

How to use the OCL2 Interpreter

Claas Wilke

March 11, 2009

This tutorial describes how the OCL2 Interpreter provided with *Dresden OCL2 for Eclipse* can be used. A general introduction into *Dresden OCL2 for Eclipse* can be found in [WB09].

The procedure described in this tutorial was realized and tested with *Eclipse 3.4.1* [Ecl09]. In addition to that this tutorial should also run with *Eclipse 3.3.x*. Besides *Eclipse* some required plug-ins must be installed. Table 1 shows all required software to run *Dresden OCL2 for Eclipse* and the OCL2 Interpreter.

Software	Available at
Eclipse 3.4.x	http://www.eclipse.org/
Eclipse Modeling Framework (EMF)	http://www.eclipse.org/modeling/emf/
Eclipse Model Development Tools (MDT) (only with the UML2.0 meta model)	http://www.eclipse.org/modeling/mdt/
Eclipse Plugin Development Environment (only to run the toolkit using the source code distribution)	http://www.eclipse.org/pde/

Figure 1: Required software to run Dresden OCL2 for Eclipse and the OCL2 Interpreter.

1 How to Run Dresden OCL2 for Eclipse

How to install and run *Dresden OCL2 for Eclipse* and how to load models and OCL constraints will not be explained in this tutorial. This tutorial assumes that the user is familiar with such basic uses of the toolkit. A general introduction into *Dresden OCL2 for Eclipse* can be found in [WB09].

2 The Simple Example

This tutorial uses the *Simple Example* which is provided with *Dresden OCL2 for Eclipse* located in the plug-in package `tudresden.oc120.pivot.examples.simple`. An overview over all examples provided with *Dresden OCL2 for Eclipse* can be found in [Wil09]. Figure 2 shows the class structure which is described by the *Simple Example*.

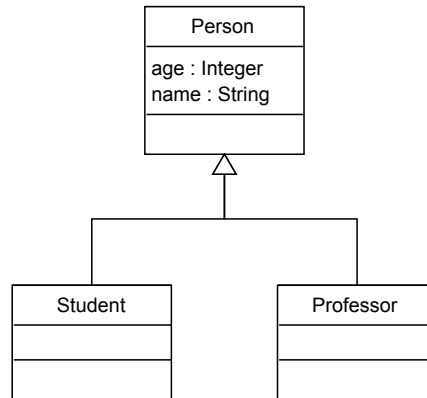


Figure 2: The class structure described by the Simple Example model.

The model of the example defines three classes: The class **Person** has two attributes **age** and **name**. Two subclasses of **Person** are defined, **Student** and **Professor**.

To import the *Simple Example* into our Eclipse workspace we create a new Java project into our Workspace called `tudresden.oc120.pivot.examples.simple` and use the import wizard *General > Archive File* to import the example provided as jar archive. In the following window we select the directory where the jar file is located (eventually the `plugins` directory into the Eclipse root folder) and we select the archive `tudresden.oc120.pivot.examples.simple.jar` and click the *Finish* button (if you use a source code distribution of *Dresden OCL2 for Eclipse* instead, you can simply import the project `tudresden.oc120.pivot.examples.simple` using the import wizard *General -> Existing Projects into Workspace*). Figure 3 shows the package Explorer containing the imported project.

The project provides a model file which contains the simple class diagram (the model file is located at `model/simple.uml`) and the constraint file we want to interpret (located at `constraints/allConstraints.oc1`). Listing 1 shows the constraints defined in the constraint file.

First, the constraint file defines three simple invariants which denote, that the **age** of every **Person** must always be zero or greater than zero. Furthermore, the **age** of every **Student** must be greater than 16 and the **age** of every **Professor** does not have to be lesser than 30.

In addition to that the constraint file contains a definition constraint which defines a new operation `getAge()` which returns the **age** of a **Person**. A precondition checks, that the **age** must be defined before it can be returned by the operation `getAge()`. And finally, a postcondition checks, whether or not the result of the operation `getAge()` is the same as the **age** of the **Person**.

