TECHNISCHE
UNIVERSITÄT
DRESDEN

Faculty of Computer Science Institute of Software- and Multimedia-Technology, Software Technology Group

DRESDEN OCL2 FOR ECLIPSE

MANUAL FOR INSTALLATION, USE AND DEVELOPMENT

Claas Wilke

Last update on August 13, 2009

ABSTRACT

This document contains the documentation of Dresden OCL2 for Eclipse. In the first Chapter the general use of Dresden OCL2 for Eclipse and its installation will be explained. Afterwards, use cases like OCL interpretation and code generation will be explained.

Please be aware, that Dresden OCL2 for Eclipse is a project at the Software Technology Group, University of Dresden, that is realized and developed during student theses. Some parts have been developed as prototype. Thus, the toolkit is far from being complete. To report bugs and errors or request additional features or answers to specific questions visit our project site at Sourceforge [URL09f] or visit our toolkit website [URL09i].

The procedure described in this manual has been realized and tested with *Eclipse 3.5* [URL09d]. We recommend to use the *Eclipse Modeling Tools Edition* which contains all required plug-ins to run Dresden OCL2 for Eclipse. Otherwise you need to install at least the plug-ins enlisted in Table 2.

CONTENTS

I USING DRESDEN OCL2 FOR ECLIPSE

1 GETTING STARTED WITH DRESDEN OCL2 FOR ECLIPSE

Chapter written by Claas Wilke

This Chapter generally introduces into *Dresden OCL2 for Eclipse*. Dresden OCL2 for Eclipse is the last version of the *Dresden OCL Toolkit* and is based on a *Pivot Model*. The pivot model was developed by Matthias Bräuer and is described in his Großer Beleg (Minor Thesis) [Brä07]. Further information about the toolkit is available at the website of the Dresden OCL Toolkit [URL09i]. This Chapter starts with the installation of the needed *Eclipse* plug-ins to run Dresden OCL2 for Eclipse. Afterwards, it describes how to load a domain-specific model, an instance of such a model, and OCL constraints defined on such a model.

1.1 HOW TO INSTALL DRESDEN OCL2 FOR ECLIPSE

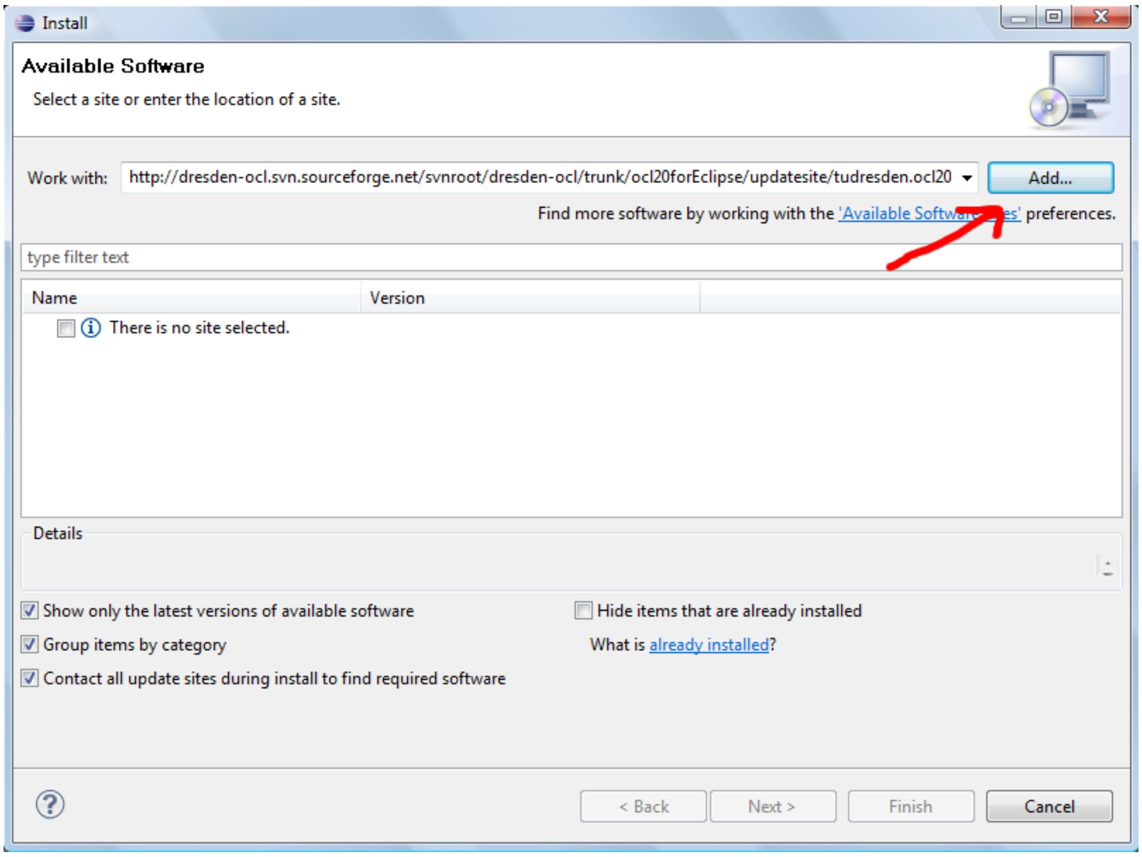
Four different possibilities exist to install Dresden OCL2 for Eclipse. (1) You may install the plug-ins using the update site available at [URL09h], (2) you may install the plug-ins using the binary distribution available at the SourceForge project site [URL09f], (3) you may run the the source code distribution available at the SourceForge project site [URL09f], or (4) you may checkout and run the source code distribution from the SVN available at [URL09g]. This Section will explain the possibilities (1) and (4).

1.1.1 Installing Dresden OCL2 for Eclipse using the Eclipse Update Site

To install Dresden OCL2 for Eclipse via the *Eclipse Update Site*, you have to start an *Eclipse* instance and select the menu option *Help -> Install New Software ...*. Enter the path

```
http://dresden-ocl.svn.sourceforge.net/svnroot/dresden-ocl/trunk/ocl20forEclipse/
updatesite/tudresden.ocl20.updatesite_1.2.0
```

and press the *Add...* button (see Figure 1.1). In the new opened window you can additionaly enter a name for the update site (see Figure 1.2).



Work with: http://dresden-ocl.svn.sourceforge.net/svnroot/dresden-ocl/trunk/ocl20forEclipse/updatesite/tudresden.ocl20

Find more software by working with the 'Available Software Sites' preferences.

There is no site selected.

☑ Show only the latest versions of available software

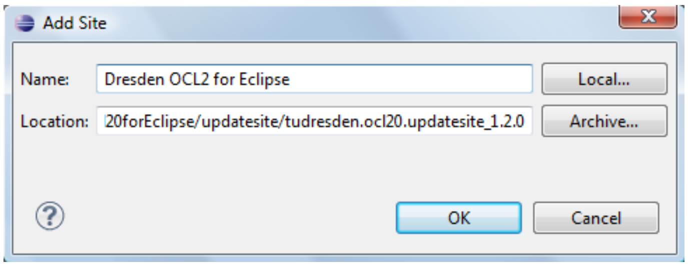
☐ Hide items that are already installed

☑ Group items by category

What is already installed?

