
```
context Trip
inv: self.passengers->includes(self.driver)
```

We can also require that every Car is identified by its registration plate:

```
Context Car
inv: Car::allInstances()->isUnique(no)
```

The above is an example of getting a collection of all instances of a meta-class.

A common constraint pattern involves a kind of cyclic commutativity constraint. These are sometimes hard to understand, so we have a look at a special example here. A person can have zero or more children, and a child has exactly two parents:



It is expected that for a given parent, the parent is included in the set of parents of each of its children:

```
context Person
inv: self.children->forAll(c | c.parent->includes(self))
```

and dually each person is included in the set of children of its parents:

```
context Person
inv: self.parent->forAll(p | p.children->includes(self))
```

⚠ **Q. Create an instance of the class diagram that violates one of these constraints.**

Remark: In this particular case this integrity constraint can be maintained automatically by EMF, if the two references (parents and children) are related with the **EOpposite** tag. So OCL is not strictly required. This however, only works for simple cases. In more complex examples, you would need to write constraints like that in OCL.

4 Crash Summary of OCL

- In OCL navigation works like in Java:

  `ClassName.attribute.relation.operation().attribute ...`

  Operation calls are allowed, but then you should be careful that the operations have no side effects.

- Invariants can be named, to allow easier references:

  ```
  context Trip

  inv driverIsPassenger:  passengers.include(driver)
  ```

- Referring to enumerations

  `EnumerationClass::Literal`; for instance `Color::red`

- Supported operators include: `implies and or xor not`, `if then else` (this is the same as `? :` in Java); `>= <= > < = <> + - / * a.mod(b) a.div(b) a.abs() a.max(b) a.min(b) a.round() a.floor() string.concat(string) string.size() string.toLower() string.toUpper() string.substring(int,int)`

- If navigation arrives at more than one object (via a link with multiplicity exceeding 1), we obtain a collection as the value. Use collection operators on this:

  - `Course.students->size ()` — size of the collection class
  - `Course.students->select (isGuest())-> size()`

    select the guest students, and count them
  - `Course.students->select(isGuest())->isEmpty()`

    no guest students are allowed in the course.
  - `Course.students->forAll(age >= 18)`

    this course is only for adult students
  - `Course.students->forAll ( s | s.age >= 18)`

    equivalent to the above, but sometimes it is convenient, with a name for the iterated objects
  - `Course.students->collect(age)`

    a collection of ages on this course.

- Notice that collection operators are introduced with an arrow (unlike in Java!)

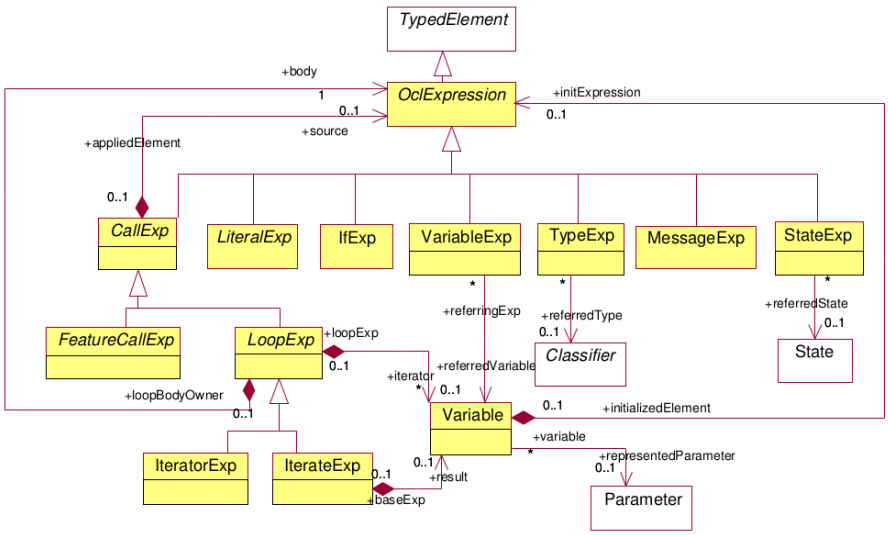


Figure 1: Core OCL Model from OCL specification ver. 2.2, http://www.omg.org/spec/OCL/2.2/

- Other collection operators:
  `notEmpty`, `includes(object)`, `includesAll(collection)`, `union(collection)`
- Four types of collections: sets, bags, ordered sets, sequences (an ordered bag)
- When you navigate through more than one association with multiplicity greater than 1 you end up with a bag.
- When you navigate just one such association you get a set
- Now you get an ordered-set or a sequence if any of these associations was marked as `ordered`
- Let expressions:

```
let guests = Course.students-select(isGuest())
in guests->forAll( age >=18 )
                  and guests->forAll( age <= 20 )
```

- Single-line comments begin with two hyphens (like in Haskell):
  `-- this is a comment`
- Multi-line comments look `/* like in java */`

5 OCL as a Domain Specific Language

OCL is a DSL itself. Its concrete syntax is specified using a context free grammar, and its abstract syntax is specified using a meta-model. The core part of this meta-model is shown

in Fig. 1. Figure 2b shows how this metamodel fits the general layered architecture of meta-modeling, that we have shown previously.