3 Preparation of the Interpretation

To prepare the interpretation we have to import the model `model/simple.uml` for which we want to interpret constraints into the *Model Browser*. We use the

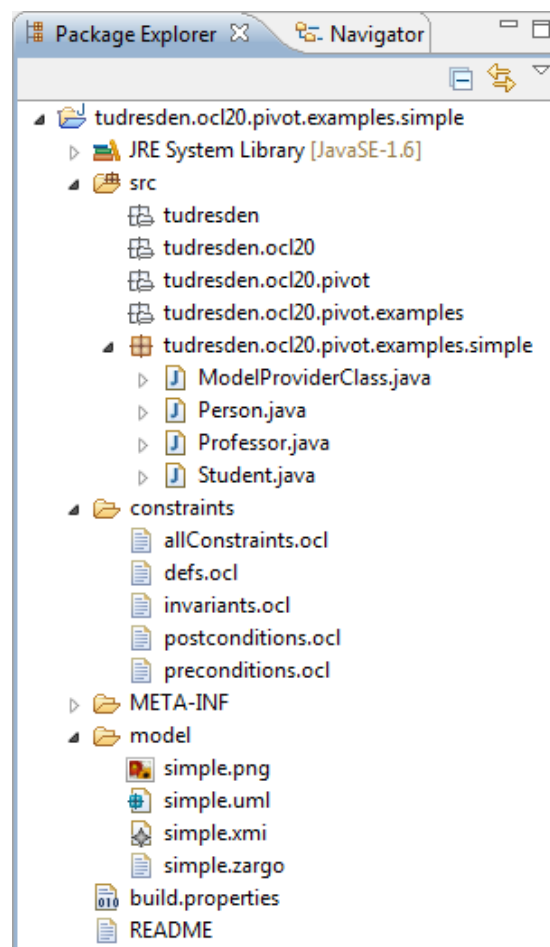


Figure 3: The package explorer containing the project which is needed to run this tutorial.

```

1  — The age of Person can not be negative.
2  context Person
3  inv: age >= 0
4
5  — Students should be 16 or older.
6  context Student
7  inv: age > 16
8
9  — Professors should be at least 30.
10 context Professor
11 inv: not (age < 30)
12
13 — Returns the age of a Person.
14 context Person
15 def: getAge(): Integer = age
16
17 — Before returning the age, the age must be defined.
18 context Person::getAge()
19 pre: not age.ocIsUndefined()
20
21 — The result of getAge must equal to the age of a Person.
22 context Person::getAge()
23 post: result = age

```

Listing 1: The constraints contained in the constraint file.

import wizard for domain specific models of the toolkit to import the model. This procedure is explained in the already mentioned general tutorial ([WB09]). Furthermore, we have to import a model instance for which the constraints shall be interpreted into the *Model Instance Browser*. We use another import wizard to import the model instance `bin/tudresden/ocl20/pivot/examples/simple/ModelProviderClass.class`. Finally, we have to import the constraint file `constraints/allConstraints.ocl` containing the constraints we want to interpret. The import is done by an import wizard again. Afterwards, the *Model Browser* should look like illustrated in figure 4 and the *Model Instance Browser* should look like shown in figure 5.

The loaded model instance contains three instances of the classes defined in the *Simple Example* model. One instance of **Person**, one instance of **Student** and one instance of **Professor**. For these three instances we now want to interpret the imported constraints.

4 Interpretation

Now we can start the interpretation. To open the *OCL2 Interpreter* we use the menu option **Dresden OCL2 > Open OCL2 Interpreter**. The *OCL2 Interpreter View* should now be visible (see figure 6).

