3C05/D22 Problem Class 18/3/2005

Modelling with UML, MOF, and OCL

James Skene, a research fellow in the department has developed an open-source OCL parser and type checker. In this problem class you will use his software to develop a type-correct MOF model including OCL constraints. His software is at an alpha-testing stage, and you may encounter bugs during the exercise. You will use the sourceforge.net bug-tracking software to submit bug reports if necessary.

Accompanying pages provide documentation for James's system, and instructions for using sourceforge.

EXERCISE:

1. Develop a MOF model of a unix- or windows-like file system. Your model should at least include directories and files, and may include advanced features such as symbolic links and user and group permissions. The model may be a UML class diagram created in the Posiedon CE editor and imported into the King editor, or you may use the concrete syntax of the MOF supported by the King editor. Regardless of the approach you take, you will end up with your model in the concrete syntax of the MOF.
2. Embed some OCL constraints into your model, by including them in the relevant places in the specification created in the first step. You should at least write the constraints that files should be uniquely named in a directory, and that the directories should never be nested in themselves, even transitively. This last constraint will require you to define a query operation on at least one of your classes.

EMOF/OCL Syntax

EMOF is a language for describing data structures. Normally it looks just like UML class diagrams. However, these require a graphical editor that would have been hard to develop, so EMOF/OCL editor can either import a diagram from an XMI file, or use a textual syntax to represent these models.

Like UML class diagrams, EMOF models can contain embedded OCL expressions. This document describes the concrete syntax for EMOF/OCL models used in our editor.

specification ::= “specification” <name> “{“ (class | package) * }

package ::= “package” <name> “{“ (type | package)* “}”

type ::= primitive | class | enumeration

primitive ::= “primitive” <name>
(
(“OCL_INTEGER” | “OCL_STRING” | “OCL_BOOLEAN” | “OCL_REAL)
(“IDENTICAL”)?
)?

enumeration ::= “enumeration” <name> “{“
(<name>)+
“}”

class ::= “class” <name> (“extends” typePath (“, “ typePath)*)? “{“
(attribute | operation | invariant)*
“}”

attribute ::= (“component”)? <name> “:” typePath (multiplicity)?
(“opposite” <name>)?

operation ::= <name> “(“ (parameter (“,” parameter)*)? “)”
“:” typePath (multiplicity)? (“=” “{“ expression “}”)?

invariant ::= “invariant” “{“ expression “}”

typePath ::= (“::”)? <name> (“::” <name>)*

multiplicity ::= “[“ (“*” | <integer> (“,” (“*” | <integer>))?) “]”

Names may be any combination of letters and numbers starting with a letter, or may be double quoted Java-style strings.

The expression production corresponds to any valid OCL2 expression. According to this syntax, any OCL expression must be surrounded by braces.

Using the EMOF/OCL editor

Start the king editor using

```
% cd /cs/research/sse/common0/rigel/sw/king
% java –jar emofocl.jar
```

The EMOF/OCL editor supports 3 different types of editing windows. These can be accessed from the 'Editors' menu.

The three types of editor that will be relevant to the exercise are:

The UML editor – This allows you to load and view a UML model from an XMI file. The button marked 'export' on the editor will convert the UML model to the MOF textual syntax and open the King specification editor to display the new model.

The EMOF/OCL specification editor – This allows you to edit a text file to contain a MOF/OCL model. Clicking the parse button will cause the specification to be parsed. As the specification editor is quite primitive, you may prefer to copy and paste between the specification editor and a text editor of your choice.

The EMOF/OCL model editor – If your specification parses successfully, this editor will display the specification converted into the abstract syntax of OCL/MOF.

Any syntax or type errors are reported in the console window, accessible from the tools menu, and are also reported as exceptions in the terminal from which you ran the tool.

REPORTING BUGS:

If the editor does something that you think it shouldn't, please report it. You can either email me a report at j.skene@cs.ucl.ac.uk, or preferably use the sourceforge bug tracking system to do it. This can be accessed by clicking 'bugs' in the main menu of the project's home:

http://sourceforge.net/projects/uclmda/

You will need to be registered to the sourceforge site and logged on to submit a bug report.

The EMOF/OCL application is open source, so feel free to use it as you please. I have a number of ideas for projects related to the code if anyone is interested in contributing.

Using the Poseidon UML Editor

Poseidon CE from Gentleware is a free UML editor. It will export models in an XMI format (an XML based file format) that is compatible with the King editor's UML model editor.

Start Poseidon using:
% /cs/research/sse/common0/rigel/sw/king/poseidonCE-3.0.1/bin/poseidon.sh

You will need to register the product on its first use. Use the online registration service to do this.

