NEPTUNE II User Manual
Version 2.0.1
10 February 2013

Table of contents

Environment variables

Before installing NEPTUNE, you must:

☑ Check for the existence of the JAVA_HOME environment variable. If it does not exist, you must define it with a valid path to the current Java development kit. NEPTUNE requires at ***least version 1.6*** of the JDK in order to work correctly.

	NEPTUNE platform only works with the JDK 1.5 or JDK 1.6. JDK 1.7 is not currently supported.

	JRE is insufficient because metamodels are compiled and you need a Java version including Java compiler

☑ Add to your path an access to the Java and javac commands. To access these executables, you must add the variable "%JAVA_HOME%\bin" (windows form) or "$ {JAVA_HOME} /bin" (Unix form) to the path.

Examples:

- *In unix using sh shell, you must declare:*

```
JAVA_HOME=/usr/local/jdk1_6
export JAVA_HOME
path=$path:$JAVA_HOME/bin
export path
```

- *In windows XP, you must:*

```
Parameters->system->advanced->Environment variables
Create a new variable for a user:
      JAVA_HOME    "C:\Program Files\Java\jdk1.6.0_13"
Modify the path variable:
      Add at the end of this path   "…;%JAVA_HOME%\bin"
```

	We have tested tested NEPTUNE on Mac OS X Snow Leopard and Mac OS X Lion.The tests that we have carried out are partly. So if you find a bug, please send a mail to millan[at]irit[dot]fr

Installation

Installation on Windows environment

To install Neptune in Windows environment, execute the Neptune-x.y.exe file.

Installation on Linux and Mac environment

To install Neptune in Linux or Mac environment, execute in a terminal the command :

```
java -jar neptune-x.y.jar
```

neptune-x.y.jar is a self extracting file containing the NEPTUNE application

	If you install Neptune in /usr/local/Neptune or in /opt/Neptune, you must have superuser right access.

	After installation, do not forget to modify the right execution file for the .../Neptune/run.sh file.

Graphical User Interface Description

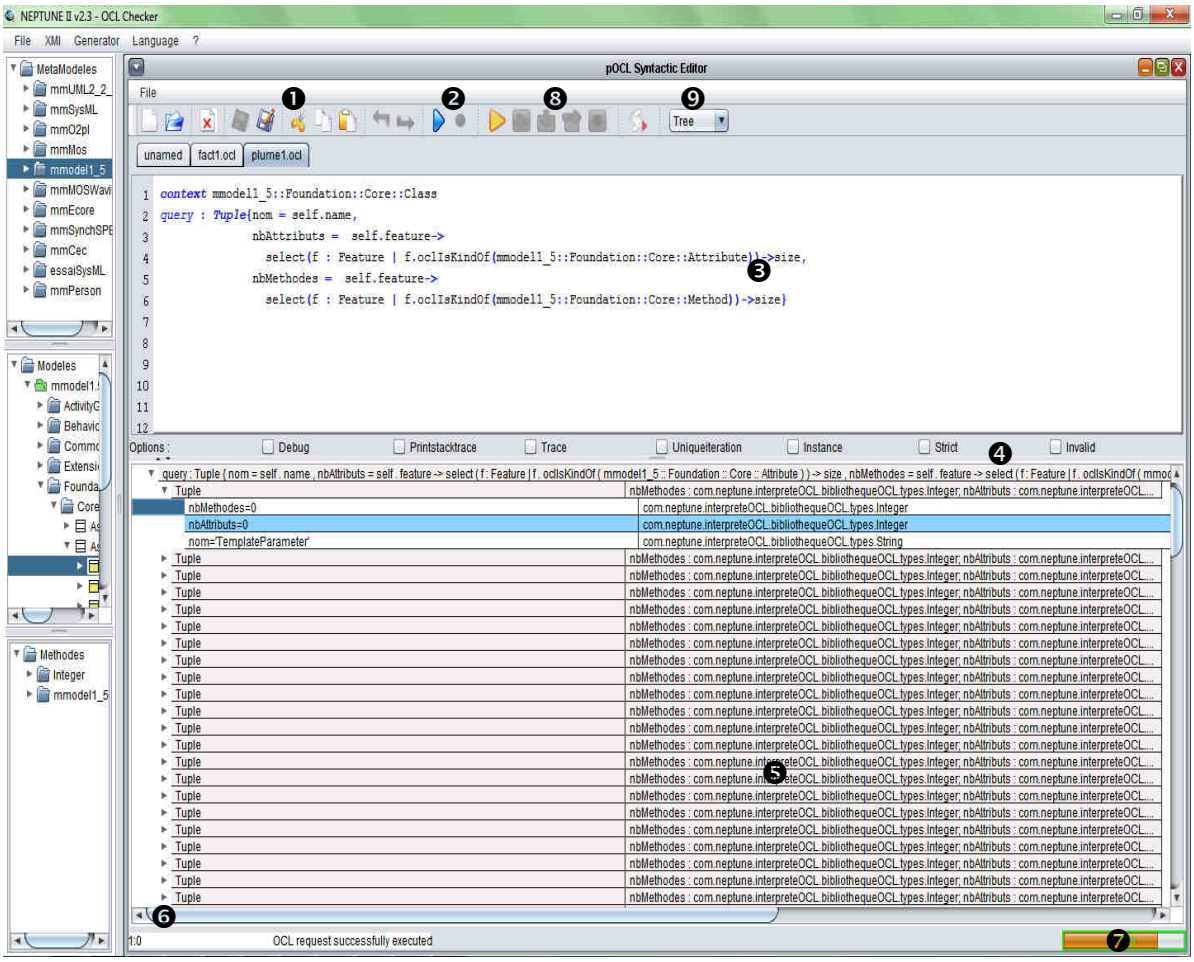


Figure 1: Neptune User Interface

As shown in Figure 1, the NEPTUNE GUI consists of two areas. The first area on the left of the window contains three browsers that allow navigating in metamodels, in models and in meta-classes' methods. The second area contains internal frames for editing and executing pOCL rules.

The browsers

Each browser uses icons showing the kind of elements handled.

Common icons for both metamodel and model browsers

	A meta-class.
Att	A meta-class or object's attribute.
→	An association between the meta-class and the specified meta-class.

Metamodel browser specific icons

 An inheritance link between the meta-class and the specified meta-class.

 An enumerative.

Model browser specific icons

 An object instance of the meta-class in which the object appears.

 An object part of an opposite association end of the considered object.

Model browser specific icons for debugging/simulation mode

 A parent node of the model three indicates that one of these children has a breakpoint. Selecting a parent node put a breakpoint on each of these children

 A parent node of the model three indicates that the OCL evaluator is stopped on one of these children

 A meta-class's instance (object) has a breakpoint

 The OCL evaluator is stopped on a meta-class's instance of the metamodel three

Method browser specific icons

 A method of the meta-class in which the method is defined.

	If no model is specified in an OCL rule, the model selected in the model browser is used.

The internal frames

Toolbars

❶ contains all the functionalities for creating, saving, editing pOCL rules.

 Create a new tab to edit and to process pOCL rules.

 Open a file containing a pOCL rule. Clicking on this button displays a dialog box where you select a rule.

 Close the active tab. If the rule currently edited is modified, a dialog box where you save the rule before closing the tab.

 Save the active tab. If it is the first save of the rule, a dialog box is displayed where you select a directory to save the rule and you provide the name of the file used to store the rule.

Save the active tab. Clicking on this button always opens the dialog box where you select the directory to save the rule and you provide the name of the file used to store the rule.

Cut the selected part of the rule and save it in the clipboard.

Copy the selected part of the rule and save it in the clipboard.

Paste the text in the clipboard into the editor at the current cursor position

Undo

Redo

❷ contains the button to process a rule and the button to save the result of this runtime on disc.

Process the rule currently edited

Stop the runtime of the rule currently executed

Save the result of the runtime in text file. A dialog box opens where you select the repository where to save the rule and you provide the name of the file used to store the result is displayed.

❸ contains the button to process a rule in debug or simulation mode and the buttons to continue and stop the execution (see ***Model debugging and animation***

	This functionality is not pertinent for huge models because the number of objects presents in a model may be very high.

Model debugging and).

Process the rule currently edited in debug/animation mode

Continue the execution of the rule stopped on a breakpoint

Stop the runtime of the rule currently executed in debug/simulation mode

❾ is a drop-down list for selecting the view of the request evaluation

Tree	The result of the request runtime is displayed as a tree where each node is a request and its first sub-node an instance of the metamodel where the request is evaluated. This is the default view. (Figure 1, ❺ - see ***Display the result of a request)***
Table	The result is displayed as a table. This display can be used only when the result of the request is a tuple (see ***Display the result of a request)***

Graphic	The result is displayed as a pie chart, or bar graph or dot char. As for the table display the result of the request must be a tuple. (See ***Display the result of a request)***

	For performance reason it is impossible to process several rules simultaneously. Indeed, processing several rules simultaneously requires using synchronous collections, which increase the access time significantly. For huge model, this increase is too prejudicial.

Tabs

A tab consists of four zones. The first one (❸) is the rule editor. This editor supports the syntactic coloration. OCL keywords are displayed in blue and bold italic style. pOCL specific keywords are displayed in blue and italic style. Strings are displayed in blue and comments in light gray. Four characters replace tabulation. The second area (❹) contains the six options of the pOCL interpreter.

Debug	Display in the console information about the nodes crossed by the interpreter. ***This option is only used by the expert to debug the interpreter.***
Printstacktrace	Display in the console the runtime stack when errors are generated by the interpreter. ***This option is only used by the expert to debug the interpreter.***
Trace	Display in the console information for debugging NEPTUNE interpreter. In particular, this option displays intermediate results. ***This option is only used by the expert to debug the interpreter.***
Uniqueiteration	This option indicates to the interpreter that the rule is only executed for the first instance processed. After this first iteration, the interpreter returns the result. This option can be used for example for rules beginning by "X.allInstances."
Instance	This option indicates to the interpreter that the rule is only executed for the selected instance. The instance is selected in the model browser. After this iteration, the interpreter returns the result.
Strict	This option imposes to the interpreter to verify that each expression is completely and correctly typed. In particular, this option forbids use of implicit variables in iterate expressions.

The third area (❺) displays the result of the processed rule. Each gray line displays the exact type of the rule processed on an instance. This gray line is a node containing the value calculated by the interpreter. If the result is a tuple, each leaf corresponds to an attribute of the tuple. If the result is a collection, each leaf corresponds to a value contained in the collection. The attributes or the values are displayed alternately in white and in blue.

The fourth area (❻) displays the current position of the cursor in the editor and the duration of the runtime once terminated.

❼ is common to all tabs. This progress bar indicates the amount of memory available for NEPTUNE. Clicking on it runs the garbage collector.

The menus

NEPTUNE menus

Four menus are available in the main menu bar of the NEPTUNE application (see Figure 2).

Figure 2: Neptune menubar

The first menu groups (Figure 3) the option to open the editor and consoles.

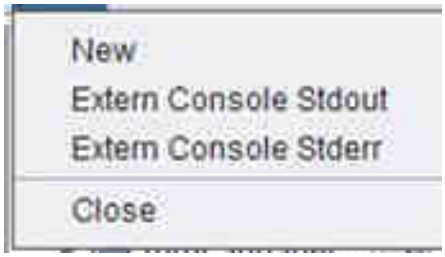


Figure 3: File menu

New	Open a new pOCL editor.
Extern Console Stdout	Open a console displaying the standard output messages.
Extern Console Stderr	Open a console displaying the error messages in red.
Close	Close the NEPTUNE tool. This action is similar to clicking on the top right cross.

The second menu (Figure 4) groups the options to handle models and metamodels.

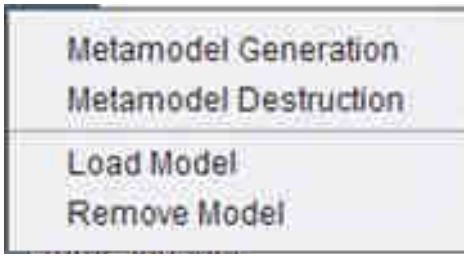


Figure 4: XMI menu

Metamodel Generation	(See section ***How to generate a metamodel?***)
Metamodel Deletion	(See section ***How to delete a metamodel?***)
Load Model	(See section ***How to load a model?***)
Remove Model	(See section ***How to remove a model?***)

The third menu (Figure 5) allows changing the current language used by the NEPTUNE interface. Currently NEPTUNE supports French and English. By default, the English language is selected.