By now, the *OCL2 Interpreter View* does not contain any result. Besides the results table, the view provides four buttons to control the *OCL2 Interpreter*. The buttons are shown in figure 7. With the first button (from left to right) constraints can be prepared for interpretation. The second button can be used

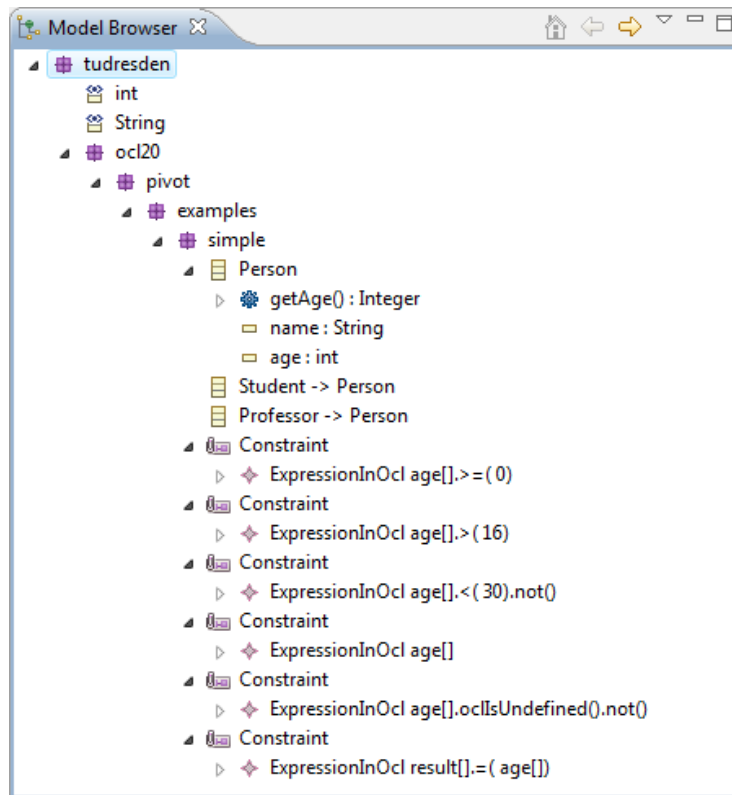


Figure 4: The model browser containing the simple model and its constraints.

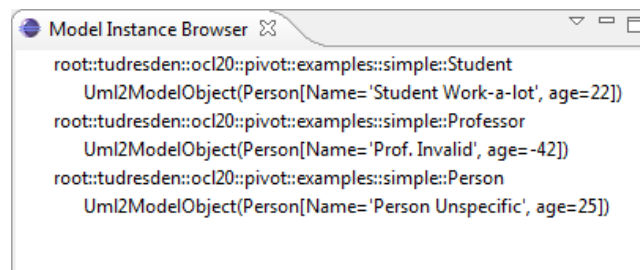


Figure 5: The model instance browser containing the simple model instance.

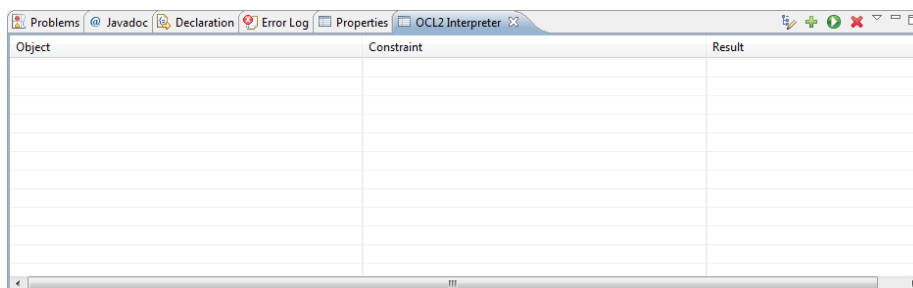


Figure 6: The OCL2 Interpreter containing no results.

to add variables to the *Interpreter Environment*. The third button provides the core functionality, it can be used to start the interpretation. And finally, the fourth button provides the possibility to delete all results from the *OCLE2 Interpreter View*. The functionality of the buttons will be explained below.



Figure 7: The buttons to control the OCL2 Interpreter.

4.1 Interpretation of Constraints

To interpret constraints, we simply select them in the *Model Browser* and press the button to interpret constraints (the third button from the left). First, we want to interpret the three invariants defining the range of the **age** of **Persons**, **Students** and **Professors**. We select them in the *Model Browser* (see figure 8) and click the *Interpret* button. The result of the interpretation is now shown in the *OCLE2 Interpreter View* (see figure 9).

