How to use the Java Code Generator Ocl2Java

Claas Wilke

March 10, 2009

This tutorial describes how the Java Code Generator *Ocl2Java* provided with *Dresden OCL2 for Eclipse* can be used. A general introduction into *Dresden OCL2 for Eclipse* can be found in [WB09]. A detailed documentation of the development of *Ocl2Java* can be found in [Wil09b].

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* you also need to install some required plug-ins. Table 1 shows all required software to run *Dresden OCL2 for Eclipse* and *Ocl2Java*.

Software	**Available at**
Eclipse 3.4.x	http://www.eclipse.org/
Eclipse Modeling Framework (EMF)	http://www.eclipse.org/modeling/emf/
AspectJ Development Tools (AJDT) (only to run the generated code)	http://www.eclipse.org/aspectj/
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: Software needed to run *Dresden OCL2 for Eclipse* and *Ocl2Java*.

1 How to run Dresden OCL2 for Eclipse

How to run the toolkit *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 use of the toolkit. A general introduction into *Dresden OCL2 for Eclipse* can be found in [WB09].

2 The example used during this tutorial

This tutorial uses the *Simple Example* which is provided with *Dresden OCL2 for Eclipse* located in the plug-in package `tudresden.ocl20.pivot.examples.`

`simple`. An overview over all examples provided with *Dresden OCL2 for Eclipse* can be found in [Wil09a]. Figure 2 shows the class structure which is described by the *Simple Example*.

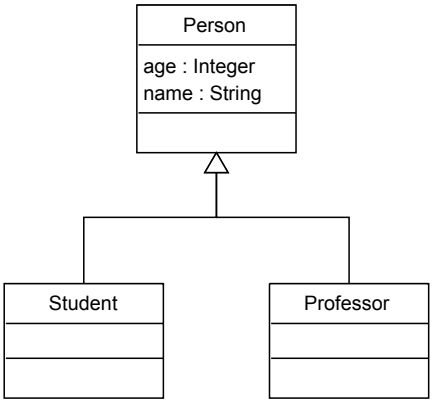

Figure 2: The class diagram described by the simple example model.

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.ocl20.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 were the jar file is located (eventually the `plugins` directory into the Eclipse root folder) and we select the archive `tudresden.ocl20.pivot.examples.simple.jar` and click the *Finish* button (if you use a source code distribution of *Dresden OCL2 for Eclipse* instead, you can simple import the project `tudresden.ocl20.pivot.examples.simple` using the import wizard *General -> Existing Projects into Workspace*).

Next, we have to import a second project called `tudresden.ocl20.pivot.examples.simple.constraints`. We can use the same mechanism explained above, but instead of a Java project we now create an AspectJ project before we import the archive file (if the wizard to create an AspectJ project is not available you have to install the *AspectJ Development Tools* first). Figure 3 shows the package Explorer containing both imported projects.

Now we have imported all files we need to run this tutorial. The first project provides a model file which contains the simple class diagram which has been explained above (the model file is located at `model/simple.uml`) and the constraint file we want to generate code for (the constraint file is located at `constraints/invariants.ocl`). Listing 1 shows one invariant which is contained in the constraint file for which we want to generate code. For this invariant which denotes, that the `age` of any `Person` must be greater or equal to zero at any time during the life cycle of the `Person`.

The second project provides a class `src/tudresden.ocl20.pivot.examples.simple.constraints.InvTest.java` which contains a jUnit test case which