☑ Contact all update sites during install to find required software

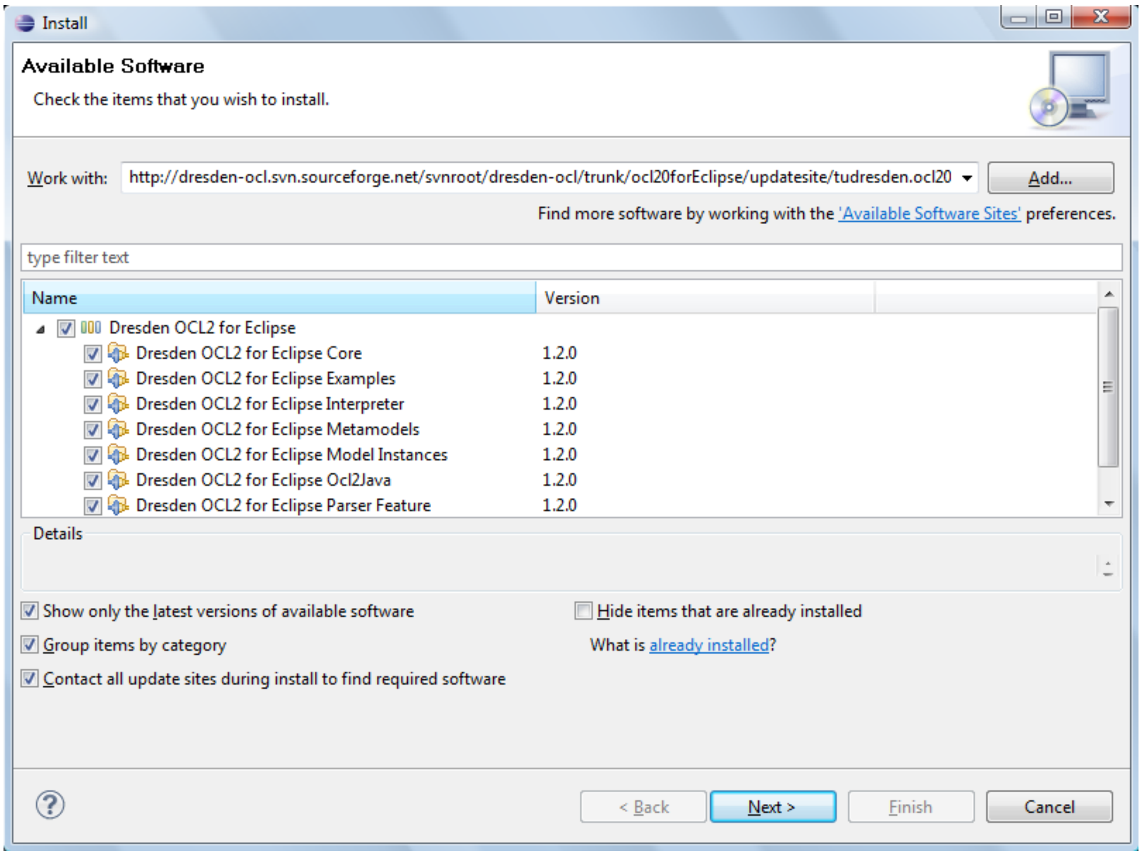
Figure 1.1: Adding an Eclipse Update Site (Step 1).



Name: Dresden OCL2 for Eclipse

Location: 20forEclipse/updatesite/tudresden.ocl20.updatesite_1.2.0

Figure 1.2: Adding an Eclipse Update Site (Step 2).


Install

Available Software

Check the items that you wish to install.

Work with: http://dresden-ocl.svn.sourceforge.net/svnroot/dresden-ocl/trunk/ocl20forEclipse/updatesite/tudresden.ocl20

Find more software by working with the 'Available Software Sites' preferences.

type filter text

Name	Version
☑ Dresden OCL2 for Eclipse	
☑ Dresden OCL2 for Eclipse Core	1.2.0
☑ Dresden OCL2 for Eclipse Examples	1.2.0
☑ Dresden OCL2 for Eclipse Interpreter	1.2.0
☑ Dresden OCL2 for Eclipse Metamodels	1.2.0
☑ Dresden OCL2 for Eclipse Model Instances	1.2.0
☑ Dresden OCL2 for Eclipse Ocl2Java	1.2.0
☑ Dresden OCL2 for Eclipse Parser Feature	1.2.0

Details

☑ Show only the latest versions of available software

☐ Hide items that are already installed

☑ Group items by category

What is already installed?

☑ Contact all update sites during install to find required software

Figure 1.3: Selecting features of Dresden OCL2 for Eclipse.

Now you can select the features of Dresden OCL2 for Eclipse which you want to install. Select them and click on the *Next >* button (see figure 1.3). An overview about all features of Dresden OCL2 for Eclipse can be found in table 3. Follow the wizard and agree with the user license. Then the Toolkit will be installed. Afterwards, you should restart your Eclipse application to finish the installation.

1.1.2 Importing Dresden OCL2 for Eclipse from the SVN

To use Dresden OCL2 for Eclipse by checking out the source code from the SVN you need to install a SVN client. In the following the use the *Eclipse Subversive* plug-in and at least one of the *SVN Connectors* available at [URL09e].

After installing Eclipse Subversive, a new *Eclispe Perspective* for access to SVN should exist. The perspective can be opened via the menu *Window > Open Perpective > Other... > SVN Repository Exploring*. In the view *SVN Repositories* you can add a new repository (see Figure 1.4) using the URL `https://dresden-ocl.svn.sourceforge.net/svnroot/dresden-ocl/`.

After pressing the *Finish* button the SVN repository root should the visible in the *SVN Repositories* view. To checkout the plug-ins, you now select them in the repository directory `trunk/ocl20forEclipse/eclipse` and use the *Checkout...* function in the context menu (see figure 1.5).

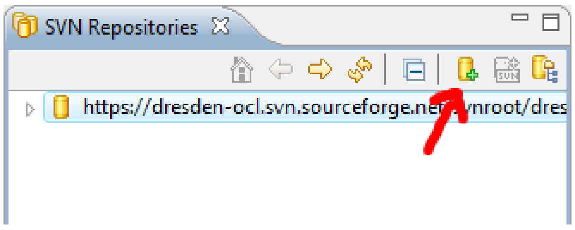


Figure 1.4: Adding a SVN repository.

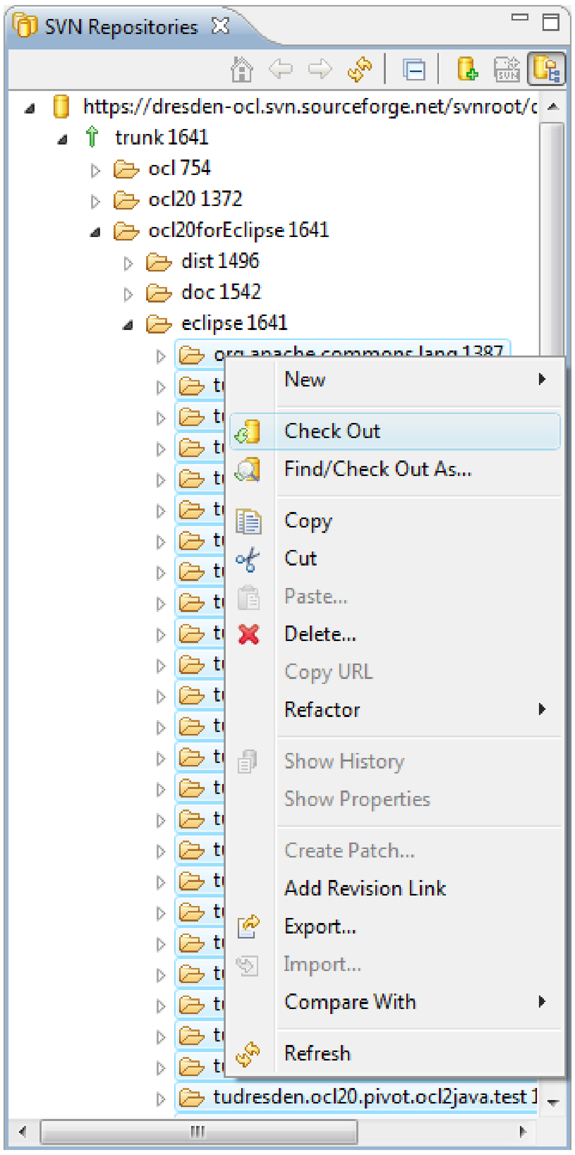


Figure 1.5: Checkout of the Dresden OCL2 Toolkit plug-in projects.