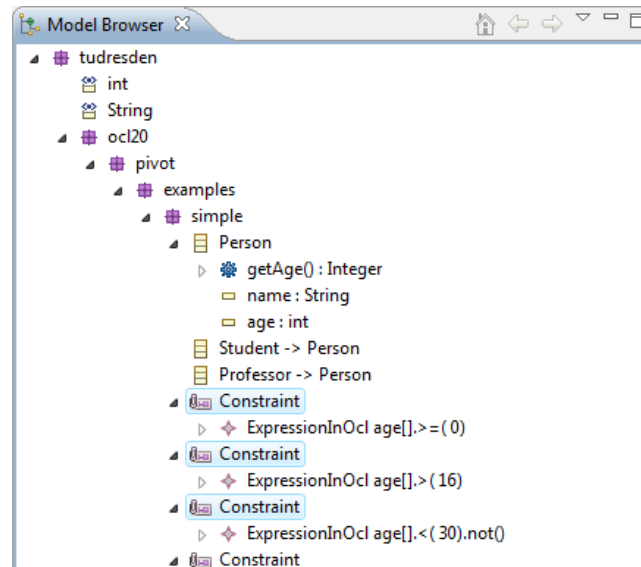


Figure 8: The three age invariants selected in the Model Browser.

Object	Constraint	Result
Uml2ModelObject(Person[Name='Student Work-a-lot', age=22])	invariant : age[] >= (0)	JavaOclBoolean(true)
Uml2ModelObject(Person[Name='Student Work-a-lot', age=22])	invariant : age[] > (16)	JavaOclBoolean(true)
Uml2ModelObject(Person[Name='Prof. Invalid', age=-42])	invariant : age[] < (30).not()	JavaOclBoolean(false)
Uml2ModelObject(Person[Name='Prof. Invalid', age=-42])	invariant : age[] >= (0)	JavaOclBoolean(false)
Uml2ModelObject(Person[Name='Person Unspecific', age=25])	invariant : age[] >= (0)	JavaOclBoolean(true)

Figure 9: The results of the three age invariants for all model instances.

The invariant `age >= 0` has been interpreted for all three model objects. The results for the `Person` and the `Student` instances are `true` because their `age` is greater than zero. The result for the `Professor` instance is `false` because its `age` is `-42`.

The two other invariants were only interpreted for the `Student` or the `Professor` instance because their context was not the class `Person` but the class `Student` or the class `Professor`. Again the `Student`'s result is `true` and the `Professor`'s result is `false`.

4.2 Preparation of Constraints

Some constraints cannot be interpreted because they are no constraints in the natural sense of the word constraint. OCL enables us to use constraints to define new attributes and methods or to initialize attributes and methods. Such `def`, `init` and `body` constraints cannot be interpreted because they have no result. They can only be used to alter the results of other constraints which shall be interpreted.

The *Simple Example* constraint file contains a definition constraint, which defines the method `getAge()` for the class `Person`. Before we can refer to this method in other constraints we have to prepare the definition constraint to ensure, that the interpretation of other constraints will finish with the right results.

To prepare the definition constraint, we select it in the *Model Browser* (see figure 10) and click the *Prepare* button (the first button from the left).

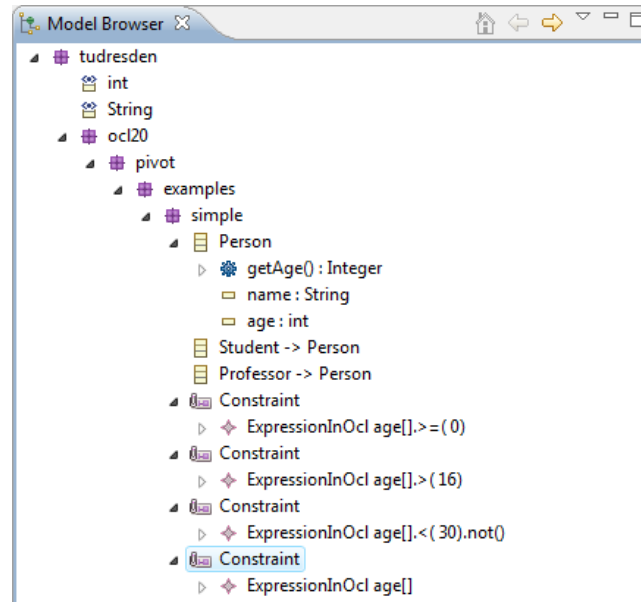


Figure 10: The definition constraint selected in the Model Browser.