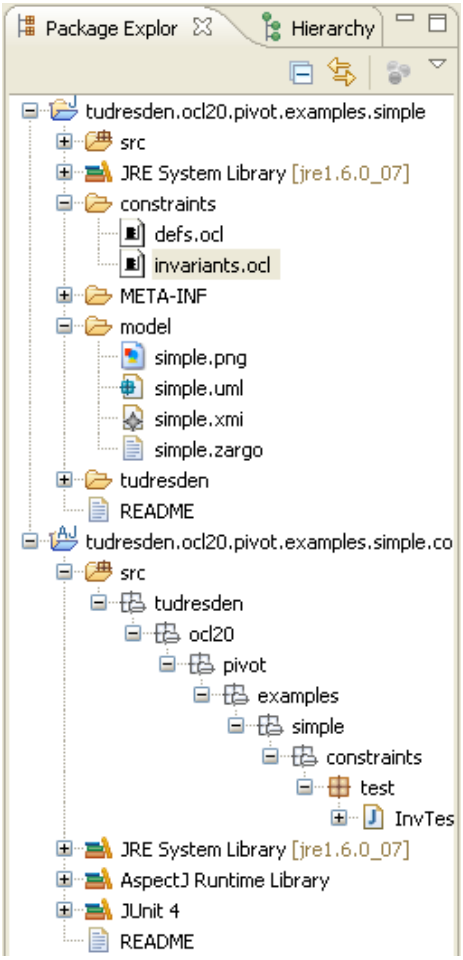

Figure 3: The package explorer containing the two projects which are needed to run this tutorial.

```
1 -- The age of Person can not be negative.
2 context Person
3 inv: age >= 0
```

Listing 1: A simple invariant.

checks, whether or not the mentioned constraint is enforced by generated code. The test case creates two `Persons` and tries to set their `age`. The `age` of the second `Person` is set to `-3` and thus the constraint is violated. The test case expects that a runtime exception is thrown, when the constraint is violated.

The code for the mentioned constraint has not been generated yet and thus the exception will not be thrown. We run the test case by right clicking on the Java class in the *Package Explorer* and selecting the menu item *Run as -> JUnit Test*. The test case fails because the exception is not thrown (see figure 4). To fulfill the test case we have to generate the ApsectJ code for the constraint which enforces the constraint's condition. How to create such code will be explained in the following.

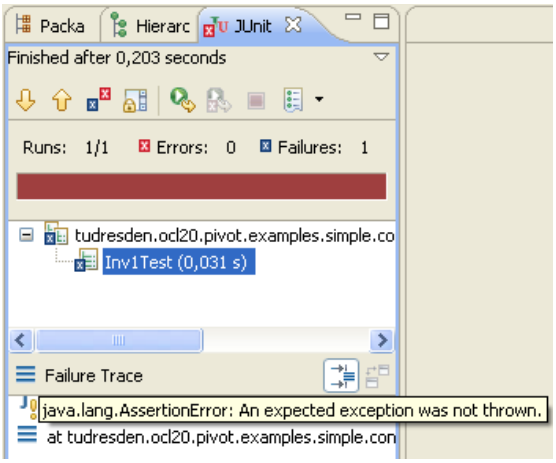

Figure 4: The result of the jUnit test case.

3 Prepare the Code Generation

To prepare the code generation we have to import the model `model/simple.uml` into the *Model Browser*. We use the import wizard for domain specific models of the toolkit to import the model. This procedure is explained in the already mentioned general tutorial ([WB09]). Then we have to import the constraint file `constraints/invariant.ocl` which is done by an import wizard again. Afterwards, the model explorer should look like illustrated in figure 5. Now we can start the code generation.

4 The Code Generation

To start the code generation we click on the menu item *DresdenOCL* and select the item *Generate AspectJ Constraint Code*.

4.1 Selecting a Model

A wizard opens and we have to select a model for code generation (see figure 6). We select the `simple.uml` model and click the *Next* button.

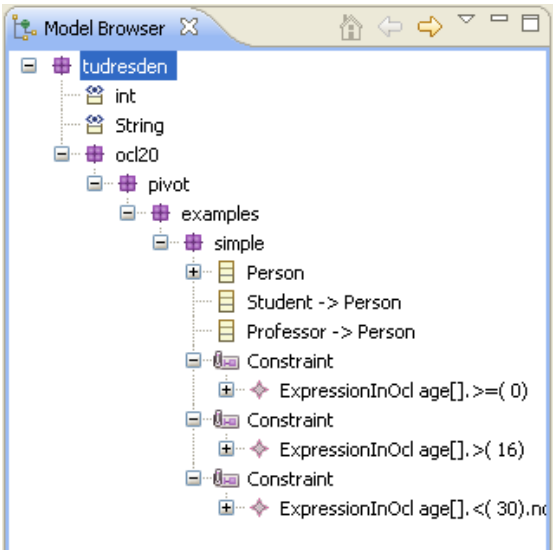

Figure 5: The model browser containing the simple model and its constraints.

4.2 Selecting Constraints

As a second step we now have to select the constraints for which we want to generate code. We only select the constraint which enforce that the `age` of any `Person` must be equal to or greater than zero and click the *Next* button (see figure 7).