You can use Poseidon to produce the structural part of your MOF/OCL model. The types and packages you create in Poseidon will translate into types and packages in your MOF specification. Attributes of classes will convert into attributes in the MOF models. Associations in the model will convert into opposed pairs of attributes in the MOF model.

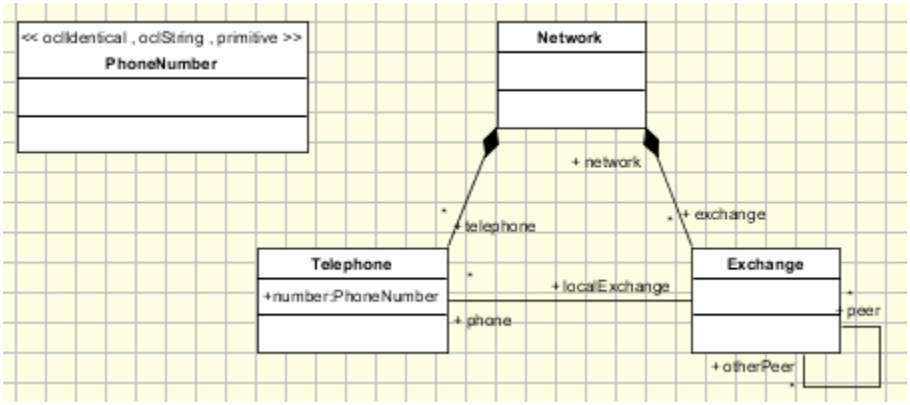
To ensure that your model converts correctly you should follow these rules:

1. Don't define any operations on classes, just attributes and associations.
2. If Poseidon creates a java package at any stage, delete it. Check that none of your attributes have undefined types or Java types.
3. Make sure that you explicitly name every association end in your model. These names become the attribute names in the MOF model. Association ends are not named by default, so you will need to set them explicitly by selecting them and completing the name field.
4. Make sure all attribute and association names are unique within the scope of a class. This includes all inherited attribute and association names.
5. Posiedon will create Java types for use as primitives in your model (such as int, String etc.) Do not use these as the types of your attributes. Instead, create a new class for each primitive type you wish to use. Call them whatever you like, but stereotype them with the following stereotypes: <<primitive>>, <<identical>> and one of <<oclBoolean>>, <<oclReal>>, <<oclInteger>> or <<oclString>> depending on what type of primitive you wish the attribute to be. Each type you define like this will therefore have 3 stereotypes attached to it.
6. If you wish you may create enumeration types by stereotyping a class with <<enumeration>>. The attribute names of such a class convert into the enumeration literals. The attributes may be typed in any way, this will not transfer to the model.

Use the UML editor in the EMOF/OCL application to convert your model into the concrete syntax of the MOF. The editor will report errors if it doesn't like your model.

Example

The following is a simple MOF/OCL model of a telephone network. The classes look like this:



Converted into the concrete syntax with invariants added:

```
specification NetworkModel {

  primitive PhoneNumber OCL_STRING IDENTICAL

  primitive Boolean OCL_BOOLEAN IDENTICAL

  primitive Integer OCL_INTEGER IDENTICAL

  class Telephone {

    number : PhoneNumber
    localExchange : Exchange opposite phone
    network : Network opposite telephone

    connectedTo(t : Telephone) : Boolean = {

      localExchange.connectedTo(t)
    }
  }

  class Network {

    component exchange : Exchange[*] opposite network
    component telephone : Telephone[*] opposite network

    invariant {

      telephone->forAll(t |

        -- All no phone number in the network should be the prefix
        -- of any other.
        Sequence(Integer) { 0 .. t.number.size() }->forAll(l : Integer |

          not telephone->exists(p : Telephone |

            t.number = p.number.substring(0, l)
          )
        )
        and
        -- All phones in the network must be able to reach all other
        -- phones
        telephone->forAll(p : Telephone |

          t.connectedTo(p)
        )
      )
    }
  }
```

```
  class Exchange {

    peer : Exchange[*] unique opposite otherPeer
    otherPeer : Exchange[*] unique opposite peer
    phone : Telephone[*] opposite localExchange
    network : Network opposite exchange

    connectedTo(t : Telephone) : Boolean = {

      phone->exists(p | p = t)
      or
      peer->exists(connectedNotVia(t, Set(Exchange) { self }))
    }

    connectedNotVia(t : Telephone, notVia : Exchange[*] unique) : Boolean = {

      phone->exists(p | p = t)
      or
      (peer - notVia)->exists(connectedNotVia(t, notVia->including(self)))
    }

    invariant {

      -- peer/otherPeer relationship is symmetric
      peer->forAll(p | otherPeer->exists(o | o = p))
      and
      otherPeer->forAll(o | peer->exists(p | o = p))
    }
  }
}
```

Note the addition of the primitive types, and the fact that the type PhoneNumber is equivalent to the OCL string type.