The preparation does not result with a visible result in the *OCL2 Interpreter View*. But the method definition of the constraint has been added to the *Interpreter Environment* of the *OCL2 Interpreter*. Thus, we can interpret the next

constraint now. This constraint is the precondition which checks that the **age** of any **Person** must be defined before the method **getAge()** can be invoked.

We select the constraint in the *Model Browser* (see figure 11) and click the *Interpret* button. The result of the interpretation is shown in figure 12.

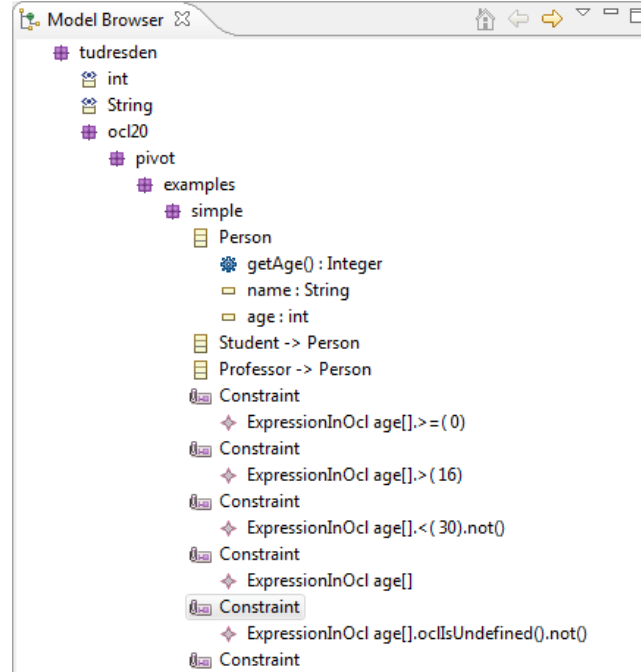


Figure 11: The precondition selected in the Model Browser.

Object	Constraint	Result
Uml2ModelObject(Person[Name='Student Work-a-lot', age=22])	context getAge(); precondition : age[].oclIsUndefined().not()	JavaOclBoolean(true)
Uml2ModelObject(Person[Name='Prof. Invalid', age=-42])	context getAge(); precondition : age[].oclIsUndefined().not()	JavaOclBoolean(true)
Uml2ModelObject(Person[Name='Person Unspecific', age=25])	context getAge(); precondition : age[].oclIsUndefined().not()	JavaOclBoolean(true)

Figure 12: The results of the precondition for all model instances.

The interpretation finishes for all three instances successfully because the attribute **age** has been set for all three instances.

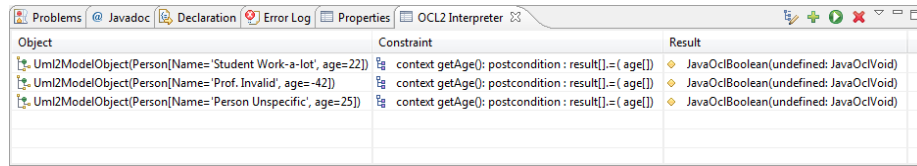
4.3 Adding Variables to the Environment

By preparing the definition constraint we added some information to the *Interpretation Environment* which was necessary to interpret other constraints. For some constraints we have to add further information which can not be provided by the preparation of other constraints.

For example, our last constraint a postcondition compares the result of the method **getAge()** with the attribute **age** of the referenced **Person** instance. Therefore, OCL provides the special variable **result** in postconditions which

contains the result of the constrained method's execution. Using the *OCL2 Interpreter View* we cannot execute the method `getAge()` and store the result in the `result` variable. We can interpret the postcondition in a specific context which has to be prepared by hand only. We have to set the result variable manually.