4.3 Selecting a Target Directory

Next, we have to select a target directory to which our code shall be generated. We select the source directory of our second project (which is `tudresden.ocl20.pivot.examples.simple.constraints/src`) (see figure 8). Please note, that we select the source directory and not the package directory into which the code shall be generated! The code generator creates or uses contained package directories depending on the package structure of the selected constraint. Additionally we can specify a sub folder into which the constraint code shall be generated relatively to the package of the constrained class. By standard this is a sub directory called `constraints`. We don't want to change this setting and click the *Next* button.

4.4 Specifying General Settings

On the following page of the wizard we can specify general settings for the code generation (see figure 9).

We can disable the inheritance of constraints (which would not be useful in our example because we want to enforce the constraint for `Persons`, but for `Students` and `Professors` as well). We can also enable that the code generator will generate getter methods for new defined attributes of `def` constraints.

More interesting is the possibility to select one of three provided strategies, when invariants shall be checked during runtime:

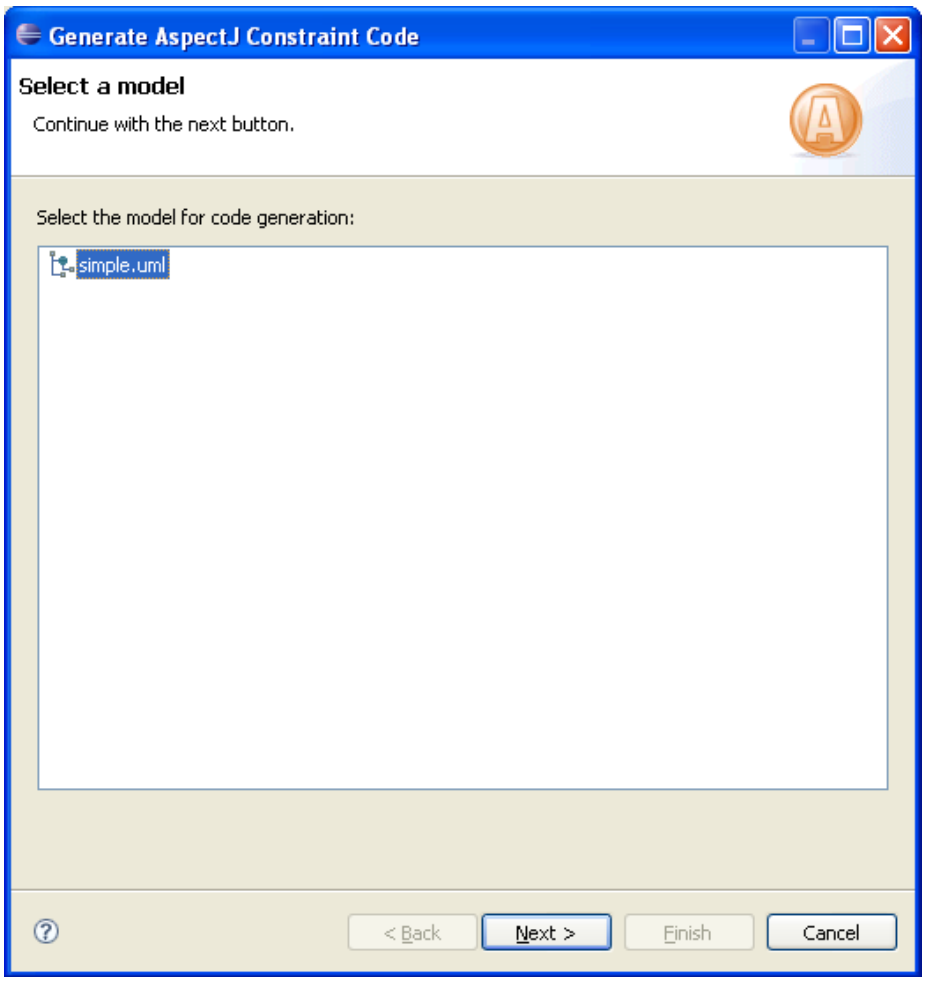

Figure 6: The first step: Selecting a model for code generation.