Figure 5: Language menu

The fourth menu (Figure 6) contains additional functions as "*Help*" which displays help information (not yet implemented) and "*About*" which displays NEPTUNE contributors and NEPTUNE license.

Figure 6: ? menu

Editor menu

Figure 7: pOCL Syntactic Editor menu bar

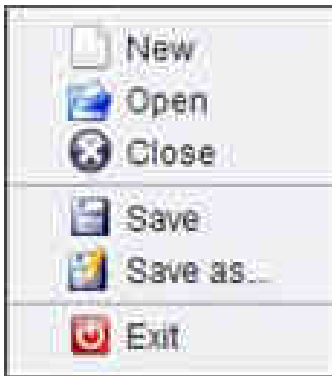


Figure 8: File pOCL editor menu

New	Create a new tab to edit and to process pOCL rules.
Open	Open a file containing a pOCL rule. Clicking on this button displays a dialog box where you select a rule.
Close	Close the active tab.
Save	Save the active tab. If it is the first save of the rule, a dialog box is displayed where you select a directory to save the rule and you provide the name of the file used to store the rule.
Save as	Save the active tab. Clicking on this button always opens the dialog box where you select the directory to save the rule and you provide the name of the file used to store the rule.
Exit	Close the current pOCL syntactic editor.

Contextual menus

Metamodel browser menu

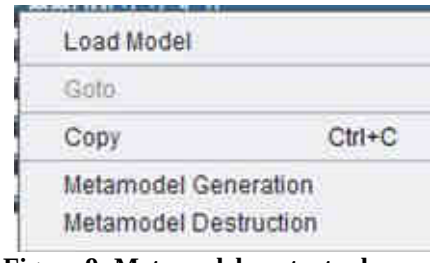
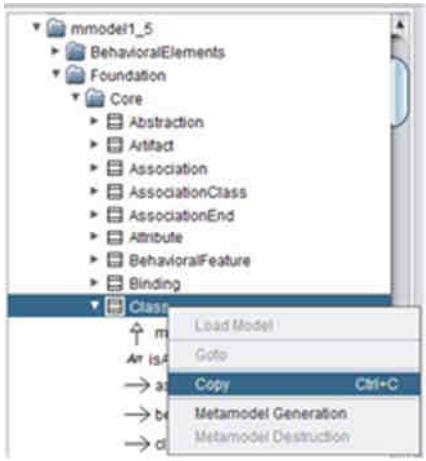


Figure 9: Metamodel contextual menu

Load Model	(See section ***How to load a model?***)
Goto	Goto allows following either an association end or an inheritance link. Click on "***Goto***" to access the description of the class in association with the selected class, or to the ancestor of the class if the considered link is an inheritance link.
Copy	Copy the path of the selected element in the browser.



Example 1: Example of the copy option

Example 1 shows an example of how to use the copy option. In this case the information in the clipboard is "*mmodel1_5::Foundation::Core::Class*".

Metamodel Generation	(See section ***How to generate a metamodel?***)
Metamodel Deletion	(See section ***How to delete a metamodel?***)

Model browser menu

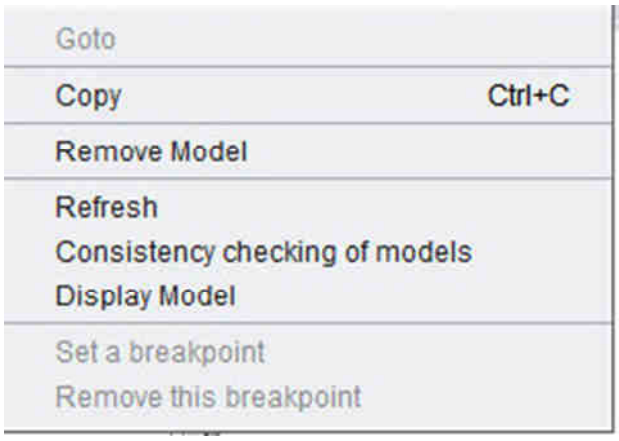


Figure 10: Model contextual menu

Goto	Goto allows following an association. Click on "***Goto***" to access the object in association with the selected object.
Remove model	(See section ***How to remove a model?***)

Copy	Copy the path of the selected element in the browser.
Refresh	Refresh allows displaying the models not displayed in the model browser. This is useful when a user creates a new model during a transformation.
Consistency checking of models	(See section ***Check of the conformity of a model)***
Display Model	(See section ***How to display a model***)
Set breakpoint	Tag a meta-class or an instance in order to indicate to the OCL interpreter that it will stop the rule runtime each time it access through an operation to an instance of the tagged meta-classes or to the tagged instances.
Remove this breakpoint	Remove the breakpoint set on a meta-class or on an instance

Menu Editor

Figure 11: pOCL editor context menu

Select all	Select all the text in the pOCL editor.
Cut	Cut the selected part of the rule and save it in the clipboard.
Copy	Copy the selected part of the rule and save it in the clipboard.
Paste	Paste the text in the clipboard into the editor at the current cursor position.

Display the result of a request

Tree visualization

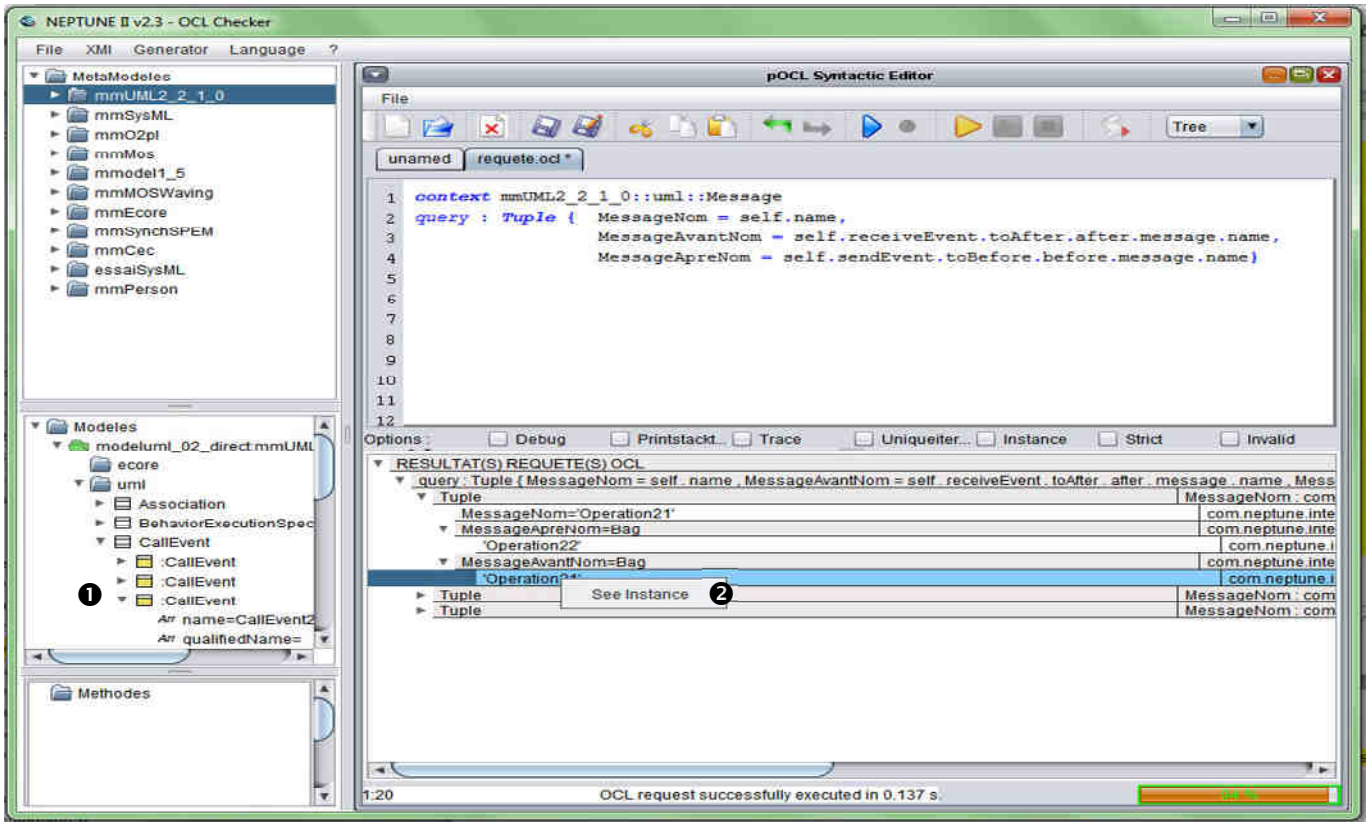


Figure 12: display of the results using the tree representation

This mode is the default mode for displaying the results of a request. It is possible, using the contextual menu and the "***see instance***" item (see Figure 12, ❷), to reach from a specific result the instance of the metamodel corresponding to this result (i.e. the instance on which the OCL rule was evaluated.

Table visualization

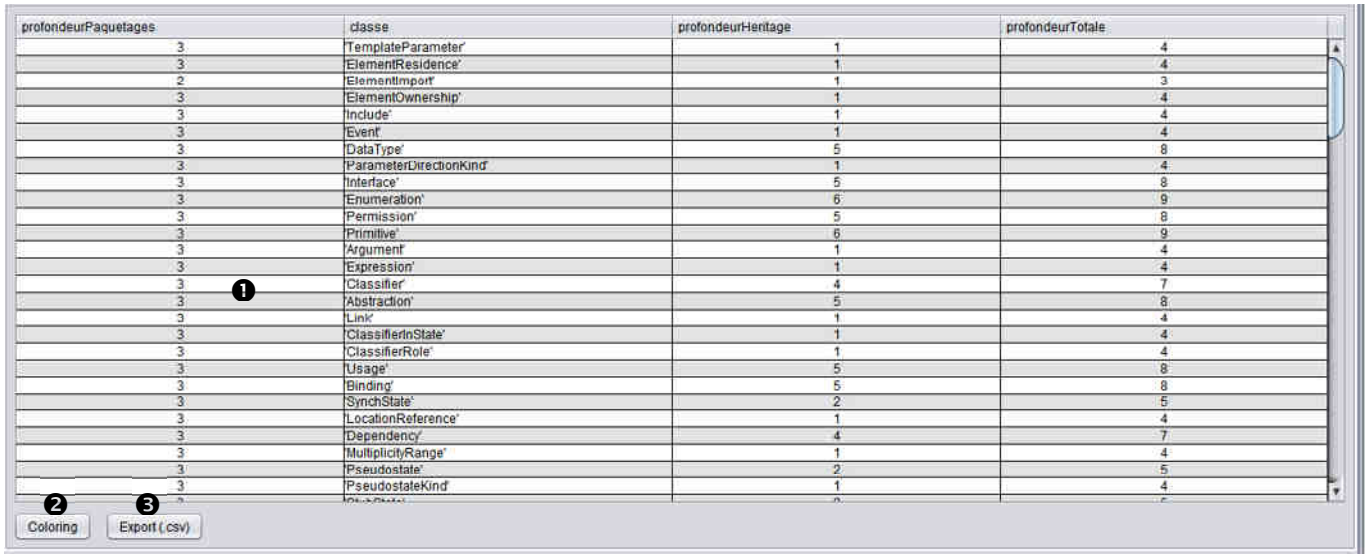


Figure 13: display of the results using the table representation

This visualization allows displaying the results of a request using a table. However, visualizing as a table requires that the OCL request uses the pOCL keyword "***query***" and that this query returns a tuple (Example 2). Each column contains the values of an attribute of the tuple constituting the result of the

request. The column's name is the same one as the attribute's name. This representation of the results is very useful for representing metrics for example. In addition, this representation offers tree functionalities for analyzing the results.