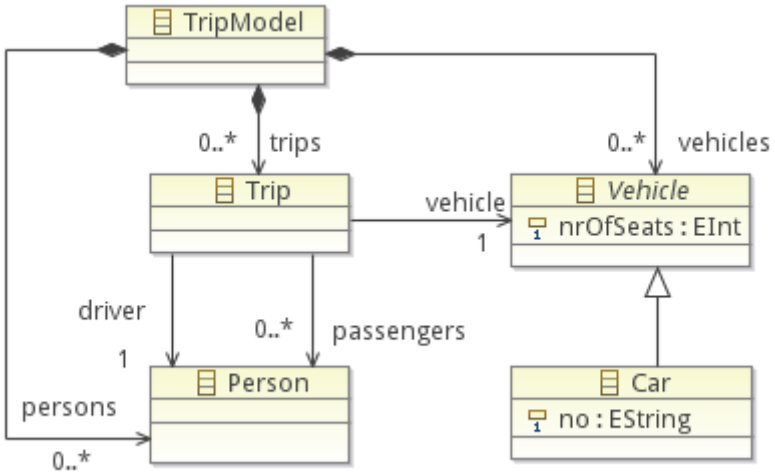
Figure 2a explains another point: that OCL Language, and OCL constraints semantically are similar to ecore itself, so one can also draw them one level higher. This is because the constraints written at a given level, constrain instances one level below. Thus meta-model constraints have similar semantic effect to the meta-models (M2), and thus OCL is a specification language at M3, similarly to ecore.

6 EMF and OCL in Practice

The easiest way to play with OCL is to create an ecore Meta-model, derive an instance of this model dynamically, and then open interactive OCL console in Eclipse, which allows you to type in constraints and have them evaluated immediately. [2]

To derive a dynamic instance create an ecore meta-model, and right click on one of its classes. Choose “create dynamic instance”. You will be asked to give a name of the xmi file, in which the instance will be stored. Now the editor opens and you can add children to the object created. Attributes and references can be specified in the properties view (right click on an object and choose “Show properties view”).

In EMF all objects in a model need to be owned by some model class through composition (black diamond). You will not be able to create them otherwise. Thus it is usual practice to create one more meta-class representing the model itself, which owns all the other model elements. So our example looks more or less like below (note the TripModel class):



[2] In installation Eclipse (Indigo), for some reason I need to install “OCL examples and editors“ component from the mirror ”Indigo - http://download.eclipse.org/releases/indigo“, before the console was available

Default editors for your models only provide for editing what was specified in our metamodel. So for example three instances of Person are visually indistinguishable. If you want objects to have visible identity, you need to give them names or identifiers. Often this is done by creating one abstract class *NamedElement* and making this class a generalization of all other classes in the model (then sometimes it is also convenient to make this class owned by the model class, instead of drawing compositions to all individual concrete classes like in the example above). If the name is unique, then it makes sense to mark that it has a property ID, in the advanced properties of the **EAttribute**. Then the tree editor will display it next to the object.

Before writing any constraints, you can check if your model satisfies the multiplicity constraints of the meta-model. You do this by selecting "Validate" in the context-menu of the instance model.