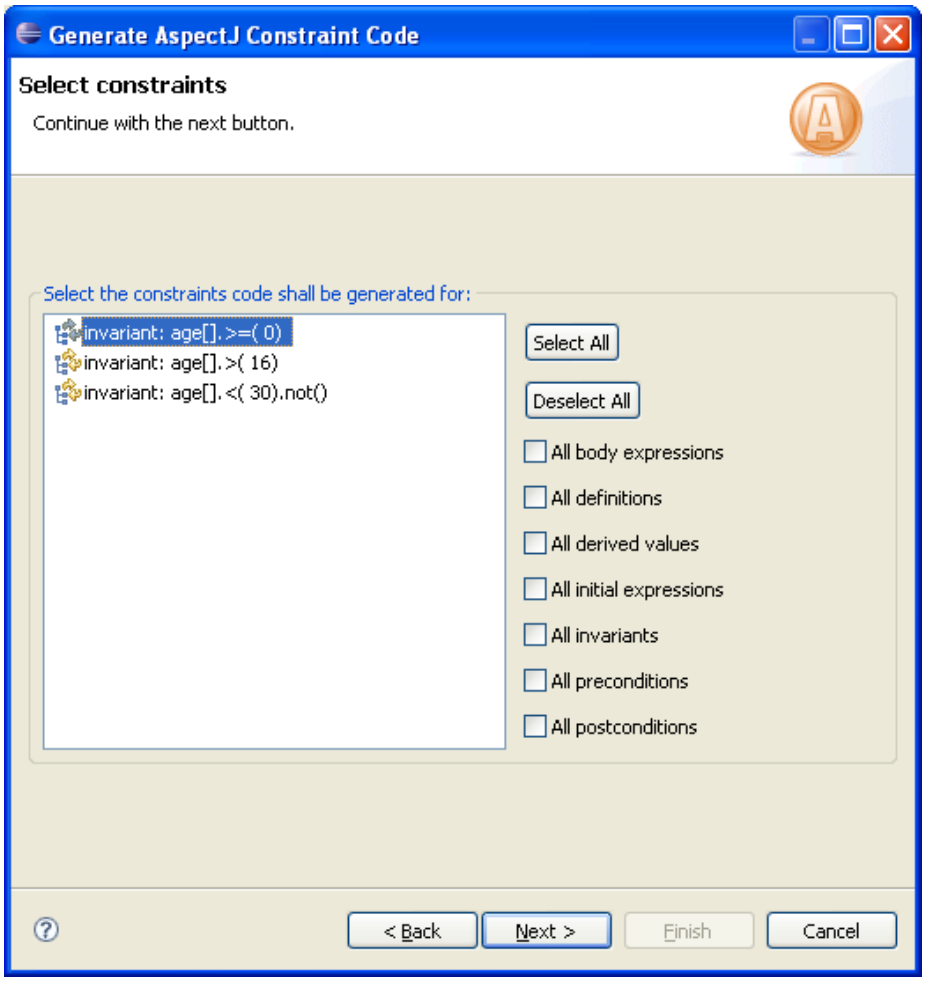

Figure 7: The second step: Selecting constraints for code generation.