1.1.3 Which Plug-ins do I need at least?

An often asked question is "Which plug-ins are at least required to run Dresden OCL2 for Eclipse?" Well, the answer is: "That depends."

That depends on the things you want to do with Dresden OCL2 for Eclipse. Table 3 shows a list of the current plug-ins of Dresden OCL2 for Eclipse, that are related to different features. You should install at least the *Core* feature, at least one of the *Metamodels*, and the *Parser* feature. The *Interpreter* and *OCL2Java* features are only required if you want to interpret constraints or to generate code from constraints. If you import or interpret model instances, you need to install the *Model Instances* feature as well. The examples of the *Example* feature are only required to run the examples provided in the tutorials available at [URL09i]. We recommend to install all provided features.

1.1.4 Building the OCL2 Parser

If you decided to run Dresden OCL2 for Eclipse as source code plug-ins from an Eclipse workspace, you need to build the *OCL2 Parser* via an *Ant* build script. If you installed the Toolkit using the update site, you can skip this Subsection of the tutorial.

To build the OCL2 Parser select the file `build.xml` in the project `tudresden.ocl20.pivot.ocl2parser` and open the context menu via a right mouse click. Select the function *Run As ... > Ant Build ...* (see Figure 1.6).

A new window should open. Select in the sub menu *JRE* the check box *Run in the same JRE as the workspace* and click on the button *OK* (see figure 1.7). Afterwards the OCL2 Parser should be generated without errors. If an error like "Unable to find javac compiler." occurs, you might be trying to run the *Ant* script with a *Java Run-time Environment* instead of a *JDK* (For errors like) use the *Installed JREs...* button in the same window to select a JDK instead.

After executing the build script successfully you need to update the projects in your workspace. Update the project `tudresden.ocl20.pivot.oclparser` via context menu (*Refresh*, see figure 1.8).

Additionally you need to recompile all depending projects. Select the function *Project > Clean... > Clean all projects* in the Eclipse menu to clean all projects. Now all the projects should not contain any errors anymore and should be executable.

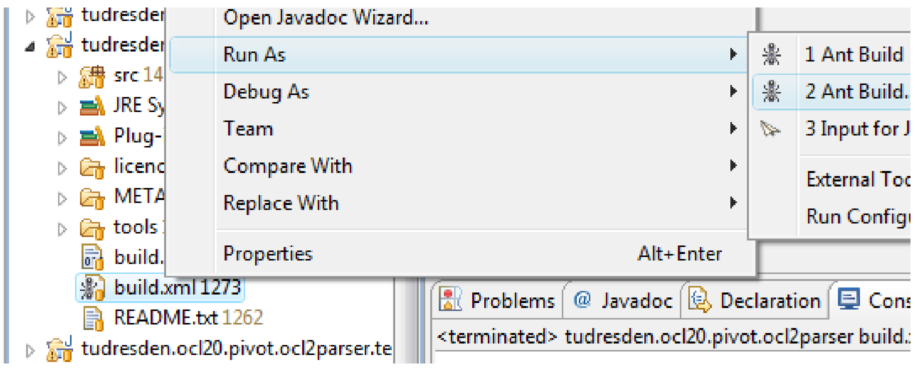
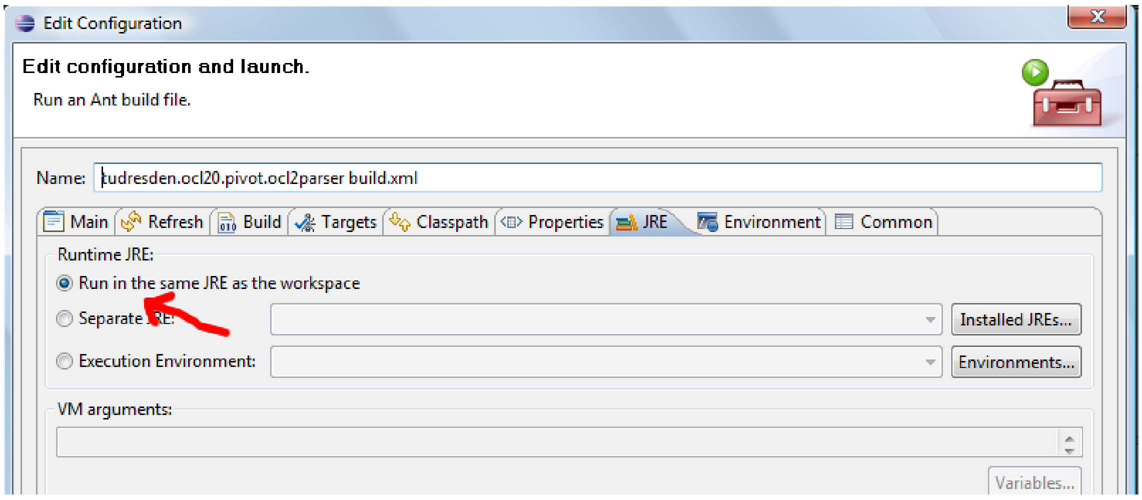


Figure 1.6: Executing the OCL2 parser build script.


Edit Configuration

Edit configuration and launch.

Run an Ant build file.

Name: tudresden.ocl20.pivot.ocl2parser build.xml

Runtime JRE:

(●) Run in the same JRE as the workspace

( ) Separate JRE:

( ) Execution Environment:

VM arguments:

Figure 1.7: Settings of the JRE for the Ant build script.

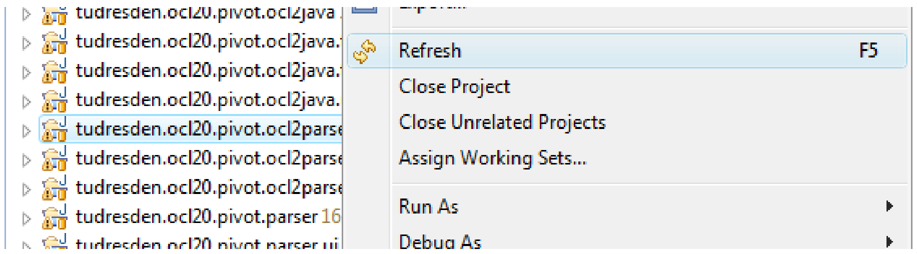

tudresden.ocl20.pivot.ocl2java
tudresden.ocl20.pivot.ocl2java.
tudresden.ocl20.pivot.ocl2java.
tudresden.ocl20.pivot.ocl2java.
tudresden.ocl20.pivot.ocl2parse
tudresden.ocl20.pivot.ocl2parse
tudresden.ocl20.pivot.ocl2parse
tudresden.ocl20.pivot.parser 16
Refresh F5
Close Project
Close Unrelated Projects
Assign Working Sets...
Run As
Debug As

Figure 1.8: Refreshing the project "tudresden.ocl20.pivot.oclparser".