- The values present in the table can be ordered by clicking on the name of the column: One click for having an ascending order and another click for descending ordering. For strings this ordering uses the alphabet ordering;
- The values may be colored taking into account threshold chosen by the user. The button (Figure 13, ❷) opens the dialog box presented Figure 14. constituted of five elements. The drop-down list (❶) contains the attribute name of the tuple. The input-fields (❷ and ❸) allow providing the minimum and maximum thresholds. Check-boxes (() allow the selection of what the user wants to color; the values less than the minimum threshold, the values between the thresholds and the values greater than the maximum threshold. The buttons (❺) indicate the colors used for coloring the different rows of the table according to the threshold. Clicking on a button opens a dialogue box allowing the modification of the color corresponding to a threshold;

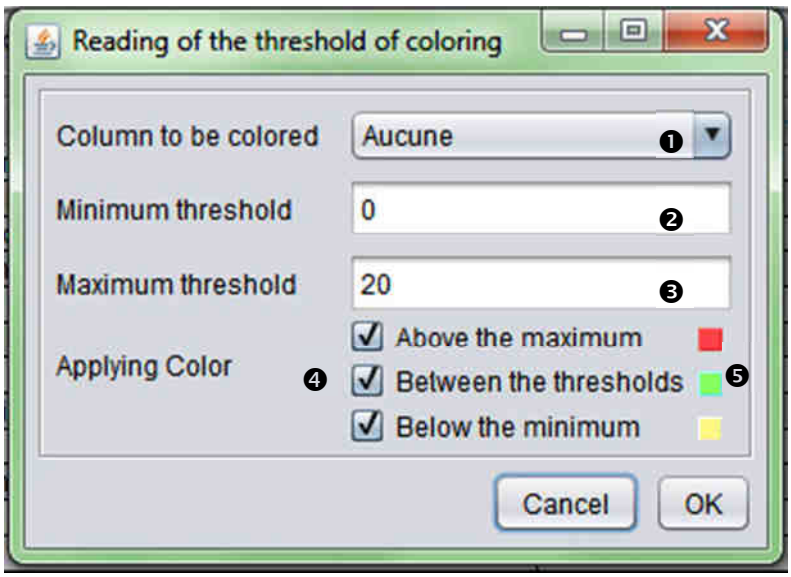


Figure 14: color dialogue box

Example 3 shows the result of an evaluation using a collared table.

- ❸ allows exporting a table as a cvs file. This button opens a dialog box for selecting the name of the file where typing the name of the file.

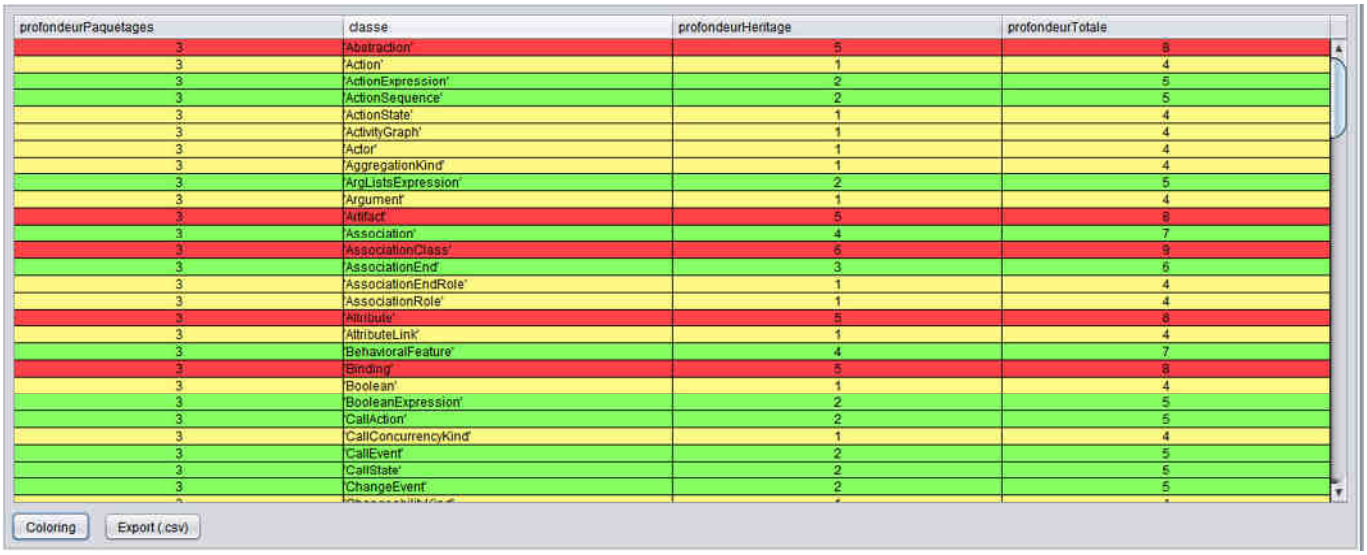
```
context Foundation::Core::Class

def: profondeur(ens : Bag(GeneralizableElement), n : Integer, espaceNom : String) :
                                    Integer =
if (ens->isEmpty) then n
else
        self.profondeur(ens.parent->flatten->select(
            ge : GeneralizableElement |  ge.namespace.name=espaceNom), n+1,
                espaceNom)
endif

def: nbPaqetage(ensP : ModelElement, n : Integer) : Integer =
       if (ensP.namespace->isEmpty) then n
       else
                    self.nbPaqetage(ensP.namespace, n+1)
       endif
```

```
query Exemple: Tuple{
      classe = self.name,
      profondeurPaquetages = self.nbPaqetage(self, 0),
      profondeurHeritage = self.profondeur(Bag{self}, 0, self.namespace.name),
      profondeurTotale = (self.profondeur (Bag{self}, 0, self.namespace.name) +
                                          self.nbPaqetage(self, 0))
}
```

Example 2: query returning a result that can be displayed as a table



Example 3: colored table

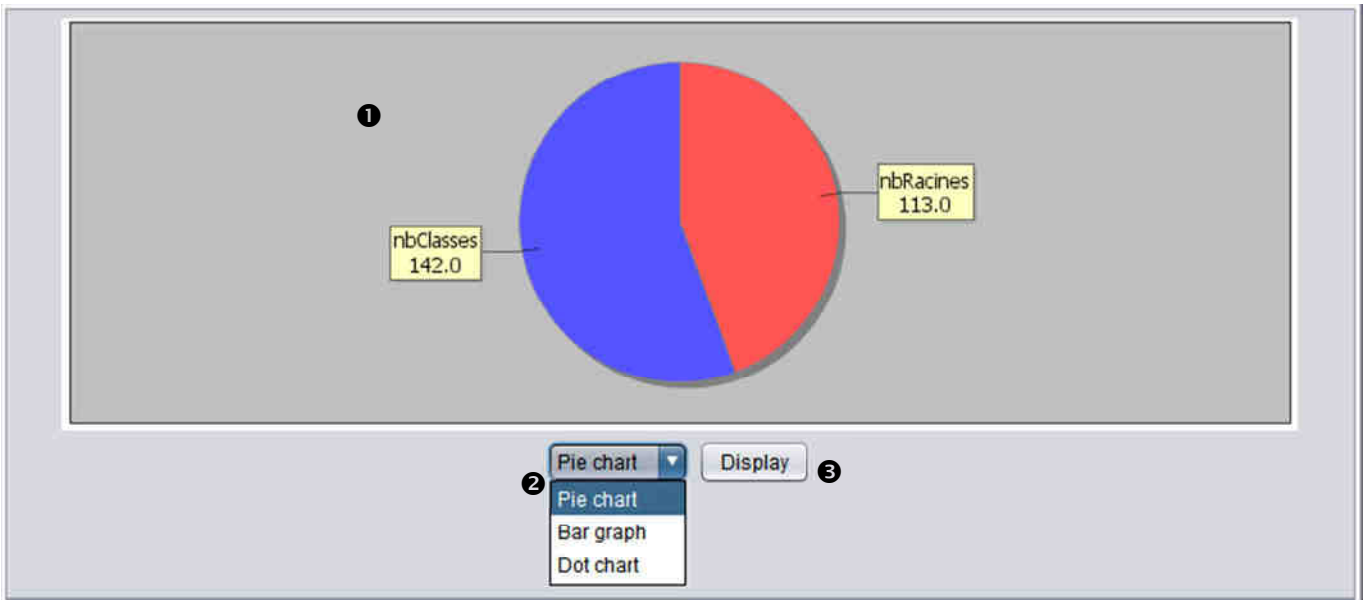
Graphic visualization

The last visualization is a graph displaying using pie charts, bar graphs or dot chars. As for the table visualization, visualizing as a graphic requires that the OCL request uses the pOCL keyword "***query***" and that this query returns a tuple. Generally, using this kind of visualization requires using the OCL interpreter option "***uniqueiteration***" (see Figure 1, ❹).

```
context mmodel1_5::Foundation::Core::Class
def : calculClasse(classes : Set(mmodel1_5::Foundation::Core::Class)) :
    TupleType(nbRacines : Integer, nbClasses :Integer) =
      Tuple {
        nbRacines = classes->select(cla : Class | cla.generalization->notEmpty)
                  ->size,
        nbClasses = classes->size}
query : calculClasse(mmodel1_5::Foundation::Core::Class::allInstances)
```

Example 4: OCL request for graphically displaying

Example 4 presents a request calculating the number of classes of a model and the number of classes that have at least an ancestor. The result is displayed in Example 3.



Example 5: graphic representation

❷ allows changing the graphic representation of the result ; the different possible views are pie charts, bar graphs and dot charts. Clicking ❸ changes the representation taking into account the representation selected in the drop-down list.

How to load a model?

There are two options to load a model.

☑ The first one is to select in the metamodel browser (see Figure 15, ❶) the metamodel which the model conforms to and to click on the ***'Load Model'*** option. A dialog box opens (see Figure 16) where the "***Name of Metamodel***" drop-down list is deactivated. Click on the button "…" (see Figure 16, ❹) to open the File chooser dialog box and select the XMI file containing your model. You can modify the model name in the text field "***Name of the model***".

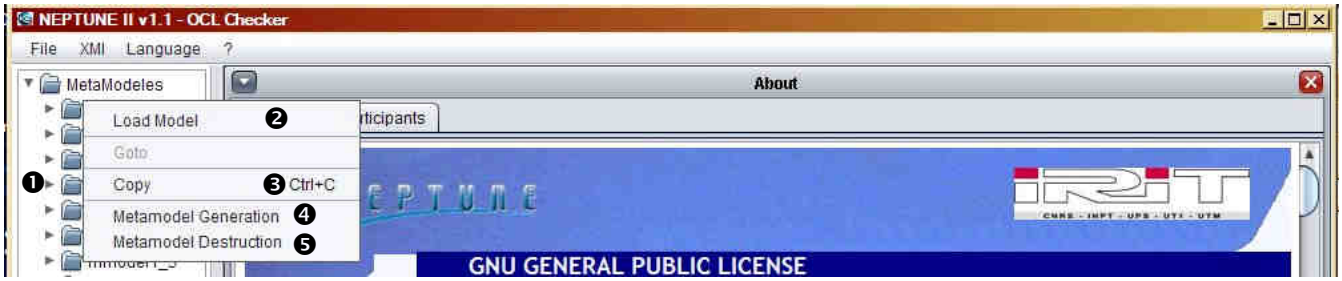


Figure 15: Metamodel browser and its menu

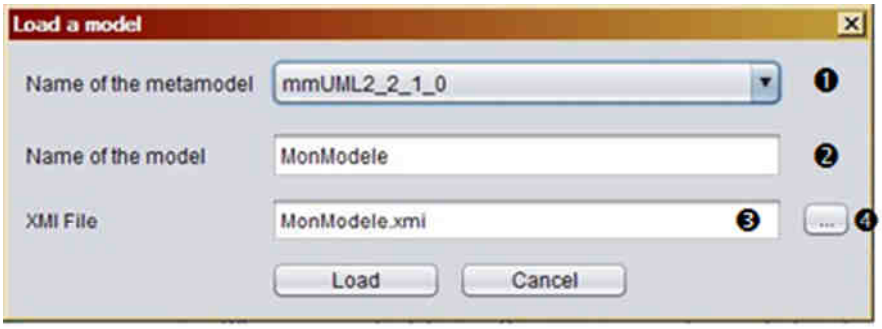


Figure 16: Load a model window

☑ The second one consists in selecting the "***load Model***" option in the XMI menu (see Figure 17, ❸). This action opens the "***Load a model window***" presented in Figure 16. In this case, the "***Name of Metamodel***" drop-down list is activated. You must select the metamodel corresponding to your model and click on the button "…" (see Figure 16, ❹) to open the file chooser dialog box then select the XMI file containing your model. You can modify the model name in the text field "***Name of the model***".