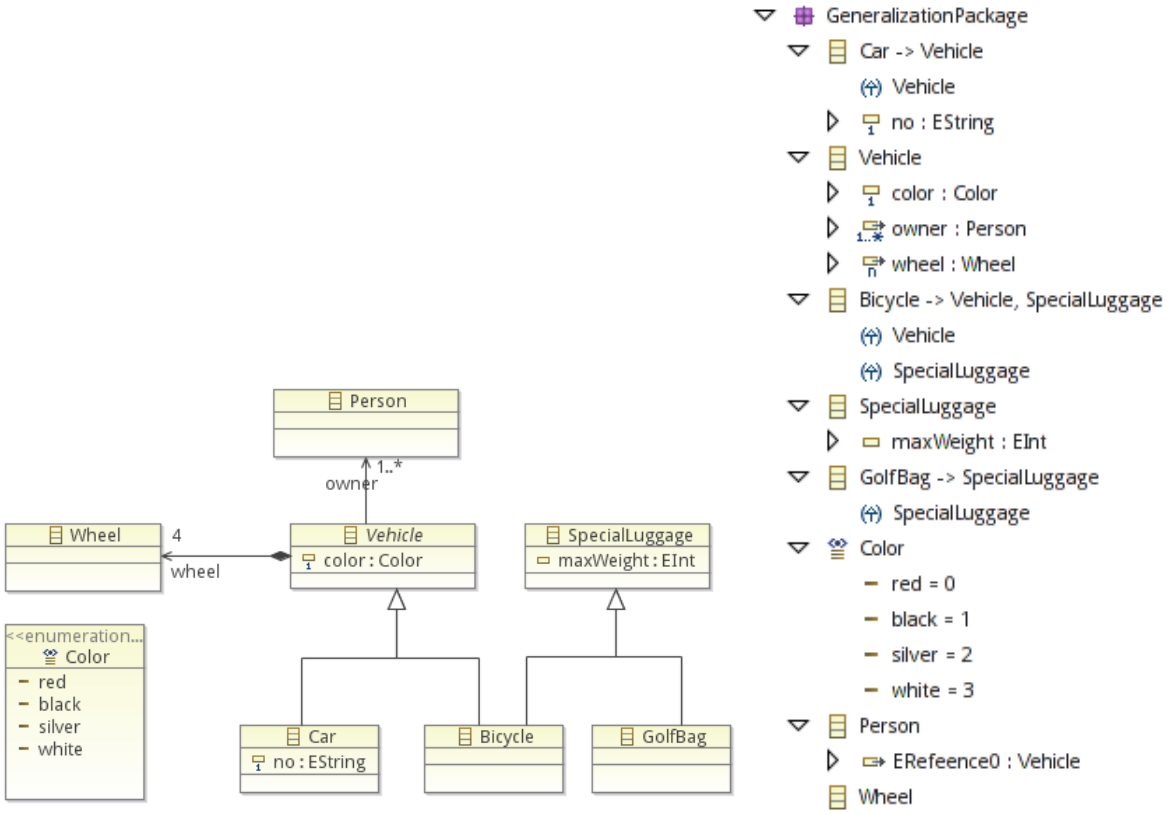
Once you created the instance dynamically, right-click on one of its elements, and open OCL console. In the console you can directly write invariant constraints. Omit the context and the `inv` keyword — instead make sure that the context element is selected in the model editor.

Note that constraints can be evaluated on M1 and M2 level (see console toolbar). At the M1 level, you can select a meta-class as context, and check if the constraint parses correctly. At the M2 level, you can select a model element (such as a concrete trip), and check if the constraint parses and evaluates to true.

This is very convenient to develop constraints (simple errors can be found immediately) — after that remember to store your constraint in a suitable file, or in the model, depending how you are going to use this.

Note: Code completion works in the OCL console (ctrl-space).

Finally, note that EMF by default supports tree editors, not diagram editors like above. This is a small syntactic difference, that is easy to get used to. Below you find a class diagram (one of the above examples) and a corresponding Tree View of the same diagram. Elements nested by composition are nested in the tree - so for instance attributes are nested under classes.



Such an editor can be automatically generated for your languages (or it can even run reflectively, as an interpreter for your meta-model). In my experience, being the main editor, the tree editor is much more stable than the diagram editor. The diagram editor stores the layout information in a diagram file, and occasionally the two files (models and diagrams) get out of sync, leading to complex errors. In such situation try to edit the ecore file in the tree editor, and if this does not help, simply delete the diagram file and work always in the tree editor (or reinitialize the diagram).

In the tree editor "-1" is used instead of "*" for multiplicity constraints.

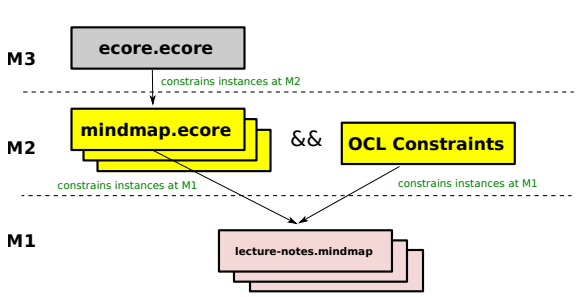
References

[1] Frank Budinsky, David Steinber, Ed Merks, Raymond Ellersick, and Timothy J. Groose. *Eclipse Modeling Framework*. Addison-Wesley, 2004.

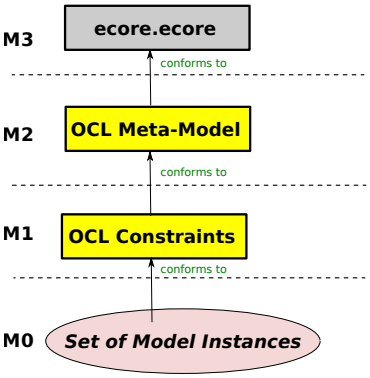
[2] Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, 2002. http://doi.acm.org/10.1145/505145.505149.

[3] Daniel Jackson. *Software Abstractions*. MIT Press, 2006.

[4] Jos Warmer and Anneke Kleppe. *The Object Constraint Language*. Addison-Wesley, 2003.



(a) OCL constrains instances in the same way as ecore does



(b) OCL as a language in the meta-modeling hierarchy

Figure 2: Two views on OCL in the metamodeling hierarchy