1.2 LOADING MODELS, MODEL INSTANCES AND OCL CONSTRAINTS

If you installed the Dresden OCL2 for Eclipse using the update site, you can execute the toolkit by re-starting your Eclipse distribution. If you imported the Toolkit as source code plug-ins into an Eclipse workspace, you have to start a new Eclipse instance. You can start a new instance via the menu *Run > Run As > Eclipse Application*. If the menu *Eclipse Application* is not available or disabled you need to select one of the plug-ins of the toolkit first.

1.2.1 The Simple Example

The use of Dresden OCL2 for Eclipse is explained using the *Simple Example* which is located in the plug-in package `tudresden.ocl20.pivot.examples.simple`. Figure 1.9 shows the class diagram of the Simple Example.

Dresden OCL2 for Eclipse provides more examples than the *Simple Example*. The different examples use different meta-models which is possible with the *Pivot Model* architecture of the Toolkit. An overview about all examples provided with *Dresden OCL2 for Eclipse* is listed in table 4. The Simple Example can be used with two different meta-models. These are *UML 2.0* (based on *Eclipse MDT UML2*) and *EMF Ecore*.

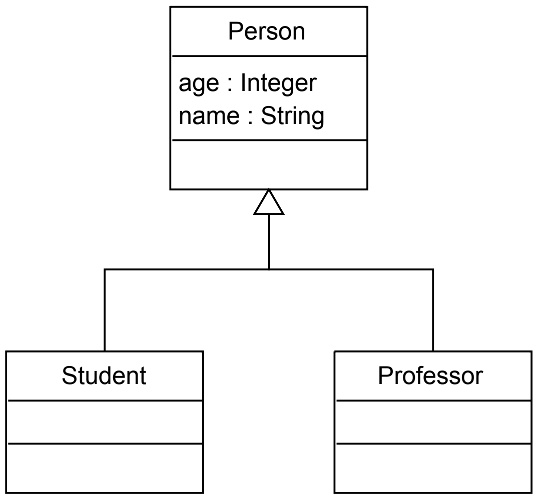


Figure 1.9: The class diagram described by the simple example model.

1.2.2 Loading a Domain-Specific Model

After starting Eclipse you have to load a model into the toolkit. If the plug-ins of Dresden OCL2 for Eclipse have been installed using the update site, the Simple Example plug-in have to be imported into the *Workspace* first. Create a new Java project into your workspace and select the *Import Wizard General > Archive File*. In the following window select the *plugins* directory in your Eclipse root folder, select the archive `tudresden.ocl20.pivot.examples.simple_1.0.0.jar` and click the *Finish* button.

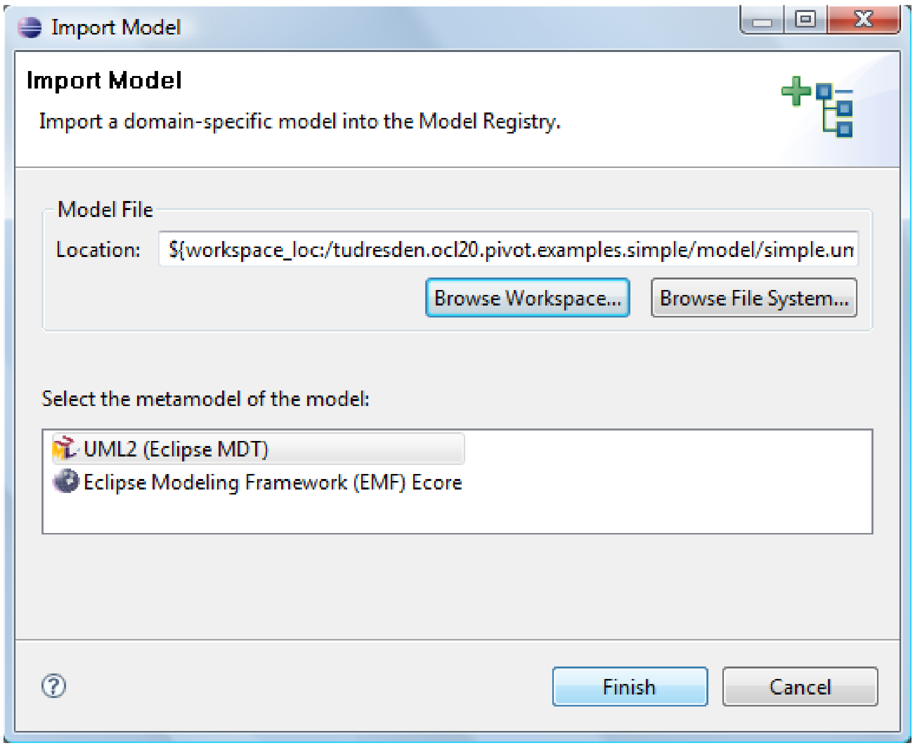
Now you can load a model. Select the menu option *Dresden OCL2 > Load Model*. In the opened wizard you have to select a model file and a meta-model for the model (see Figure 1.10). Click the button *Browse Workspace...* and select the file `model/simple.uml` inside the Simple Example Project. Then select the meta-model UML2 and press the button *Finish*.

Figure 1.11 shows the loaded Simple Example model, which uses UML2 as its meta-model. Via the menu button of the *Model Browser* (the little triangle in the right top corner) you can switch between different models (see Figure 1.12).

1.2.3 Loading a Model Instance

After loading a model, you can load a *model instance* using another wizard. Use the menu option *Dresden OCL2 > Load Model Instance*. In the opened wizard you have to select a model instance (in this tutorial we used the file `bin/tudresden/ocl20/pivot/examples/ModelProviderClass.class` of the Simple Example (see Figure 1.13). Besides the model instance resource you have to select a model for which the model instance shall be loaded and the type of model instance you want to load (we want to load a *Java Instance*).

Figure 1.14 shows the loaded model instance of the Simple Example model. Like in the model browser you can switch between different model instances. Note that the model instance browser only shows the model instances of the model actually selected in the model browser. By switching the domain specific model, you also switch the pool of model instances available in the model instance browser.



Import Model

Import Model

Import a domain-specific model into the Model Registry.

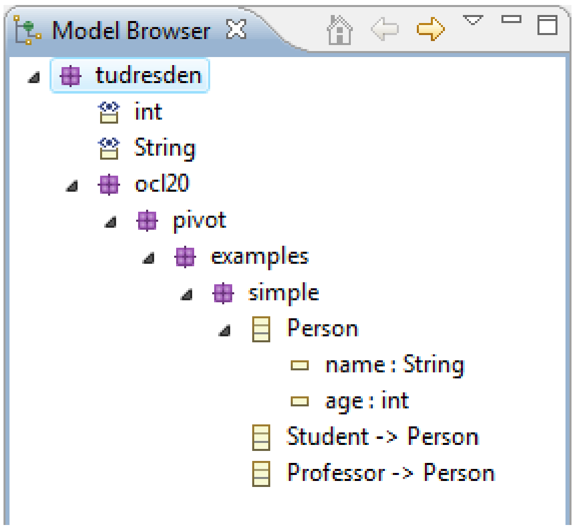
Model File

Location: ${workspace_loc:/tudresden.ocl20.pivot.examples.simple/model/simple.un

Select the metamodel of the model:

- UML2 (Eclipse MDT)
- Eclipse Modeling Framework (EMF) Ecore

Figure 1.10: Loading a domain specific model.