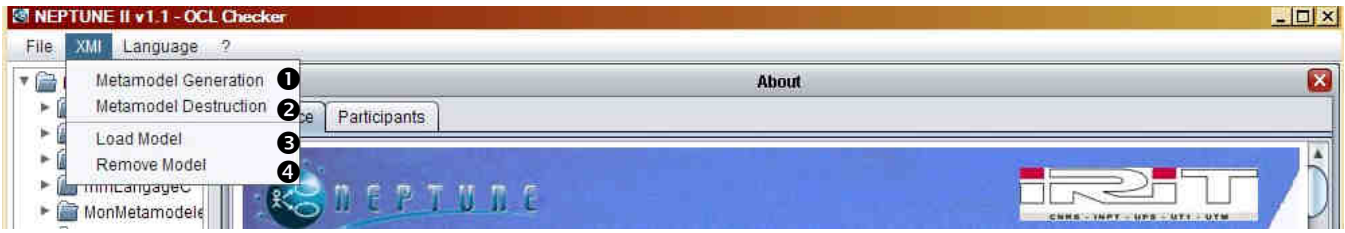


Figure 17: XMI menu

When a model is loaded, it appears in the model browser (Figure 12, ❶).

	The default extension for model files is ".xmi". However, some tools as TOCASED provides as file extension the metamodel name. In this case, you must choice in the dialog box for selecting a model file, the "all file" file type.

How to remove a model?

There are two options to remove a model.

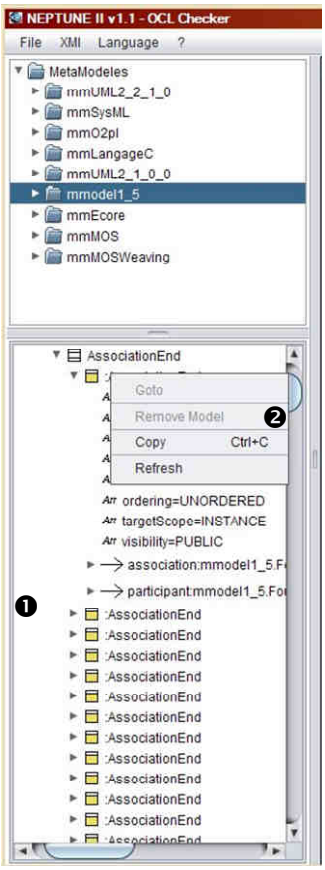


Figure 18: Model browser and its menu

☑ The first one consists to select in the model browser (see Figure 18, ❶) the model you want to remove and to click on the "***Remove Model***" option (see Figure 18, ❷). This opens the delete windows where the two text fields are deactivated. Clicking on this button opens a confirmation box where you confirm the removal. After selecting "yes", the model is removed.

☑ The second one consists in selecting the "***Remove Model***" option in the XMI menu (see Figure 17, ❹). This opens the remove model window displayed in Figure 19. In this case, the "***Name of the Metamodel***" and "***Name of the model***" drop-down lists are activated. You must first select the metamodel corresponding to your model then select the model in the list "***Name of the model***". Finally, click on "***Delete***" to complete the deletion.

Figure 19: Remove model window

How to generate a metamodel?

Generating a metamodel is a very delicate operation. Indeed, it requires having the metamodel description in XMI 1.5 or in Ecore format. In addition, the metamodel must be validated in order to be well formed. If the metamodel is created using the Eclipse platform, you must use the "Validate" option as illustrated in Figure 20 (❶) before generating a metamodel.

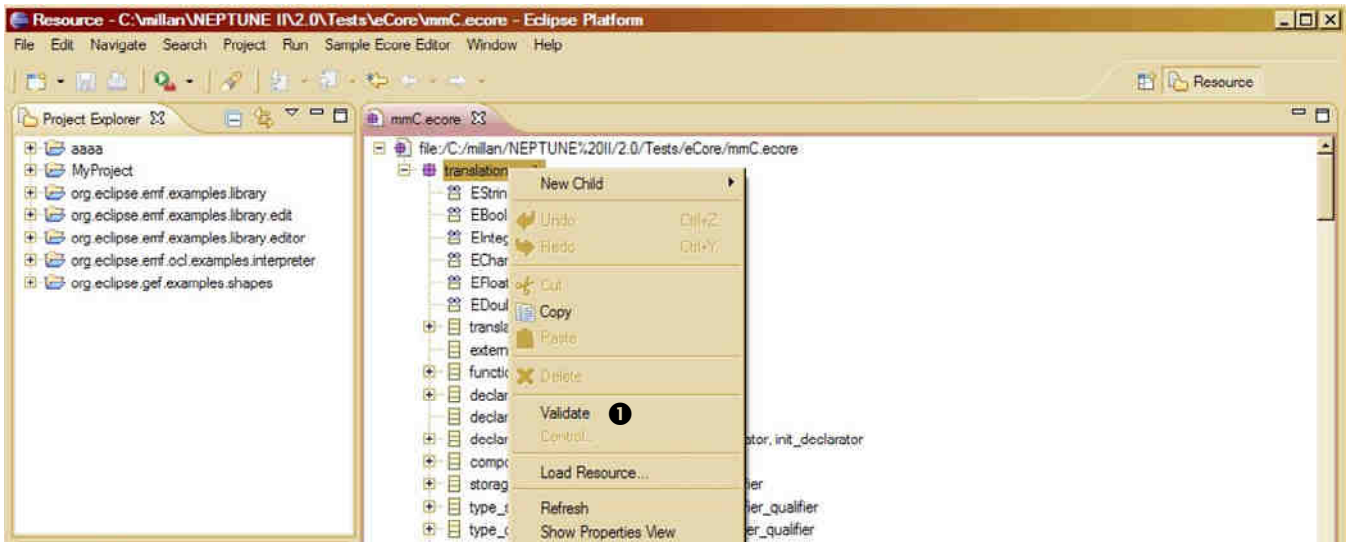


Figure 20: Eclipse menu to validate Ecore metamodels

There are two options to generate a metamodel.

☑ The first one consists to select in the metamodel browser (see Figure 15, ❶) the option "***Metamodel Generation***" (see Figure 15, ❹). This opens a dialog box (see Figure 21). Click on the button (see Figure 21, ❸) to open the File chooser dialog box and select the XMI file containing your metamodel. You can modify the metamodel name in the text field "***Metamodel name***".

Figure 21: Metamodel generation window

☑ The second one consists in selecting the "***Metamodel Generation***" option in the XMI menu (see Figure 4, ❶). This opens the load metamodel generation window displayed in Figure 21. You must click on the button "…" (see Figure 21, ❸) to open the file chooser dialog box and select the XMI file containing your metamodel. You can modify the metamodel name in the text field "***Metamodel name***".

NEPTUNE also performs a partial validation before generating a metamodel. Figure 22 shows the different steps of a metamodel validation. If no error is detected, the internal form of the metamodel is generated and compiled.

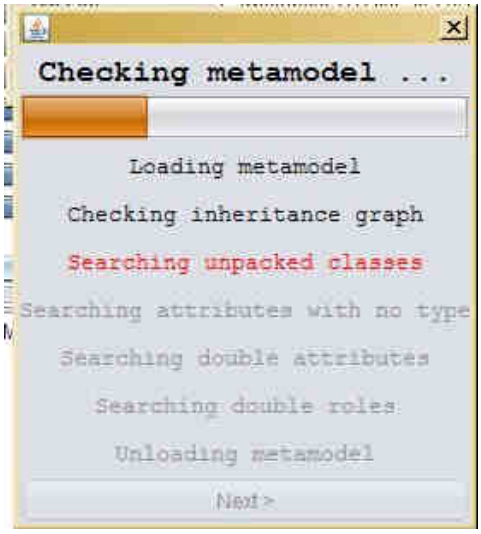


Figure 22: NEPTUNE metamodel validation

	"***Searching unpacked classes***" is only a warning. When metamodels reference other metamodels, some classes are not in a namespace because this namespace is part of another metamodel. If a warning or an error occurs, the box below (Figure 23) is displayed. This box displayed the problem type (error or warning), a description of the problem (❶) and the result of the test (❷) to find the element for which the test failed. "Save" allows saving the result on disk and "ok" allows continuing the validation.

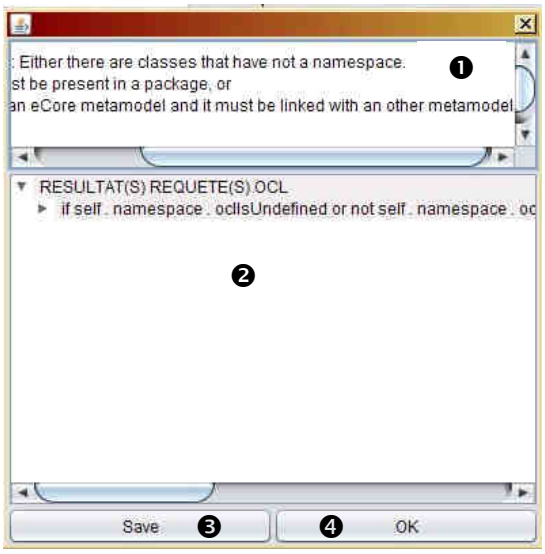


Figure 23: NEPTUNE metamodel error or warning box

	In order to avoid conflicts between the metamodel name and a namespace available in the metamodel, we recommend prefixing the metamodel name by "mm". For example, the name of the C language metamodel could be "mmLangageC".

	The NEPTUNE metamodel validation tool is not able to detect all the errors in a metamodel. So the compiling of the metamodel may can fail. It is important to carry out the results of the compilation and to correct the metamodel. **In this case, the partially compiled metamodel must be manually removed.**

Ecore metamodels can have references to meta-elements from other metamodels. In this case, the box below (Figure 24) is displayed.

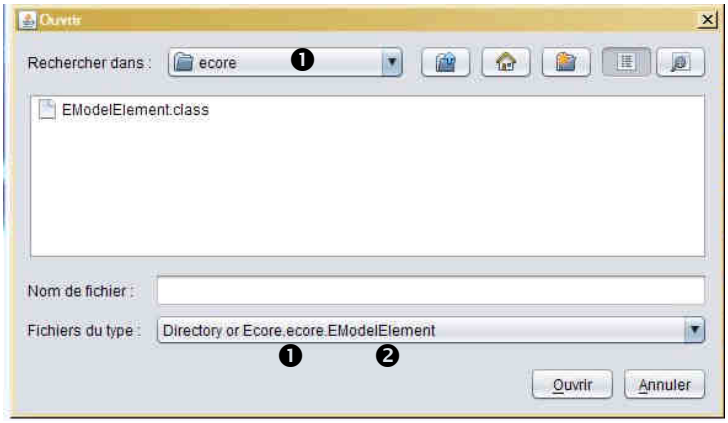


Figure 24: Meta-classes selector

This dialog box allows users to select the different elements required for generating a new metamodel. ❶ Indicates the metamodel where the required meta-class is located and ❷ indicates the meta-class. If different versions of a same metamodel co-exist in NEPTUNE, the user must select the correct one.

	We recommend to create your metamodels with software TOPCASED (http://www.topcased.org/). However, if you do not have TOPCASED, it is possible to use the ArgoUML 0.22 case tool (http://argouml-downloads.tigris.org/argouml-0.22/).

How to delete a metamodel?

There are two options to delete a metamodel.

☑ The first one consists in selecting in the metamodel browser (see Figure 15, ❶) the metamodel you want to delete. This opens the "***Metamodel Destruction***" window where the text field is deactivated. Clicking on the "***Destruction***" button opens a dialog box asking for the confirmation of the destruction. After selecting "yes", the metamodel is deleted.

☑ The second one consists in selecting the "***Metamodel Destruction***" option in the XMI menu (see Figure 4, ❷). This opens the "***Metamodel Destruction***" window displayed in Figure 25. In this case, the "Name of the Metamodel" drop-down list is activated. Select the metamodel you want to delete and click on the "***Destruction***" button.