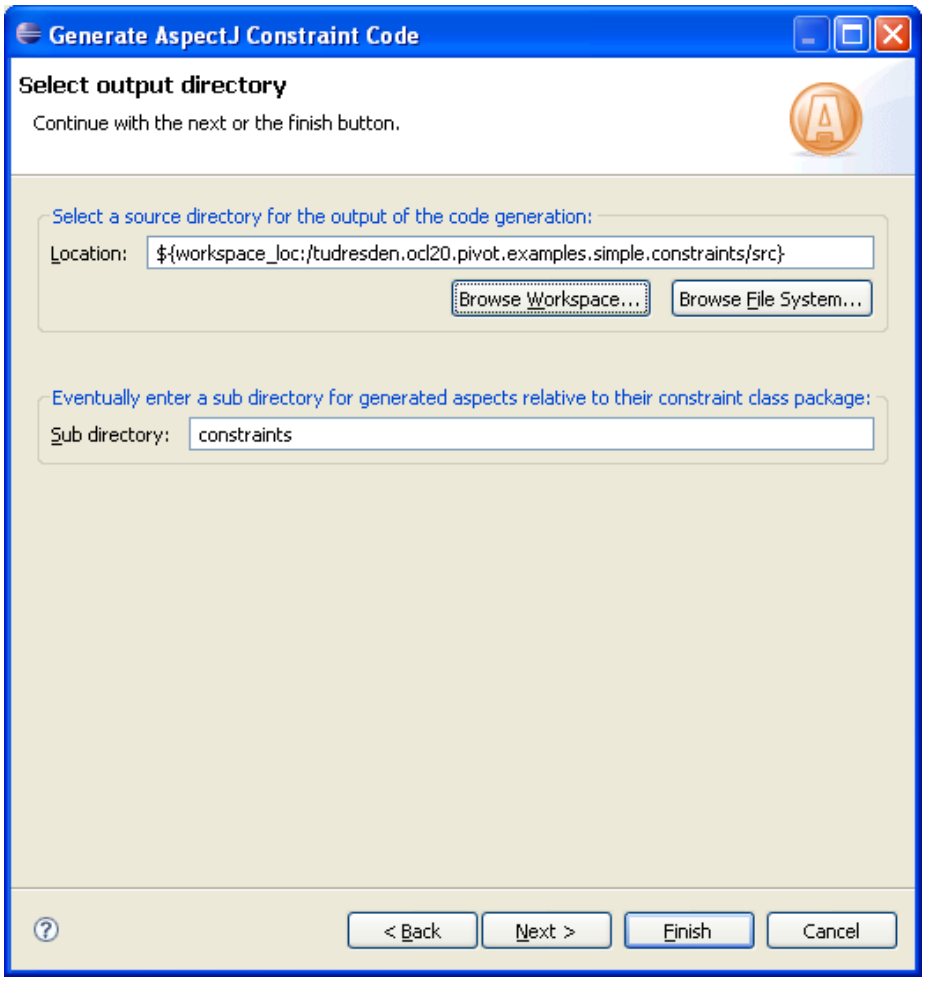

Figure 8: The third step: Selecting a target directory for the generated code.

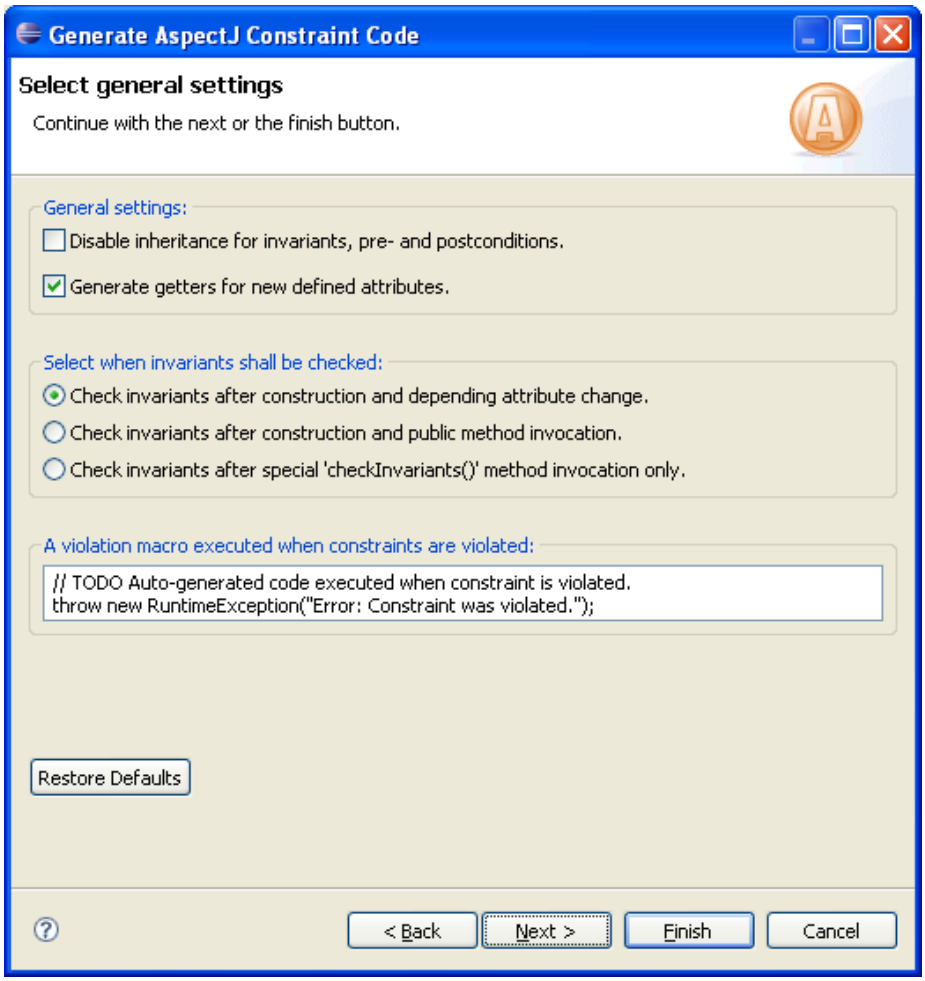

Figure 9: The fourth step: General settings for the code generation.