Model Browser

```
tudresden
    int
    String
    ocl20
        pivot
            examples
                simple
                    Person
                        name : String
                        age : int
                    Student -> Person
                    Professor -> Person
```

Figure 1.11: The loaded Simple Example model in the model browser.

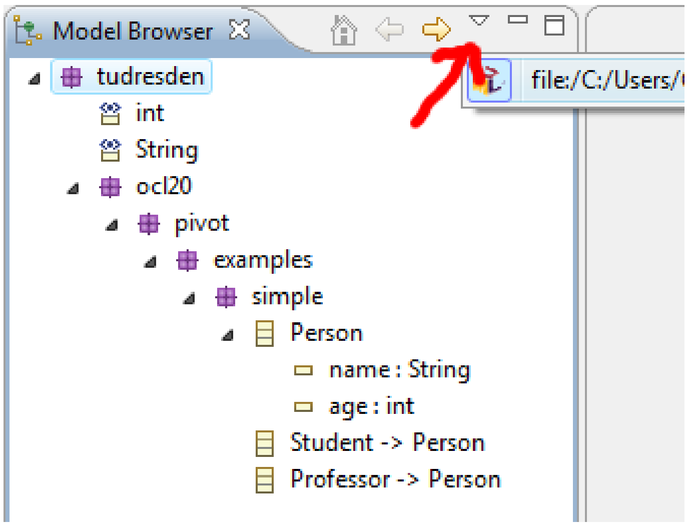
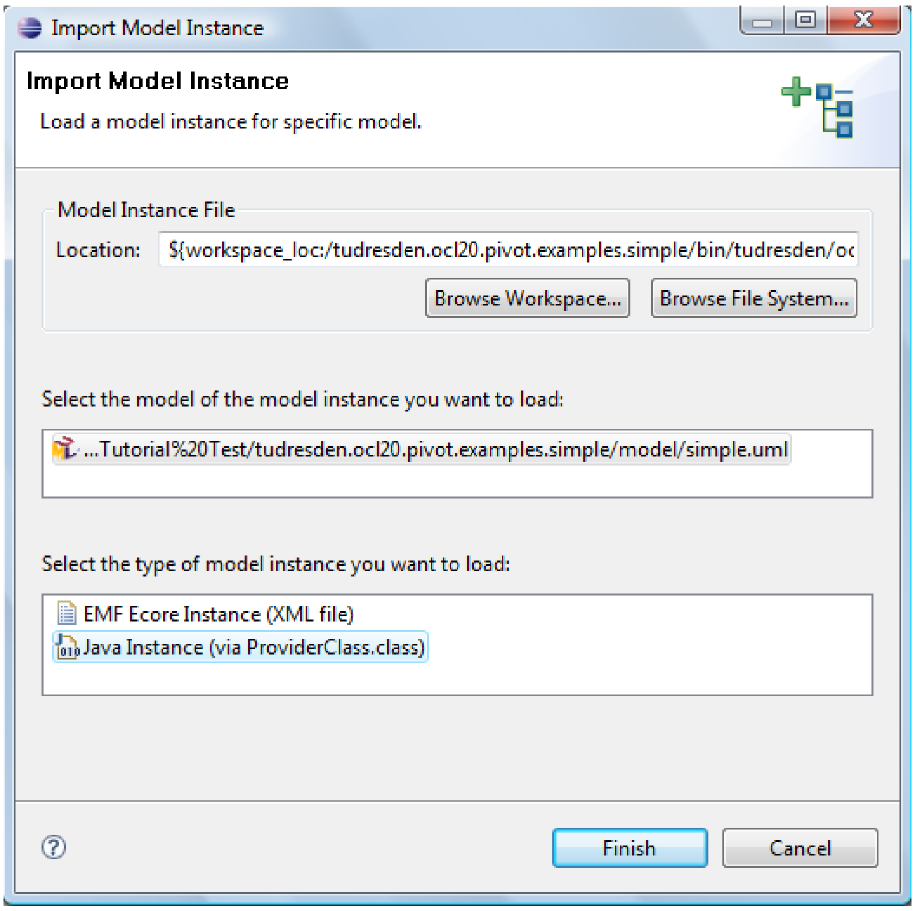


Figure 1.12: You can switch between different models using the little triangle.


Import Model Instance

Import Model Instance

Load a model instance for specific model.

Model Instance File

Location: ${workspace_loc:/tudresden.ocl20.pivot.examples.simple/bin/tudresden/oc

Browse Workspace... Browse File System...

Select the model of the model instance you want to load:

...Tutorial%20Test/tudresden.ocl20.pivot.examples.simple/model/simple.uml

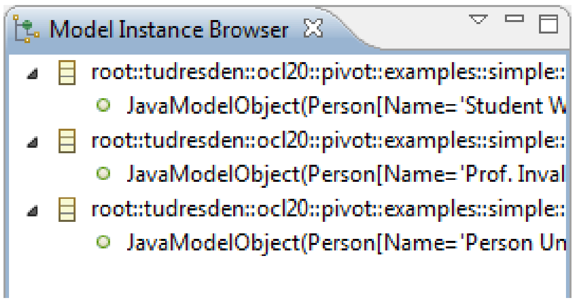
Select the type of model instance you want to load:

EMF Ecore Instance (XML file)

Java Instance (via ProviderClass.class)

Finish Cancel

Figure 1.13: Loading a simple model instance.


Model Instance Browser

root::tudresden::ocl20::pivot::examples::simple::
JavaModelObject(Person[Name='Student W
root::tudresden::ocl20::pivot::examples::simple::
JavaModelObject(Person[Name='Prof. Inval
root::tudresden::ocl20::pivot::examples::simple::
JavaModelObject(Person[Name='Person Un

Figure 1.14: A simple model instance in the Model Instance Browser.

1.2.4 Parsing OCL expressions

Before you can work with OCL constraints you have to load them like the domain-specific model and the model instance into the toolkit. Use the menu option *Dresden OCL2 > OCL Expressions* and select an OCL file. In this tutorial we used the OCL file `constraints/invariants.ocl` of the Simple Example. (see Figure 1.15). The constraints of the file `constraints/invariants.ocl` are shown in listing 1.1.

The expressions of the selected OCL file are loaded into the actually selected model. Figure 1.15 shows the `Model Browser` containing the model and the parsed expressions.

1.3 SUMMARY

This Chapter described how to use Dresden OCL2 for Eclipse. It explained how to install the toolkit's plug-ins. Afterwards, the loading of domain-specific models, model instances and OCL constraints into the toolkit has been explained.

Now, the imported models can be used to use the tools provided with Dresden OCL2 for Eclipse. For example you can use the *OCL2 Interpreter* of Dresden OCL2 for Eclipse to interpret OCL constraints for a given model and model instance (as explained in Chapter 2) or you can use the *OCL22Java Code Generator* to generate *AspectJ* code for a loaded model and OCL constraints (as explained in Chapter 3).