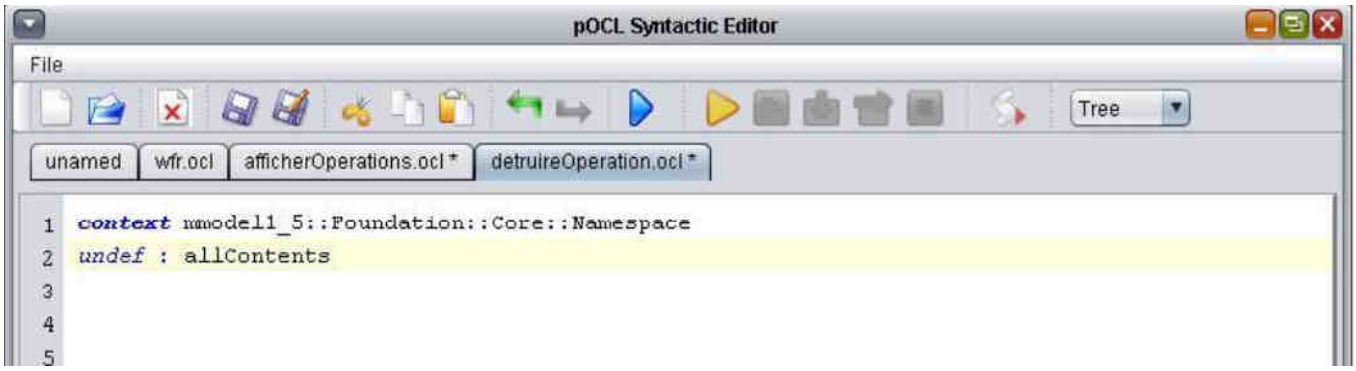
Figure 25: Delete metamodel window

	In order to avoid problems, it is recommended to exit from NEPTUNE and to start it again.

How to show and delete metaclasses' methods

There are two options to delete meta-class's methods.

☑ The first one consists in writing the request for deleting methods in the request editor (see Example 6). The deletion of a method updates the methods' browser.



Example 6: pOCL rule for deleting method

The template of the request to delete a method is as follow:

```
context name of the meta-class
undef : operation to be destroyed
```

☑ The second one consists in using the methods' browser to select the item (***delete***) in the contextual menu accessible by clicking on the method in the contextual menu (see Figure 26).

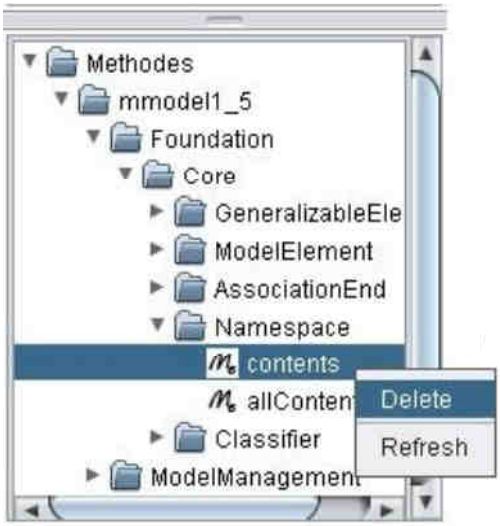


Figure 26: Methods Browser and its menu

Check of the conformity of a model

To check the conformity of a model, make a right click on the name of the concerned model and chose "*Consistency checking of models*" (see Figure 27, ❶).

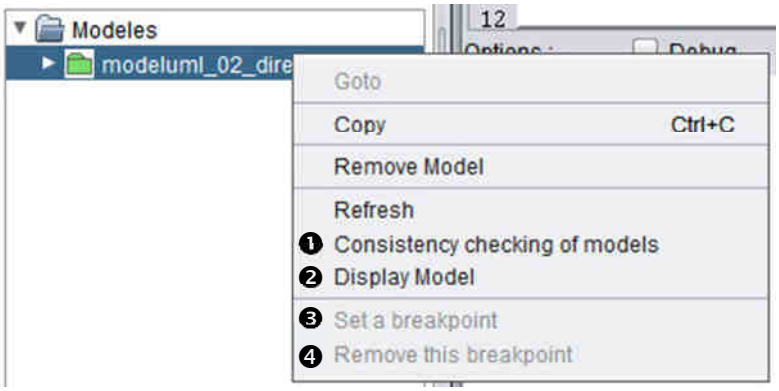
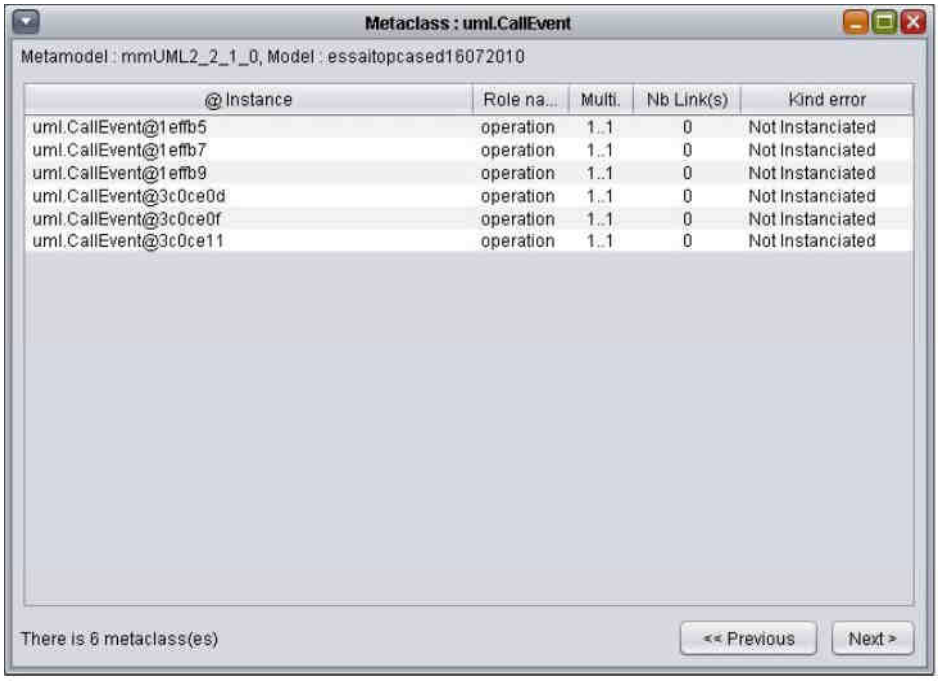


Figure 27: item "Consistency checking of models"

After that, a window opens (Example 7) and we can visualize the following board:



Example 7: display of meta-class inconstancies

In the first column, we can see the meta-class in error as well as the role of this meta-class with the expected multiplicities and the number of links in this model. Each page contains the error of only one meta-class. The buttons "***Previous***" and "***Next***" allows navigating between meta-classes having errors.

	This functionality is under development; only roles' multiplicity is checked.

How to display a model

NEPTUNE offers a functionality for displaying a model as a UML object diagram. However, this functionality is usable for model having a moderate size. For huge model, too many objects are displayed and the object diagram is unreadable. To display a model, make a right click on the name of the concerned model and chose "*display model*" (see Figure 27, ❷).

Figure 28 presents the window in which the model is displayed. Two zones constitute this window. The first one (❶) is the visualization of the model where each object can be selected and moved in order to organize the graph. The second zone provides two tabs : one for filtering the meta-elements you want to display (❷) and one to display the properties of the selected object in the visualization (❸).