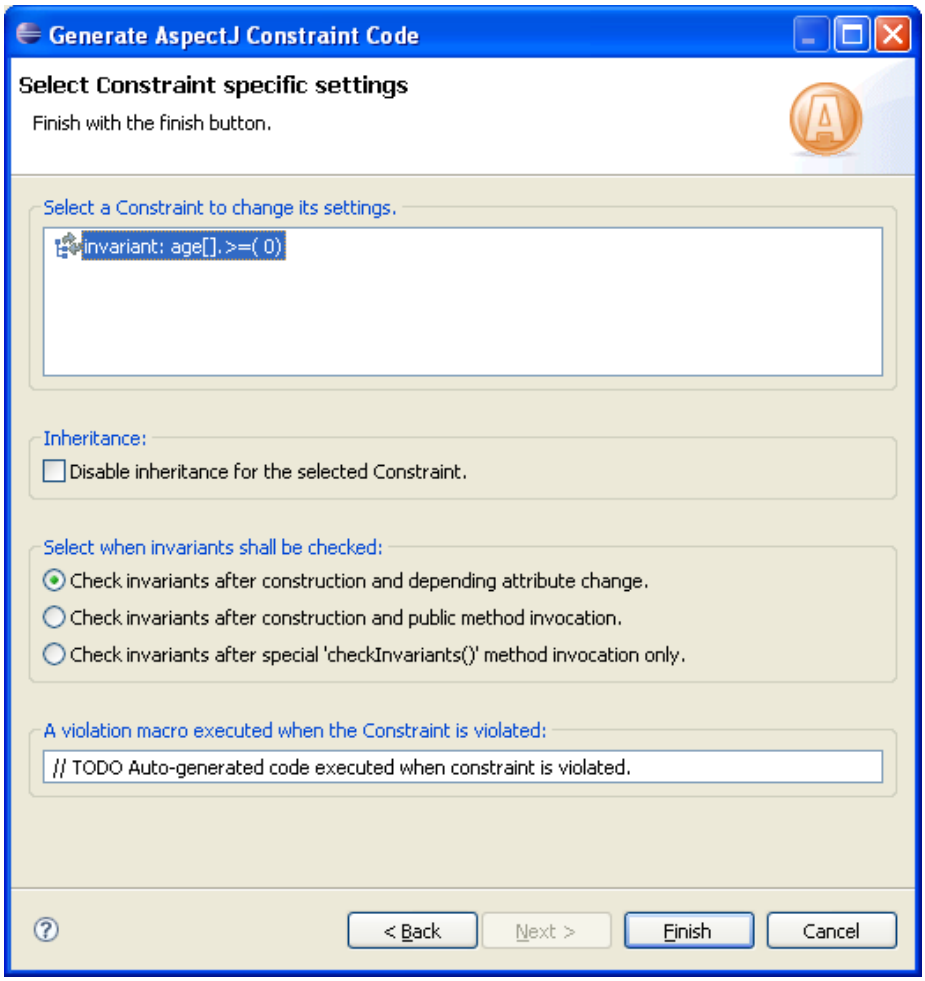

Figure 10: The fifth step: Constraint specific settings for the code generation.

1. Invariants can be checked after construction of an object and after any change of an attribute or association which is in scope of the invariant condition (*Strong Verification*).
2. Invariants can be checked after construction of an object and before or after the execution of any public method of the constrained class (*Weak Verification*).
3. And finally, invariants can only be checked if the user calls a special method at runtime (*Transactional Verification*).

This three scenarios can be useful for users in different situations. If a user wants to verify strongly, that his constraints are verified after any change of any dependent attribute he should use *strong verification*. If he wants to use attributes to temporary store values and constraints shall only be verified if any external class instance wants to access values of the constrained class, he should use *weak verification*. If the user wants to work with databases or other remote communication and the state of his constraint classes should be only valid before data transmission, he should use the scenario *transactional verification*.