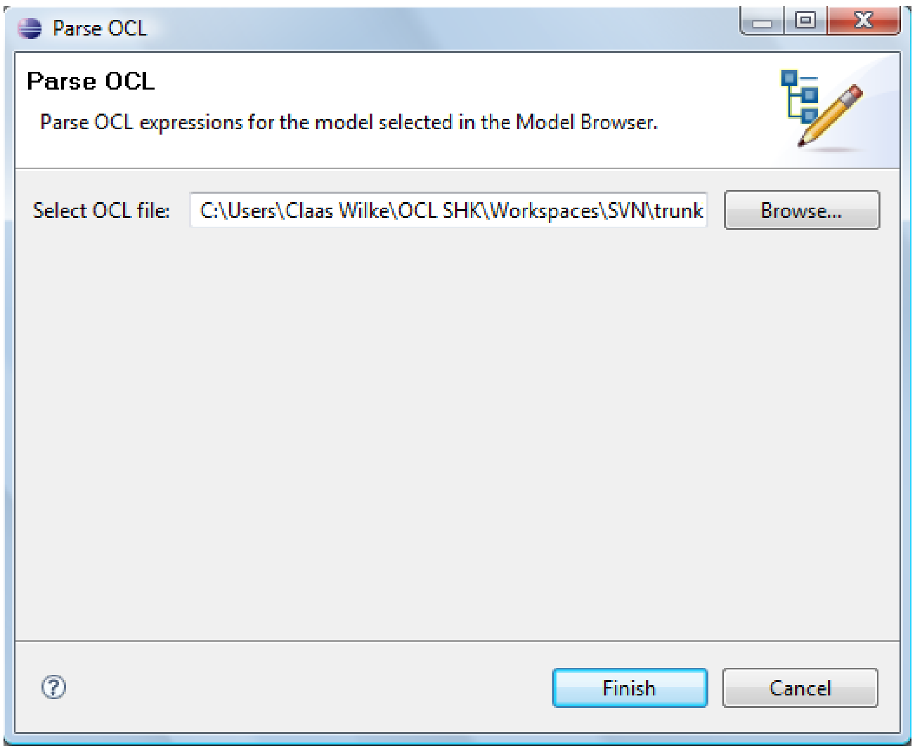
```
package tudresden::ocl20::pivot::examples::simple

-- The age of Person can't be negative.
context Person
inv: age >= 0

-- Students should be 16 or older.
context Student
inv: age > 16

-- Proffesors should be at least 30.
context Professor
inv: not (age < 30)

endpackage
```
Listing 1.1: The invariants of the simple examples.

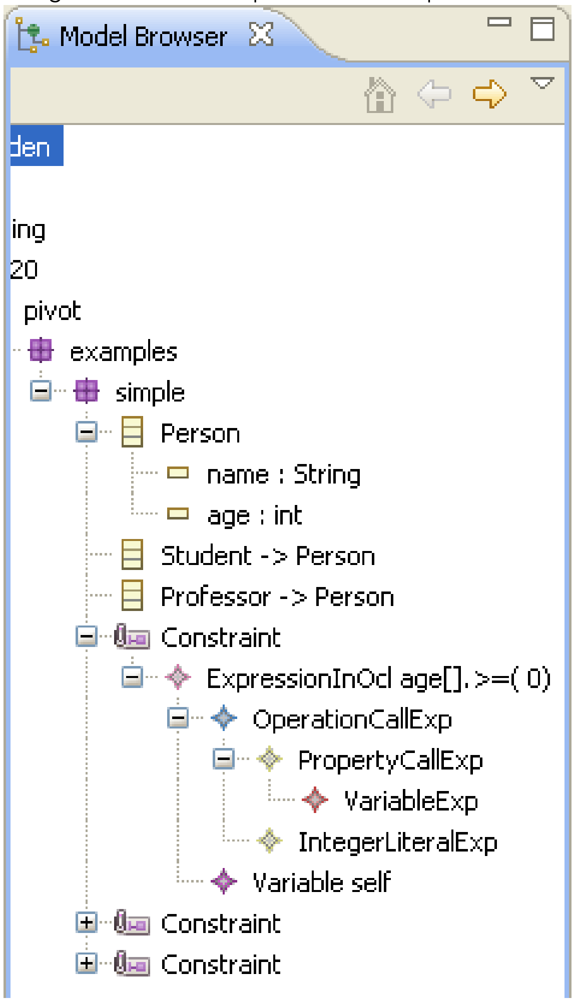


Parse OCL

Parse OCL expressions for the model selected in the Model Browser.

Select OCL file: C:\Users\Claas Wilke\OCL SHK\Workspaces\SVN\trunk

Figure 1.15: The import of OCL expressions.



```
den

ing
20
 pivot
  examples
   simple
    Person
     name : String
     age : int
    Student -> Person
    Professor -> Person
    Constraint
     ExpressionInOcl age[].>=( 0)
      OperationCallExp
       PropertyCallExp
        VariableExp
       IntegerLiteralExp
      Variable self
    Constraint
    Constraint
```

Figure 1.16: Parsed expressions and the model in the Model Browser.

2 OCL CONSTRAINT INTERPRETATION

Chapter written by Claas Wilke

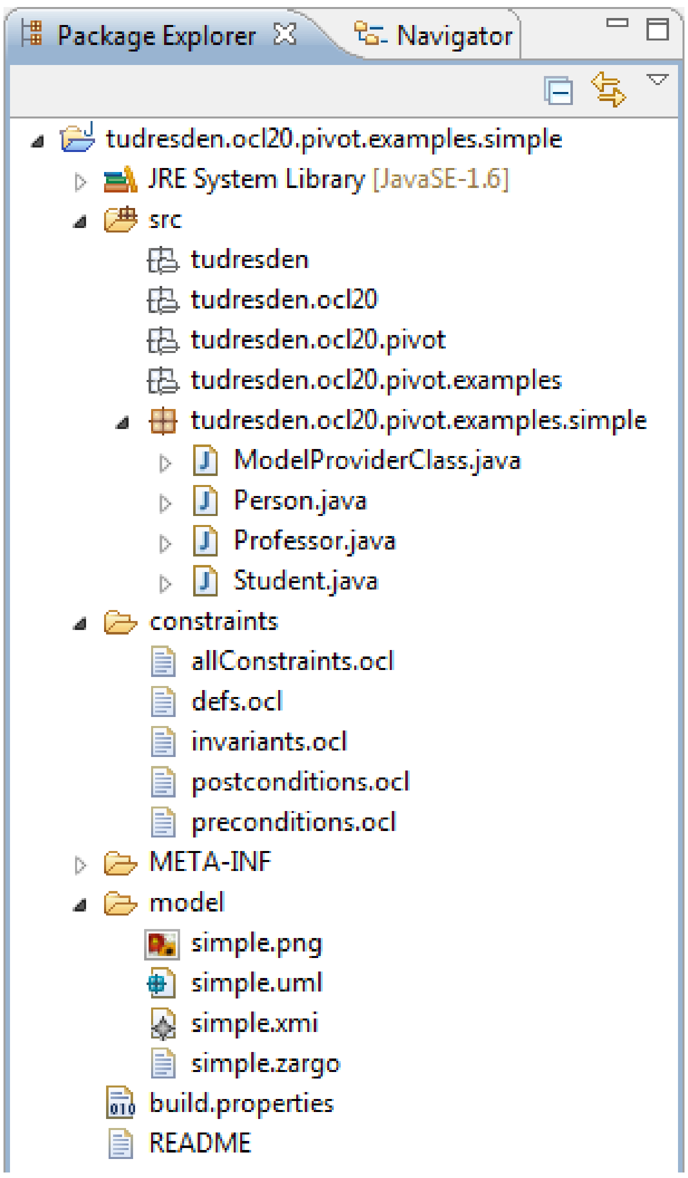
This Chapter describes how the OCL2 Interpreter provided with Dresden OCL2 for Eclipse can be used. How to install and run *Dresden OCL2 for Eclipse* and how to load models and OCL constraints will not be explained in this Chapter. This Chapter assumes that the user is familiar with such basic uses of the toolkit. A general introduction into Dresden OCL2 for Eclipse can be found in Chapter 1.