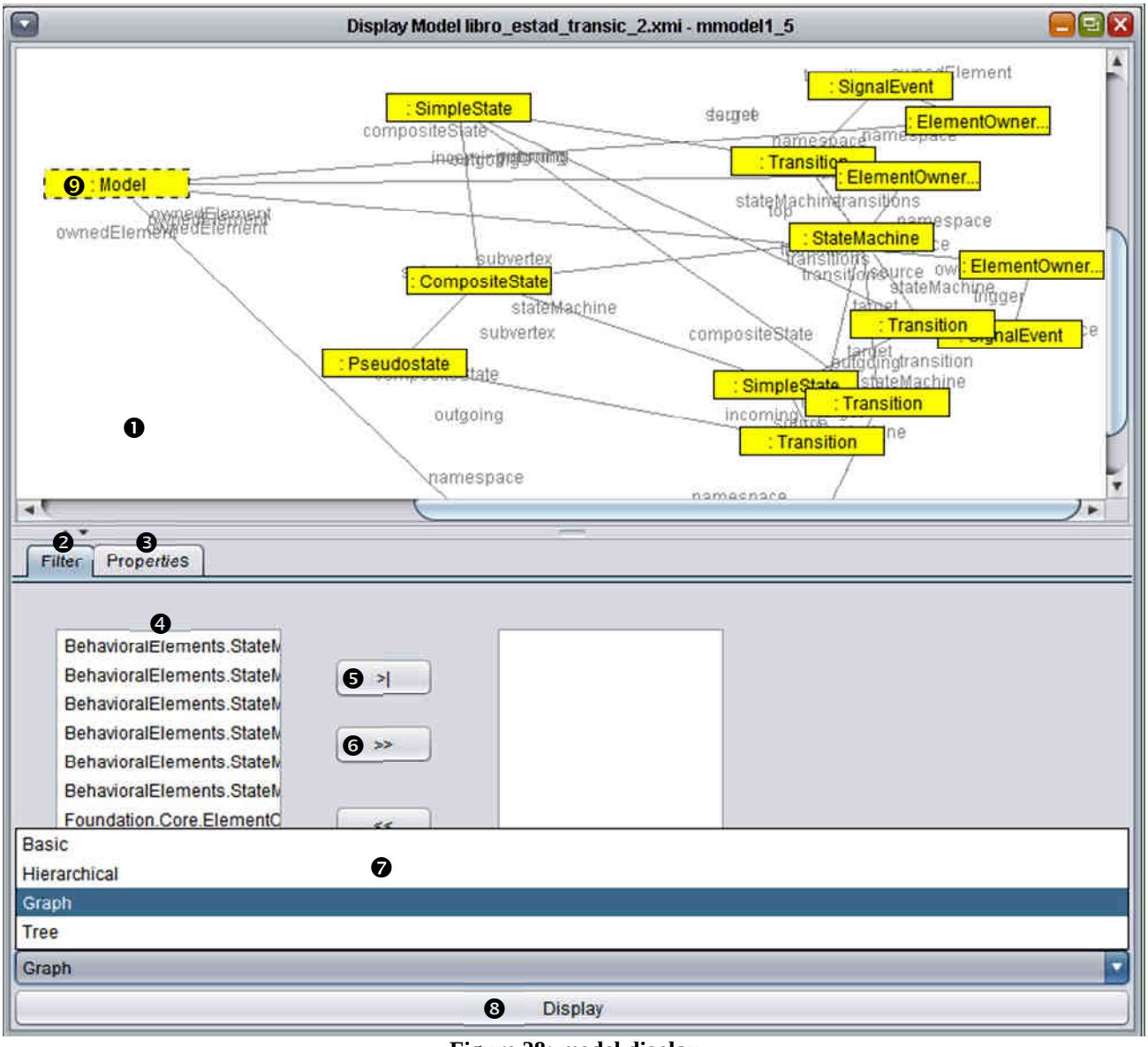


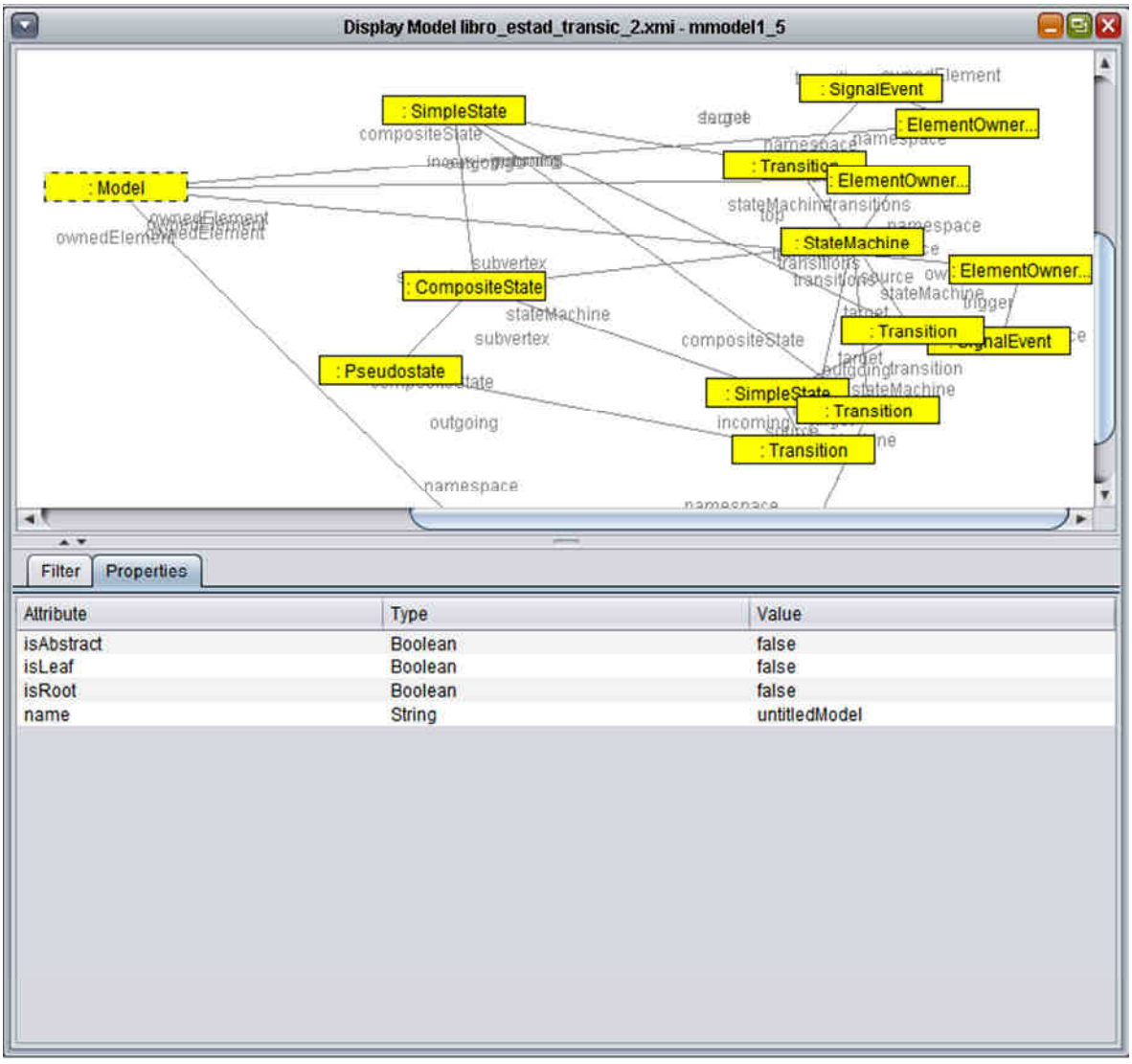
Figure 28: model display

At the beginning you must select the meta-classes of the instances you want to visualize in the list ❹. Click on the button ">>" (❻) in order to validate your choices. These choices appear in the right list. You can deselect meta-classes by selecting them in the right list and clicking on the button "<<". ">|" permits to select all the meta-classes in the left list (❹) and "|<" permits to unselect all the meta-classes present in the right list. After having chosen the meta-classes, you must choose the algorithm for organizing the visualization of the diagram. There are four algorithms implemented to automatically organize the positioning of objects in the visualization (Basic, Hierarchical, Graph and Tree). By default,

the algorithm chosen is "*Graph*". You can choose another algorithm in the drop-down list ❼. Finally click on the button ❽ for starting the display. For more information concerning these algorithms see the Jgraph documentation (*https://github.com/jgraph/jgraphx/tree/master/docs*).

	If the number of meta-classes chosen is high the time for displaying a model may be important.

For each object displayed, it is possible to show the value of its attributes. To select an object, click on its representation on the visualization zone (Figure 28, ❶). The selected object is bordered in dotted line. These attributes and their values are visible after clicking on the tab "*Properties*" (see Figure 28, ❸). Example 8 shows the value of the attributes of an instance of the meta-class "*Model*".



Example 8: visualization of the attributes of a meta-class' instance

	This functionality is not pertinent for huge models because the number of objects presents in a model may be very high.

Model debugging and animation

NEPTUNE supports a special mode in order to debug rules. In our context the debugging mode allows to set breakpoint on different elements of a model in order to verify that the rule access to these elements. As an object can be presented in a collection it is necessary to define a function that return the considered objects. For example in Example 9 the function "*findFlowTarget*" is only defined for setting breakpoint on "*uml.ActivityNode*". For setting breakpoints, two solutions exist. The first one is to select an object in the model browser, make a right click and chose "*set a breakpoint* " (see Figure 27, ❸). The second one is to select a meta-class in the model browser, make a right click and chose "*set a breakpoint* " (see Figure 27, ❸). With this option, a breakpoint is set on all the instances of this meta-class. For removing breakpoints it is similar than setting them but choosing the item "*remove this breakpoint*" in the contextual menu.

```
package mmUML2_2_1_0
context mmUML2_2_1_0[breizhmodel_01]::uml::Model

def : findActivityName(pNom : String) : mmUML2_2_1_0::uml::Activity =
        uml::Activity::allInstances->select(
            activite : uml::Activity | activite.name = pNom )->asOrderedSet->first()

def : findInitialNode(pActivite : uml::Activity) :  uml::ActivityNode =
       pActivite.node->
            select(elem  :  uml::ActivityNode  |  elem.oclIsTypeOf(uml::InitialNode))->
                  asOrderedSet()->first()

def : findFlowTarget(pFlow : uml::ActivityEdge) : uml::ActivityNode =
        pFlow.target

def : findActivityEdgeOutgoing(pActivityNode01 : uml::ActivityNode) :
                  uml::ActivityEdge =
    pActivityNode01.outgoing->
      select(elem : uml::ActivityEdge | elem.oclIsTypeOf(uml::ControlFlow)  or
             elem.oclIsTypeOf(uml::ObjectFlow) )->asOrderedSet()->first()

def : nextActivitiNode(pActivityNode02 : uml::ActivityNode) : uml::ActivityNode =
         findFlowTarget(findActivityEdgeOutgoing(pActivityNode02))

def : nextActivitiNodeRecurs(pActivityNode02 : uml::ActivityNode) :
            uml::ActivityNode =
    if pActivityNode02.oclIsUndefined then
         null
    else
         nextActivitiNodeRecurs(nextActivitiNode(pActivityNode02))
    endif

query :
 self.nextActivitiNodeRecurs(findInitialNode(findActivityName('Activity1'))).name
endpackage
```

Example 9: debugging and animating rule runtime

There are three buttons for executing a rule in this mode:


Process the rule currently edited in debugging/animating mode


Continue the execution of the rule stopped on a breakpoint


Stop the runtime of the rule currently executed in debug/simulation mode

When the rule runtime stops on a breakpoint initially red it becomes green and the breakpoint on Its meta-class also becomes green.

	The next feature is an experimental feature depending of the metamodel used. Indeed, this functionality is only implemented for UML 2.1 and in particular for its activity diagram. It is a proof of concept showing that it is possible to animate UML models only using the OCL language.

When a rule runtime stops on a breakpoint set on a decision node, a list showing the different outgoing existing for this node. The user chooses the outgoing he wants to follow and click on the button  restart runtime following the selected outgoing.

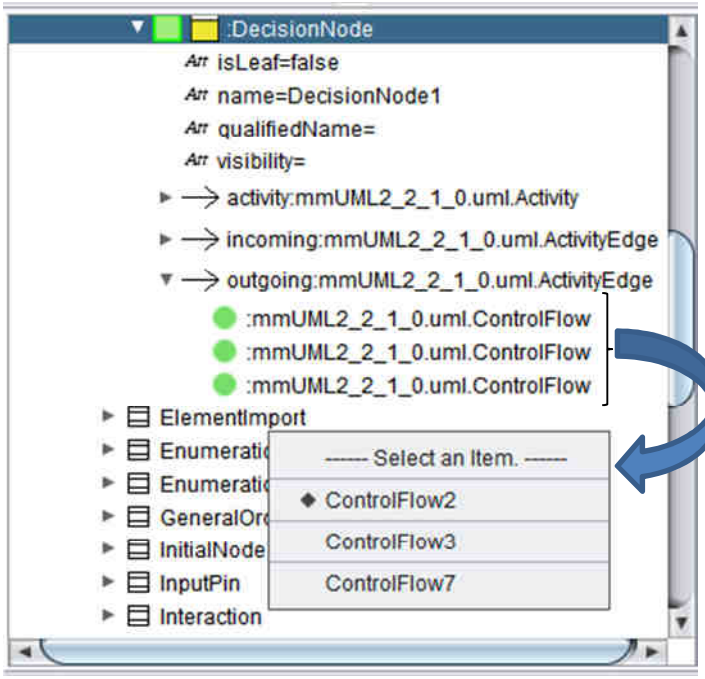


Figure 29: selecting an outgoing in a DecisionNode

Figure 29 shows the runtime of the rules of Example 9. In this figure, we note that the list named "*select an item*" contains the roles of the "*DecisionNode*".

	Some works are carried out in order to generalize this functionality to all metamodels having meta-classes supporting alternatives.

Ocl extension for model serialization

As NEPTUNE delivers tools to transform models, it is also necessary to provide functionalities for model serialization. In this version, two serializations are provided: text serialization and XMI serialization. To support serialization, the "OclVoid" and "Collection" have been enriched with new operations. There are required in order to serialize models in XMI format and two for serialization in Text format. pOcl extensions

Extension to write multimodel OCL rules

The main extension of OCL is the capability to handle simultaneously several models conform to different metamodels.

To indicate a metamodel in an OCL rule, you must prefix the path with the metamodel name.

```
package  mmO2pl::o2pl                                                    ❶

context mmO2pl::o2pl::Affectation_Contenu                                ❷
def : let instruction : Sequence(OclAny) =
      Sequence{self.affContSuiv, self.paramCont, self.valCont}

def : let registre : Bag(String) =
      mmMos[atv_bds1, atv_bds2]::mos::RootDefinition::allInstances.      ❸
            rechercheRegistre('ccdTemp')->flatten

def : estTerminal(cfs : Bag(mmSysML::uml::ControlFlow)) : Boolean =
      cfs.target->select(an : mmSysML::uml::ActivityNode |
         an.oclIsKindOf(mmSysML::uml::ActivityFinalNode))->size = cfs->size  ❹
```

Example 10: metamodel and model naming in an OCL rule

Example 10 shows three use cases of the metamodel name. In ❶, the metamodel name is used in a package definition in order to specify the metamodel where to find the package "*o2pl*" for example. In ❷, the metamodel's name is used in a context definition in order to specify the metamodel where to find the class for example "*o2pl::Affectation_Contenu*". In ❸ and ❹, the metamodel name is used in an expression definition for specifying the complete path to a specific class for example "*mmMos[atv_bds]::mos::RootDefinition*" or "*mmSysML::uml::ActivityFinalNode"*.

To specify models in OCL rule, you must add the model between square brackets separated by commas. When several models are specified, the rule is executed on all models. In ❸, "*allInstances*" is applied to all instances of the class "*mos::RootDefinition*" in the models "*atv_bds1*" and "*atv_bds2*".