If we interpret the postcondition constraint (the sixth and last constraint in the *Model Browser*) without setting the `result` variable, the constraint results in a **undefined** result for all three model instances (see figure 13).



Object	Constraint	Result
Uml2ModelObject(Person[Name='Student Work-a-lot', age=22])	context getAge(); postcondition : result[] = (age[])	JavaOclBoolean(undefined: JavaOclVoid)
Uml2ModelObject(Person[Name='Prof. Invalid', age=-42])	context getAge(); postcondition : result[] = (age[])	JavaOclBoolean(undefined: JavaOclVoid)
Uml2ModelObject(Person[Name='Person Unspecific', age=25])	context getAge(); postcondition : result[] = (age[])	JavaOclBoolean(undefined: JavaOclVoid)

Figure 13: The results of the postcondition without preparing the result variable.

To prepare the variable we click on the button to add new variables to the *Interpreter Environment* (the second button from the left) and a new window opens which we can use to specify new variables. We enter the name `result`, select the variable type `Integer` and enter the value 25. Then we press the *OK* button (see figure repic:interpreter09. The result variable has now been added to the *Interpreter Environment*.

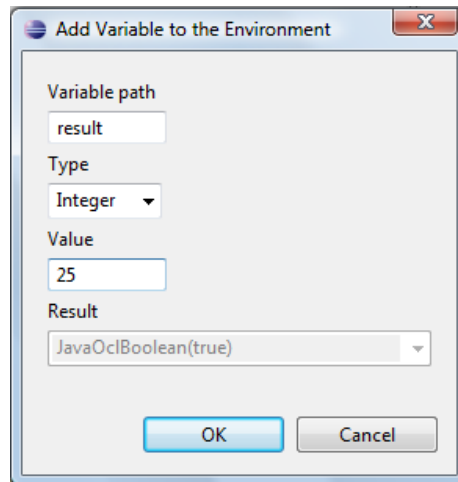


Figure 14: The window to add new variables to the environment.

Now we can interpret the postcondition again. The result is shown in figure 15. The results for the `Student` and `Professor` instances are both **false** because their `age` attribute is not equal to 25 and thus the `result` value does not match to the `age` attribute. But the interpretation for the `Person` instances succeeds because its age is 25.

Object	Constraint	Result
Uml2ModelObject(Person[Name='Student Work-a-lot', age=22])	context getAge(); postcondition : result[] = { age[] }	JavaOclBoolean(false)
Uml2ModelObject(Person[Name='Prof. Invalid', age=-42])	context getAge(); postcondition : result[] = { age[] }	JavaOclBoolean(false)
Uml2ModelObject(Person[Name='Person Unspecific', age=25])	context getAge(); postcondition : result[] = { age[] }	JavaOclBoolean(true)

Figure 15: The results of the postcondition with result variable preparation.

5 Conclusion

This tutorial described how OCL2 constraints can be interpreted using the *OCL2 Interpreter* of *Dresden OCL2 for Eclipse*. The preparation and interpretation of constraints has been explained, the addition of new variables to the *Interpreter Environment* has been shown.

More information about *Dresden OCL2 for Eclipse* can be found at [\[DOT09\]](#).

References

- [DOT09] *Website of the Dresden OCL2 Toolkit.* Website hosted by Sourceforge.net, 2009. – Available at <http://dresden-ocl.sourceforge.net/>
- [Ecl09] *Eclipse Website.* Eclipse Website hosted by the the Eclipse Foundation., 2009. – Available at <http://www.eclipse.org/>
- [WB09] WILKE, Claas ; BRANDT, Ronny: *An introduction into Dresden OCL2 for Eclipse.* Published at the Dresden OCL Toolkit Website., March 2009. – Available at http://dresden-ocl.sourceforge.net/4eclipse_usage.html
- [Wil09] WILKE, Claas: *Examples provided with Dresden OCL2 for Eclipse.* Published at the Dresden OCL Toolkit Website., February 2009. – Available at http://dresden-ocl.sourceforge.net/4eclipse_usage.html