2.1 THE SIMPLE EXAMPLE

This Chapter 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 4. An introduction into the Simple Example can be found in Section 1.2.1. 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 where 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*). Figure 2.1 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.ocl`). Listing 2.1 shows the constraints defined in the constraint file.



```
tudresden.ocl20.pivot.examples.simple
  JRE System Library [JavaSE-1.6]
  src
    tudresden
    tudresden.ocl20
    tudresden.ocl20.pivot
    tudresden.ocl20.pivot.examples
    tudresden.ocl20.pivot.examples.simple
      ModelProviderClass.java
      Person.java
      Professor.java
      Student.java
  constraints
    allConstraints.ocl
    defs.ocl
    invariants.ocl
    postconditions.ocl
    preconditions.ocl
  META-INF
  model
    simple.png
    simple.uml
    simple.xmi
    simple.zargo
  build.properties
  README
```

Figure 2.1: The package explorer containing the project which is needed to run this tutorial.

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

-- Students should be 16 or older.
context Student
inv: age > 16

-- Proffesors should be at least 30.
context Professor
inv: not (age < 30)

-- Returns the age of a Person.
context Person
def: getAge(): Integer = age

-- Before returning the age, the age must be defined.
context Person::getAge()
pre: not age.oclIsUndefined()

-- The result of getAge must equal to the age of a Person.
context Person::getAge()
post: result = age
```

Listing 2.1: The constraints contained in the constraint file.

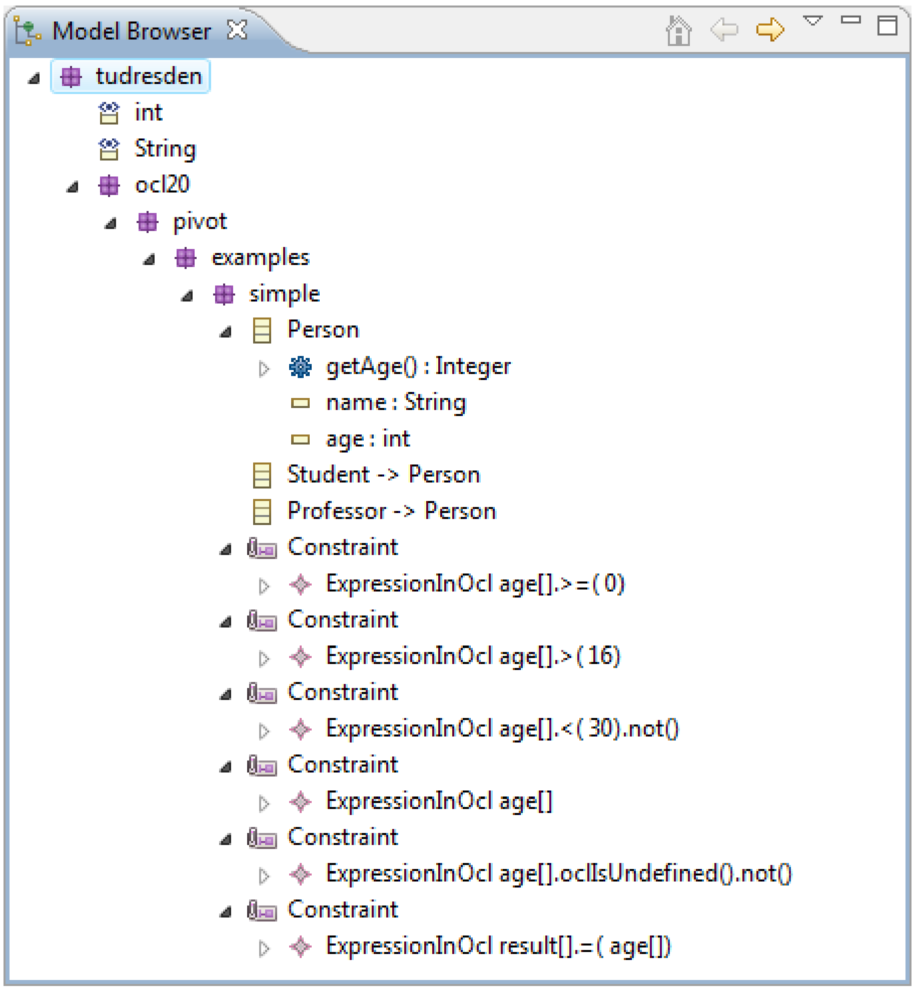
First, the constraint file defines three simple invariants that denote, that the `age` of every `Person` must always zero or greater than zero. Furthermore, the `age` of every `Student` must be greater than 16 and the `age` of every `Professor` does not have to be lesser than 30.

In addition to that the constraint file contains a definition constraint that 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`.

2.2 PREPARATION OF THE INTERPRETATION

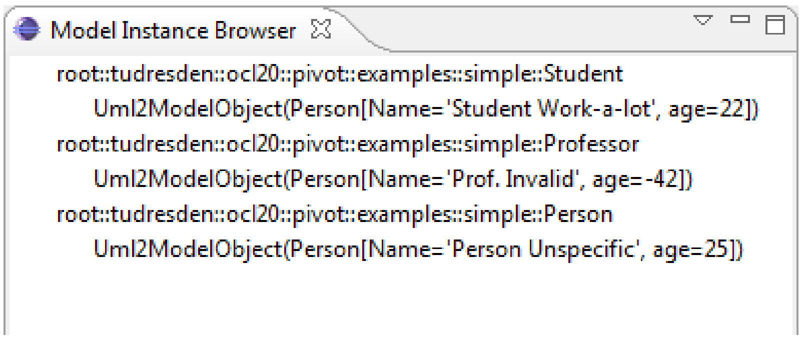
To prepare the interpretation we have to import the model `model/simple.uml` for which we want to interpret constraints into the *Model Browser*. We use the import wizard for domain-specific models of the toolkit to import the model. This procedure is explained in Section 1.2.3. Furthermore, we have 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 2.2 and the *Model Instance Browser* should look like shown in Figure 2.3.

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.



```
tudresden
    int
    String
    ocl20
        pivot
            examples
                simple
                    Person
                        getAge() : Integer
                        name : String
                        age : int
                    Student -> Person
                    Professor -> Person
                    Constraint
                        ExpressionInOcl age[].>=(0)
                    Constraint
                        ExpressionInOcl age[].>(16)
                    Constraint
                        ExpressionInOcl age[].<(30).not()
                    Constraint
                        ExpressionInOcl age[]
                    Constraint
                        ExpressionInOcl age[].oclIsUndefined().not()
                    Constraint
                        ExpressionInOcl result[].=(age[])
```

Figure 2.2: The model browser containing the simple model and its constraints.