Definition of functions on OCL datatypes

NEPTUNE supports definition of new operations on OCL datatypes. Example 11 illustrates the operation "*startsWith*" that returns true if "*self*" starts with the string "*s*".

```
context String
def: startsWith(s : String) : Boolean =
      s.size() <= self.size() and self.substring(1, s.size()) = s
```

Example 11: operation definition of OCL datatype

Syntax extension

A problem occurs when a model element has the same name as an OCL keyword.

```
<eClassifiers xsi:type="ecore:EClass" name="Comment" eSuperTypes="#//Element">
      …
      <eStructuralFeatures xsi:type="ecore:EAttribute" name="body" ordered="false"
            eType="#//String" unsettable="true">
```

```
      …
      </eStructuralFeatures>
</eClassifiers>
```

Example 12: XMI fragment of a metamodel

In Example 12, a fragment of a metamodel defines a classifier with an attribute called "*body*". "*body*" is an OCL keyword defining the operation's body.

```
context mmSysML[setstartrackertovacuumtemperaturecomp]::uml::DecisionNode
def : estTemperature(cfs : Bag(mmSysML::uml::ControlFlow),  cond : String) : Boolean
            = cfs.guard->select(c : mmSysML::uml::ValueSpecification |
                  c.oclIsKindOf(mmSysML::uml::OpaqueExpression)).
                        body->flatten.contains(cond)->includes(true)
```

Example 13: OCL rule using keyword as the model element name – version 1

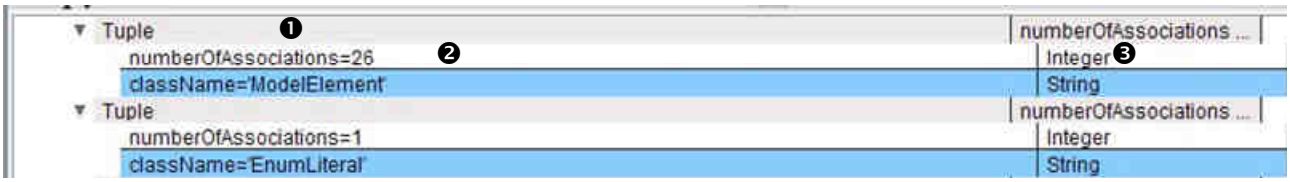
Example 13 is an OCL definition using "*body*". In this operation definition, the use of the meta-element "*body*" conflicts with the OCL keyword "*body*". In order to avoid such a conflict, meta-elements having the same name as an OCL keyword must be surrounded with underscores as in Example 14.

```
context mmSysML[setstartrackertovacuumtemperaturecomp]::uml::DecisionNode
def : estTemperature(cfs : Bag(mmSysML::uml::ControlFlow),  cond : String) : Boolean
            = cfs.guard->select(c : mmSysML::uml::ValueSpecification |
                  c.oclIsKindOf(mmSysML::uml::OpaqueExpression)).
                        _body_->flatten.contains(cond)->includes(true)
```

Example 14: OCL rule using keyword as the model element name – version 2

Three new OCL keywords have been added to NEPTUNE. In the pOCL editor, these keywords are in blue and in italic. "*query*" allows to define queries on a model. It is then possible to define views and metrics. All the results returned by a query have been correctly typed and this type is part of the returned result.

```
context mmodel1_5[umlmodel15]::Foundation::Core::Class
query : Tuple {   className : String = self.name,
                  numberOfAssociations : Integer = self.association->size}
```

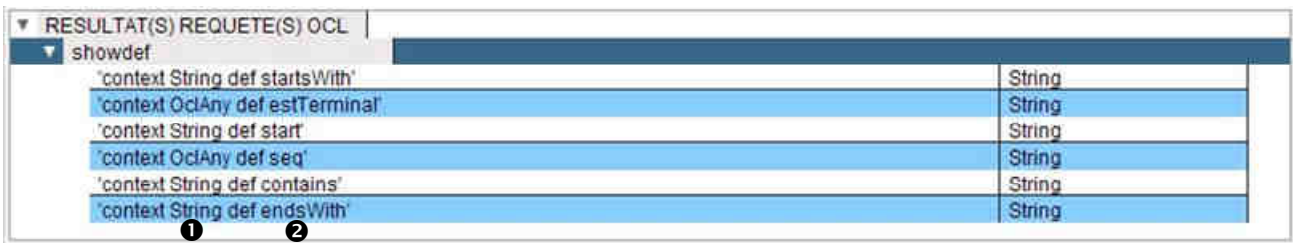


Example 15: pOCL rule using a query

Example 15 shows a rule allowing to extract the name of each class and the number of its associations. The result of this query contains an indication of the data structure used (❶), the elements composing this structure (❷) and the type of each element (❸).

"*showdef*" displays all the operations defined. The context is only required to have an OCL rule syntactically well formed. Example 16 shows the additional operation defined to write rules. An operation is defined in a context (❶) and has a name (❷).

```
context mmO2pl
showdef
```



Example 16: showdef command

An operation can be used on an element having the same context as the operation or on an element having a type compatible with the context on which the operation is defined. For example, an operation defined on real is usable on integer. Operations defined in NEPTUNE support overloading and dynamic binding.

"*undef*" removes an operation defined in a context.

```
context String
undef : startsWith
```

Example 17: undef command

Example 17 illustrates a rule to remove the "*startsWith*" defined on the context "*String*".

Library extensions

The OCL library is compliant with the OCL 2.0 standard. However, the NEPTUNE OCL library for primary types and collections includes all the operations specified in the "Object Constraint Language – OMG Available Specification – Version 2.1 (ptc/09-05-02)". In order to use NEPTUNE for different projects, we add several new operations on "String" and "Collection".

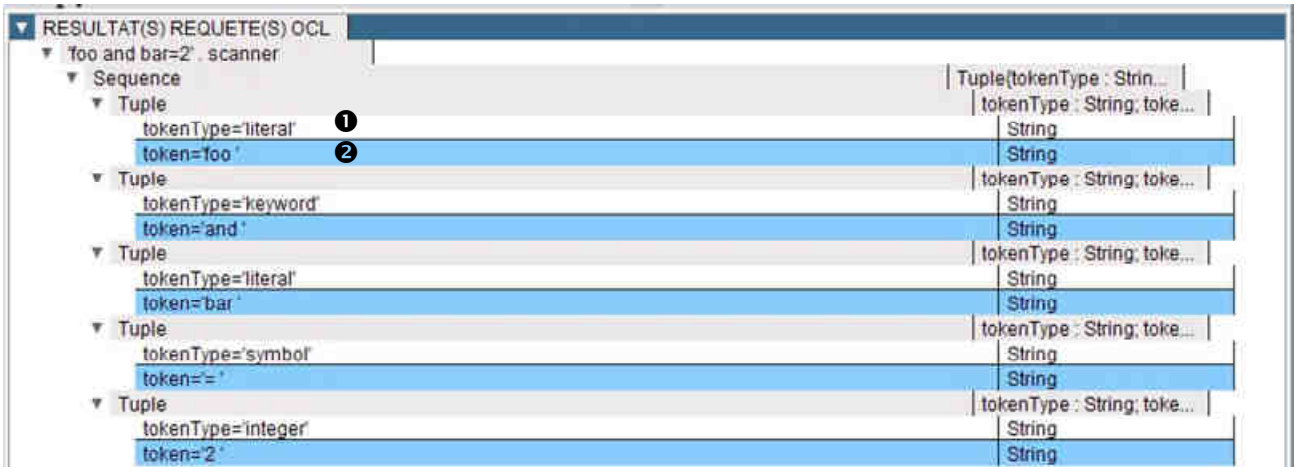
OclVoid

"oclUndefined" operator returns an oclUndefined object. This operation is mainly used in transformation in order to provide a result when an assignment or a creation failed (Example 21, ❺).

String

In order to analyze conditional expression used in decision nodes for example, the "*String*" library includes an operation called "*scanner*" returning the sequence of tokens present in the expression where the scanner operation is applied. This operation returns a sequence of tuple made of an attribute containing the token (see Example 18, ❷) and another containing the type of the token: "*keyword*", "*literal*", "*symbol*", "*integer*", "*boolean*", "*string*" and "*real*" (see Example 18, ❶).

```
'foo and bar=2'.scanner
```



Example 18: scanner operation

Collection

A model can be considered as an oriented graph. In this context, it could be useful to calculate the whole path between two nodes of a model. "*closure*" returns a collection resulting from the application of an OCL expression on the elements of a collection recursively until no new calculated element can be added. This operation implements a fixed point algorithm.

```
context mmodel1_5[umlmodel15]::Foundation::Core::Class
query : Tuple{nom = self.name, anc = self->closure(c :
      mmodel1_5::Foundation::Core::Class | c. c.generalization.parent).name}
```

Example 19: closure operation

Example 19 shows a rule allowing to complete all the ancestors of "*self*". The result of this computing includes "*self*".

Dynamic typing

The NEPTUNE pOcl interpreter supports dynamic typing of expressions. For each evaluated element the correct type is automatically computed. In this context, the OCL operator "*OclAsType*" is now useless because the dynamic type is always computed during an evaluation.

```
context mmodel1_5[umlmodel15]::Foundation::Core::Classifier                    ❶
query :  mmodel1_5::Foundation::Core::Classifier::allInstances

context mmodel1_5[umlmodel15]::Foundation::Core::Classifier
query :  mmodel1_5::Foundation::Core::Classifier::allInstances->               ❷
      select(cl | cl.oclIsKindOf(mmodel1_5::Foundation::Core::Class))

context mmodel1_5[umlmodel15]::Foundation::Core::Classifier
query :  mmodel1_5::Foundation::Core::Classifier::allInstances->               ❸
      select(cl | cl.oclIsKindOf(mmodel1_5::Foundation::Core::Class)).isActive
```

Example 20: dynamic typing

Example 20 shows three pOCL rules. Evaluation (❶) returns as a set of "*Classifier*" while evaluating (❷) returns a set of "*Class*". In this case, the evaluation of the rule shown in (❸) is valid despite the "*oclAsType*" operation is not used.

pOcl extension for model transformation

An OCL extension allows side effects in order to transform models. This extension provides for operators and modifies the semantics of the OCL sequence.

pOcl operators

"*model*" sets up the default model on which the rules will be evaluated. To create a new model, it is necessary to use a "model" operation followed by "*new*" and the metamodel and model name (Example 21 - ❶).

Syntax: ***model new*** MetamodelName***[***ModelName***]***

	The created model does not appear in the browser model (Figure 18). To display it, it is necessary to use the "refresh" item of the model browser context menu.

"*new*" creates a new instance of the mentioned meta-class. This new instance is created in the model stated in the meta-class path or in the default model (Example 21 - ❷).

"<-" is the assignment operator. The behavior of this operator depends on the left member type. If the type is a data type or a meta-class, this operator affects the result of the right expression in the left member (Example 21 - ❸). If the type is a collection this operator adds the result of the right expression in the collection specified in the left member (Example 21 - ❻). The evaluation of the assignment operator returns a boolean: true if the assignment is correctly processed and false otherwise.

"*transform*" runs the transformation. All assignments and creations are performed.

	If a rule contains assignments or creations without using the "*transform*" operator, an exception is generated indicating an inconsistency in the rule definition.

	It is impossible to create a new model element of the same type as the context of the rule.
	`context mmodel1_5[modelClass]::Foundation::Core::Class`
	`def : rule : Sequence(Boolean) =`
	`  Sequence {`
	`  class <- new mmodel1_5[modelClass]::Foundation::Core::Class,`
	`  class.name <- self.name,`
	`  class.generalization <- self.Generalization(class)`
	`  }`
	This rule is forbidden.

```
model new mmodel1_5[modelClass]                                          ❶


context mmodel1_5[libro]::BehavioralElements::StateMachines::SimpleState
def : State2Class : Sequence(Boolean) =
  Sequence {
    class <- new mmodel1_5[modelClass]::Foundation::Core::Class,         ❷
    class.name <- self.name,                                             ❸
    class.generalization <- self.Generalization(class)
   }
```

```
context mmodel1_5[libro]::BehavioralElements::StateMachines::SimpleState
def : ClassPere : mmodel1_5[modelClass]::Foundation::Core::Class =
  if Sequence {                                                          ❹
    class <- new mmodel1_5[modelClass]::Foundation::Core::Class,
    class.name <- 'State'}->excludes(false) then
      class
  else
     self.oclUndefined                                                   ❺
  endif


context mmodel1_5[libro]::BehavioralElements::StateMachines::SimpleState
def : Generalization(fils: mmodel1_5[modelClass]::Foundation::Core::Class) :
Sequence(Boolean) =
  Sequence {
    gen <- new mmodel1_5[modelClass]::Foundation::Core::Generalization,
    if  mmodel1_5[modelClass]::Foundation::Core::Class::allInstances
       -> exists(c: mmodel1_5[modelClass]::Foundation::Core::Class | c.name='State')
    then
      gen.parent <- mmodel1_5[modelClass]::Foundation::Core::Class::allInstances
       -> select(c: mmodel1_5[modelClass]::Foundation::Core::Class | c.name='State')
    else
      gen.parent <- self.ClassPere                                       ❻
    endif,
    gen.child <- fils
  }
context mmodel1_5[libro]::BehavioralElements::StateMachines::SimpleState
transform : self.State2Class                                             ❼
```

Example 21: transformation example

Example 21 illustrated a transformation consisting in generating a class diagram from a state-transition diagram using the design pattern state. This example is focused on the creation of one class by state having a common class ancestor called "*State*". The source model is called "*libro*" and the target model "*modelClass*".

It is possible to make on place transformation. However, the lack of an operator removing an element of a collection limits this kind of transformation. So on place transformation must be limited to punctually adding or modifying a property. If the transformation is more complex, it is necessary to create a new model.