Finally, we can specify a *violation macro* which specifies the code, which will be executed when a constraint is violated during runtime. By default, the *violation macro* throws a runtime exception. We also want to have a runtime exception thrown when our constraint is violated. Thus, we don't change the *violation macro* and continue with the *Next* button.

4.5 Constraint Specific Settings

The last page of the code generation wizard provides the possibility to configure some of the code generation settings constraint specific by selecting a constraint and adapting it's settings (see figure 10). We don't want to adapt the settings, thus we can finish the wizard and start the code generation by clicking the *Finish* button.

5 The Generated Code

After finishing the wizard, the code for the selected constraint will be generated. To see the result, we have to refresh our project in the workspace. We select the project `tudresden.ocl20.pivot.examples.simple.constraint` in the *Package Explorer* open the context menu with a right mouse click and select the menu item *Refresh*. Afterwards, our project contains a new generated AspectJ file called `tudresden.ocl20.pivot.examples.simple.constraints.InvAspect-01.aj` (see figure 11).

Now we can rerun our jUnit test case. The test case finishes successfully because the expected runtime exception is thrown (see figure 12).

6 Conclusion

This tutorial described how to generate AspectJ code using the *Ocl2Java* code generator of *Dresden OCL2 for Eclipse*. More information about *Dresden OCL2 for Eclipse* is available at the website of the *Dresden OCL Toolkit* [DOT09].

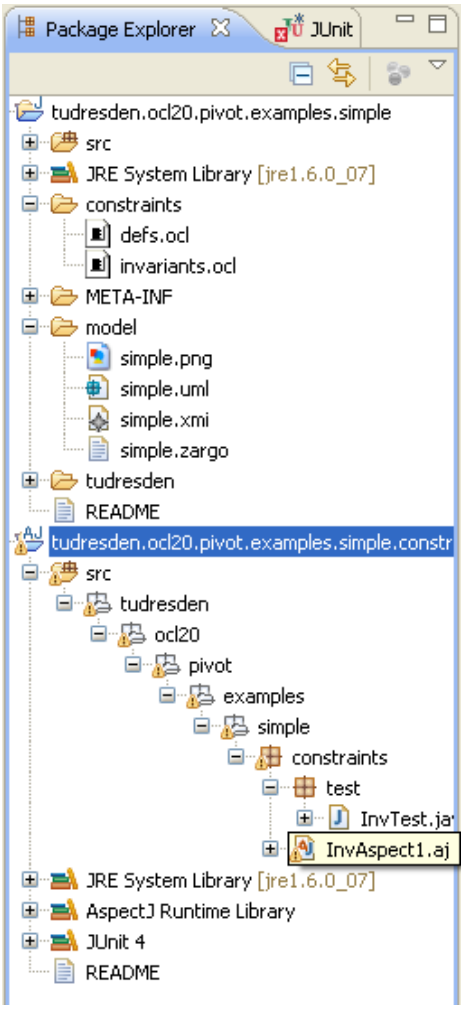

Figure 11: The package explorer containing the new generated aspect file.

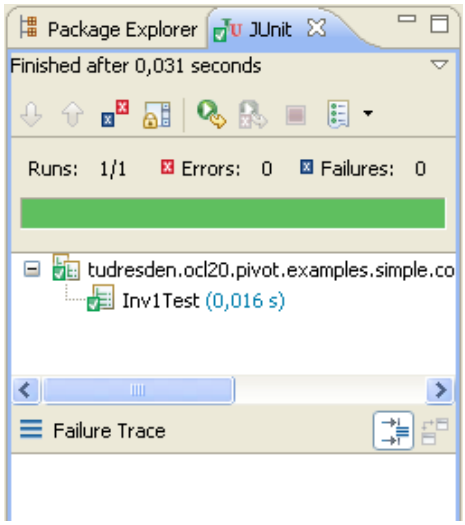

Figure 12: The successfully finished jUnit test case.

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

[Wil09a] 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

[Wil09b] Wilke, Claas: *Java Code Generation for Dresden OCL2 for Eclipse*. Großer Beleg, February 2009. – Available at http://dresden-ocl.sourceforge.net/publications.html#theses