Line ❺ in Example 21 will return an "*oclUndefined*" object. "*oclUndefined*" is an instance of the bottom type "*OclVoid*" that can be substituted to all types and *meta-classe* used.

Sequence semantic

To perform a transformation, the semantic of the "*sequence*" collection has been modified in order to force the evaluation of the sequence's elements from the left to the right. This addition to the sequence semantic does not interfere with the initial semantics because no information about the evaluation of sequence's elements is described in the OCL standard.

pOcl extension for model serialization

As NEPTUNE delivers tools to transform models, it is also necessary to provide functionalities for model serialization. In this version, two serializations are provided: text serialization and XMI serialization. To support serialization, the “OclVoid” and “Collection” have been enriched with new operations. There are required in order to serialize models in XMI format and two for serialization in Text format.

This extension is experimental and no test has been performed yet.

XMI format

“OclVoid_int outputXMI(OclVoid_int father, Bag_int<OclVoid_int> attributs)”: This operation creates a new XMI mark having the type of the context of the rule as a name. “*father*” provides either the name of the file if no name has been specified (❶), or the name of the mark (❸). “*attributs*” represents the mark’s attributes. The attribute “*xmi.id*” is automatically generated. This operation returns the element on which the rule is applied.

“*OclVoid_int outputXMI(OclVoid_int grandfather, OclVoid_int father, Bag_int<OclVoid_int> attributs)*”: This operation is similar to the previous one but references both the grandfather and the father of the mark. The “*xmi.idref*” is derived from the object invoking the operation. In (❹), “*xmi.idref*” references the “*xmi.id*” calculated from “*self.child*”. This operation returns the element on which the rule is applied.

“*Collection_int<T> outputRoleXMI(OclVoid_int pere, OclVoid_int role)*”: This operation creates attributes for association ends. This operation is only applicable on collections containing the element referenced by the association end. This operation returns the elements on which the rule is applied.

```
context ModelManagement::Model
query : self.outputXMI( 'apres.xmi',Bag{'xmi.id','isLeaf','name','isSpecification',
                        'isRoot','isAbstract'})                                    ❶

context ModelManagement::Model
query : Bag{self}->outputRoleXMI(self,'Namespace.ownedElement')                    ❷

context Foundation::Core::Class
query : self.outputXMI('Namespace.ownedElement',
            Bag{'xmi.id','name','visibility','isActive','isLeaf','isRoot'})       ❸

context Foundation::Core::Generalization
query :self.outputXMI('Namespace.ownedElement',Bag{'xmi.id','isSpecification'})

context Foundation::Core::Generalization
query : Bag {self}->outputRoleXMI(self,'Generalization.child')

context :Foundation::Core::Generalization
query : Bag {self}->outputRoleXMI(self,'Generalization.parent')

context Foundation::Core::Generalization
query : self.child.outputXMI(self,'Generalization.child', Bag{'xmi.idref'})        ❹

context Foundation::Core::Generalization
query : self.parent.outputXMI(self,'Generalization.parent',Bag{'xmi.idref'})
```

Example 22: serialization rule for class diagram in XMI format


```
<?xml version="1.0" encoding="UTF-8"?>
<XMI xmlns="UML" xmi.version="1.2" timestamp="Sat Nov 28 22:13:50 CET 2009"
xmlns:UML="org.omg.xmi.namespace.UML">
   <XMI.header>
      <XMI.documentation>
         <XMI.exporter>NEPTUNE v1.1</XMI.exporter>
         <XMI.exporterVersion>
            0.24(5) revised on $Date: 2006-11-06 19:55:22 +0100 (Mon, 06 Nov 2006) $
         </XMI.exporterVersion>
      </XMI.documentation>
      <XMI.metamodel xmi.name="UML" xmi.version="1.4"/>
   </XMI.header>

   <XMI.content>
      <UML:Model xmi.id="neptune.metamodeles.mmodel1_5.ModelManagement.Model@9b6bcd"
            isLeaf="false" name="untitledModel" isRoot="false" isAbstract="false">
         <UML:Namespace.ownedElement>
            <UML:Class
               xmi.id="neptune.metamodeles.mmodel1_5.Foundation.Core.Class@633c70"
               name="Toto" isActive="false" isLeaf="false" isRoot="false"/>
         </UML:Namespace.ownedElement>
      </UML:Model>
   </XMI.content>
</XMI>
```

Example 23: example of a serialization using rules defined in Example 22

Example 23 shows an execution of the rules of the Example 22 on the trivial class diagram containing only one class: “Toto”. The result of this execution can be loaded in the NEPTUNE II platform.

Text format

“OclVoid_int outputText(OclVoid_int file, OclVoid_int text)”: this operation creates a new text file whose name is contained in “*file*” and writes the text “*text*” in it. This operation must be invoked first. It returns the element on which the rule is applied.

“OclVoid_int outputText(OclVoid_int texte)”: this operation writes the text “*text*” in the file previously created using the operation “*outputText (OclVoid_int file, OclVoid_int text)”*. This operation returns the elements on which the rule is applied.

When serializing in text mode, the first operation used must always be “*outputText(OclVoid_int file, OclVoid_int text)”*in order to create the file “*file*”.

```
context Foundation::Core::Attribute
def : construireAtt : String =
            'private ' + self.type.name + ' ' + self.name + ' ;\n'

def : construireSetAtt : String =
        '\tpublic void set' + self.name.firstToUppercase +'(' + self.type.name +
        ' val) \n' +
        '\t{ \n\t this.'+ self.name + ' = val ;\n\t} \n'

def : construireGetAtt : String =
        '\tpublic ' + self.type.name + ' set' + self.name.firstToUppercase +'() \n' +
        '\t{ \n\t return this.'+ self.name + ' ;\n\t} \n'

context Foundation::Core::Class
query : self.outputText('toto.txt',
                        'public class ' + self.name + '\n' +
                        '{' +
                         '\t' + self.feature->select(f |
                                   f.oclIsKindOf(Foundation::Core::Attribute))->

                                iterate(att : Foundation::Core::Attribute;
                                        str : String = '' |
                                              str + att.construireAtt + '\n' +
                                               att.construireSetAtt + '\n' +
                                              att.construireGetAtt) +'}')
```

Example 24: serialization rule for class diagram in text format

```
public class Toto
{     private Integer age ;

      public void setAge(Integer val)

      {
       this.age = val ;
      }

      public Integer setAge()
      {
       return this.age ;
      }
}
```

Example 25: example of a serialization using rules defined Example 24

Example 25 shows the result of the rule defined in Example 24 performed on the class diagram of the Example 22. This rule generates a Java class schema from the UML class diagram.

Other components can be implemented for serialization of models in different formats: Java, C++, Ada…

Running NEPTUNE interpreter in batch mode

It is possible to execute the NEPTUNE interpreter in batch mode i.e. without using the graphical interface. The syntaxes are :

- ***java -Xms1548M -Xmx1548M -version:"1.6*" -jar batch.jar -metamodel idMetamodel -modelPath filePathTotheXMI -modelName name -ocl oclruleToExecute [-result pathOfTheResultFile] [-output pathOfoutput] [ocl checker options]***
- ***java -Xms1548M -Xmx1548M -version:"1.6*" -jar batch.jar -metamodel idMetamodel -modelPath filePathTotheXMI -modelName name -oclfile filePathToTheFileContainingTheRulesToExecute [-output pathOfoutput] [-result pathOfTheResultFile] [ocl checker options]***

where:

-metamodel idMetamodel	idMetamodel is the name of the metamodel of which the model is an instance (mandatory).
-modelPath filePathToTheXMI	filePathTotheXMI is the file path to the XMI file containing the model (mandatory).
-modelName name	name identifier of the model. This identifier is useful for identifying the model in an OCL rule (mandatory).
-ocl oclruleToExecute	oclruleToExecute is a string containing the OCL rule to execute.
-oclfile filePathToTheFileContainingTheRulesToexecute	filePathToTheFileContainingTheRulesToexecute is the file path to the file containing the OCLrules to execute.

Either –ocl or –oclfile option must be set. These two options are mutually exclusive.

-output pathOfoutput	pathOfoutput is the path of the file where the outputs and the errors are printed (optional).
-result pathOfTheResultFile	pathOfTheResultFile is the path of the file where the results of the evaluation are saved. The format of this file is csv (optional).
ocl checker options	These options are those defined for the NEPTUNE interpreter (***see page 10***)

Example 26 shows an example of an execution of the rules in the file ***essai.ocl*** on the model ***cec*** stored in the file ***cec.gens.*** The model ***cec*** conforms to the metamodel ***mmCec***.

```
java -Xms512M -Xmx512M -version:"1.6*" -jar "C:\NEPTUNE\batch.jar" -metamodel mmCec -
modelPath "C:\NEPTUNE\example\cec.gens" -modelName cec -oclfile
"C:\NEPTUNE\example\essai.ocl" -result " C:\NEPTUNE\example\result.csv" -output "
C:\NEPTUNE\example\sortie_.txt"
```

Example 26: execution of an OCL rule in batch mode

Conformance

	OCL-MOF subset	Full OCL
Syntax	Yes	Yes
XMI	Not supported	Not supported
Evaluation		
• allInstances	Yes	Yes
• @pre postconditions	Only syntax is checked	Only syntax is checked
• OclMessage	Not supported	Not supported
• Navigating non-navigable associations[1]	No	No
• Accessing private and protected features	Yes	Yes

"The UML 2.0 Infrastructure and the MOF 2.0 Core specifications that were developed in parallel with this OCL 2.0 specification share a common core. The OCL specification contains a well-defined and named subset of OCL that is defined purely based on the common core of UML and MOF. This allows this subset of OCL to be used with both the MOF and the UML, while the full specification can be used with the UML only.

The following compliance points are distinguished for both parts.

1. Syntax compliance. The tool can read and write OCL expressions in accordance with the grammar, including validating its type conformance and conformance of well-formedness rules against a model.

2. XMI compliance. The tool can exchange OCL expressions using XMI.

3. Evaluation compliance. The tool evaluates OCL expressions in accordance with the semantics chapter. The following additional compliance points are optional for OCL evaluators, as they are dependent on the technical platform on which they are evaluated.

• allInstances()

• pre-values and oclIsNew() in postconditions

• OclMessage

• navigating across non-navigable associations

• accessing private and protected features of an object

The following table shows the possible compliance points. Each tool is expected to fill in this table to specify which compliance points are supported." [OCL06]

To validate the NEPTUNE OCL interpreter we used the benchmark for OCL engine developed by Martin Gogolla [GKB08]. We only used tests that can be applied on metamodels. In addition, we did not make the efficiency tests due to a lack of time.

[1] Several tries have been realized in order to navigate non-navigable associations. We discard this feature due to the number of ambiguities generated by the lack of name of the association-end no-navigable. Generating random names does not seem to be a useful solution.

Limitation and troubleshooting

- The context menu of the model browser is displayed on the bottom of the NEPTUNE window and not on the element in the browser.

- The NEPTUNE platform is not able to manage the UML 2.x.x stereotypes. Indeed, UML 2.x.x stereotypes are defined at both the M1 level and M2 level. We are carrying out studies in order to propose an acceptable implementation for stereotypes.

- undo/redo is not completely operational.

- Deleting a model is not working correctly for models having errors. Java error are generated and it is necessary to stop NEPTUNE and reruns it.

Bibliography

[GKB08] Gogolla M., Kuhlmann M., Buttner F., 2008, Source for a Benchmark for OCL Engine Accuracy, Determinateness, and Efficiency.

[OCL06] Object Constraint Language - OMG Available Specification - Version 2.0 - formal/06-05-01 May 2006 - http://www.omg.org/spec/OCL/2.0/PDF

[OCL09] Object Constraint Language (Beta 2) - OMG Available Specification - Version 2.1 - ptc/09-05-02 May 2009 - http://www.omg.org/spec/OCL/2.1/Beta2/PDF

Version notes

Version	*Date*	*Author*	*Comments*
V 1	18/11/2009	Thierry Millan	Initial version
V 2 beta	24/06/2011	Jonathan Garrigues	New NEPTUNE functionalities
V 2	12/06/2012	Thierry Millan	
V 2	10/02/2013	Renan Leroux	Installation documentation

Illustration table

Example table