```
root::tudresden::ocl20::pivot::examples::simple::Student
    Uml2ModelObject(Person[Name='Student Work-a-lot', age=22])
root::tudresden::ocl20::pivot::examples::simple::Professor
    Uml2ModelObject(Person[Name='Prof. Invalid', age=-42])
root::tudresden::ocl20::pivot::examples::simple::Person
    Uml2ModelObject(Person[Name='Person Unspecific', age=25])
```

Figure 2.3: The model instance browser containing the simple model instance.

2.3 OCL 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 2.4).

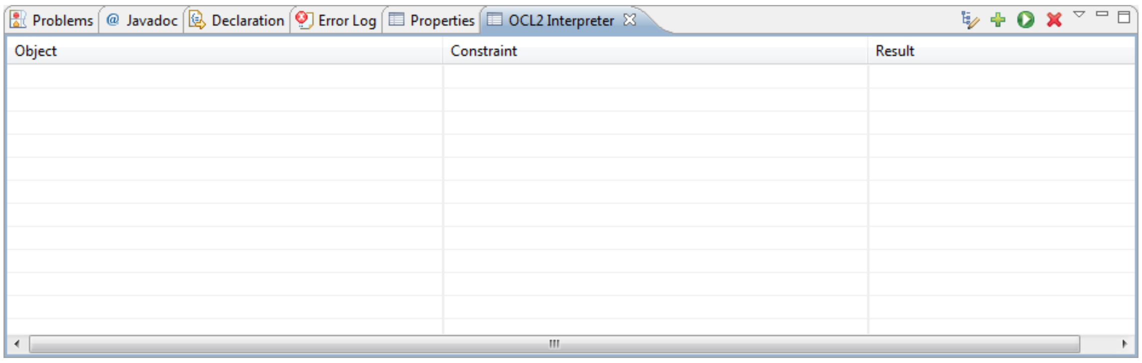


Figure 2.4: The OCL2 Interpreter containing no results.

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 2.5. With the first button (from left to right) constraints can be prepared for interpretation. The second button can be used 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 *OCL2 Interpreter View*. The functionality of the buttons will be explained below.

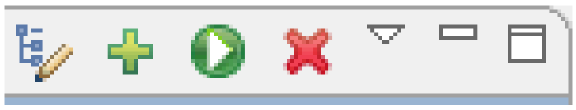


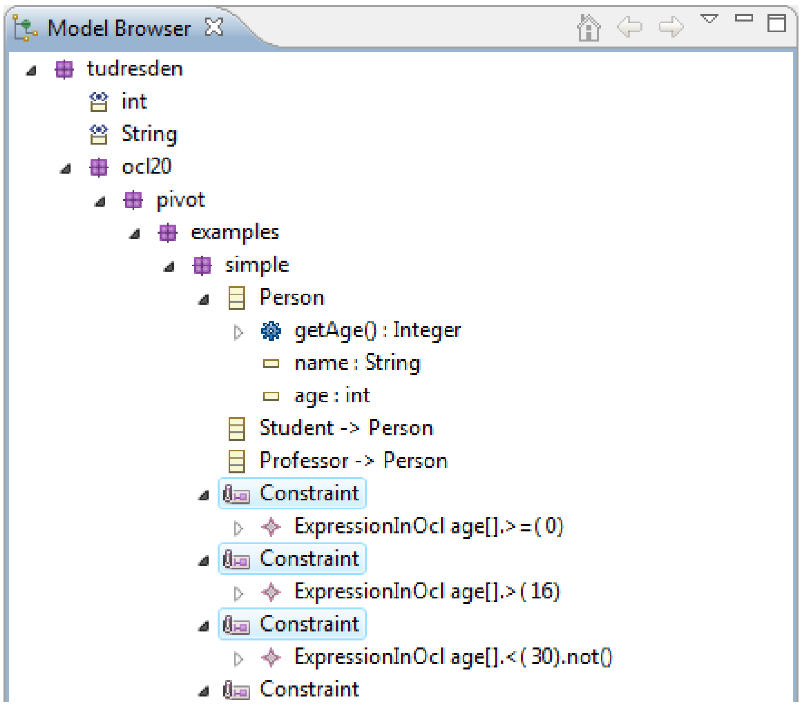
Figure 2.5: The buttons to control the OCL2 Interpreter.

2.3.1 Interpretation of Constraints

To interpret constraints, we simple 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 2.6) and click the *Interpret* button. The result of the interpretation is now shown in the *OCL2 Interpreter View* (see Figure 2.7).

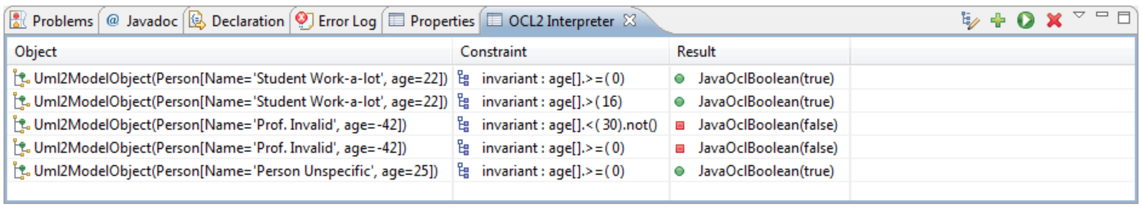
The invariant `age >= 0` has been interpreted for all three model objects. The results for the `Person` and the `Student` instances are `true` because their `age` is greater than zero. The result for the `Professor` instance is `false` because its `age` is `-42`.

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`.



```
tudresden
    int
    String
    ocl20
        pivot
            examples
                simple
                    Person
                        getAge() : Integer
                        name : String
                        age : int
                    Student -> Person
                    Professor -> Person
                    Constraint
                        ExpressionInOcl age[].>=(0)
                    Constraint
                        ExpressionInOcl age[].>(16)
                    Constraint
                        ExpressionInOcl age[].<(30).not()
                    Constraint
```

Figure 2.6: 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 2.7: The results of the three age invariants for all model instances.

2.3.2 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 2.8) and click the *Prepare* button (the first button from the left).

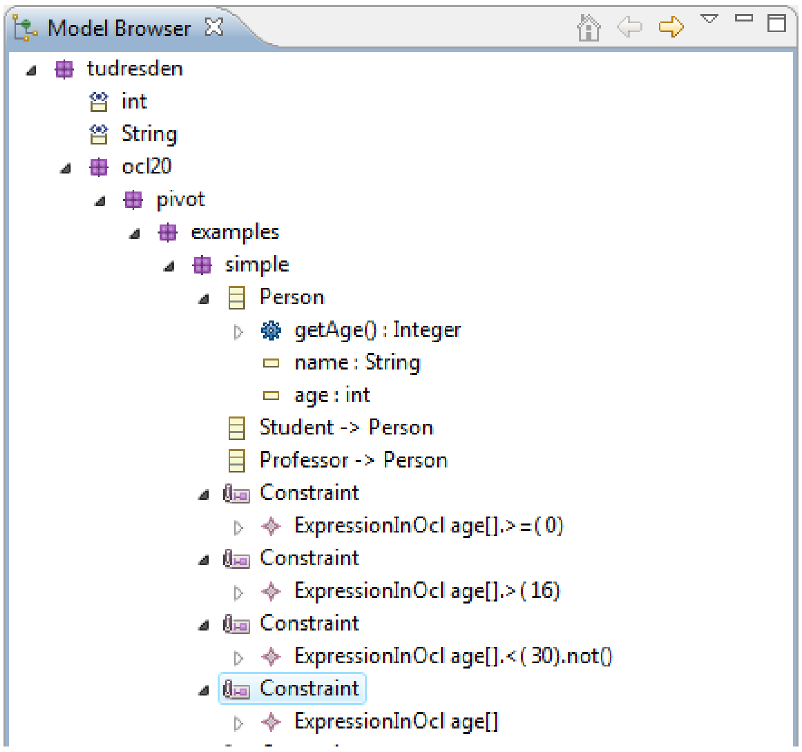
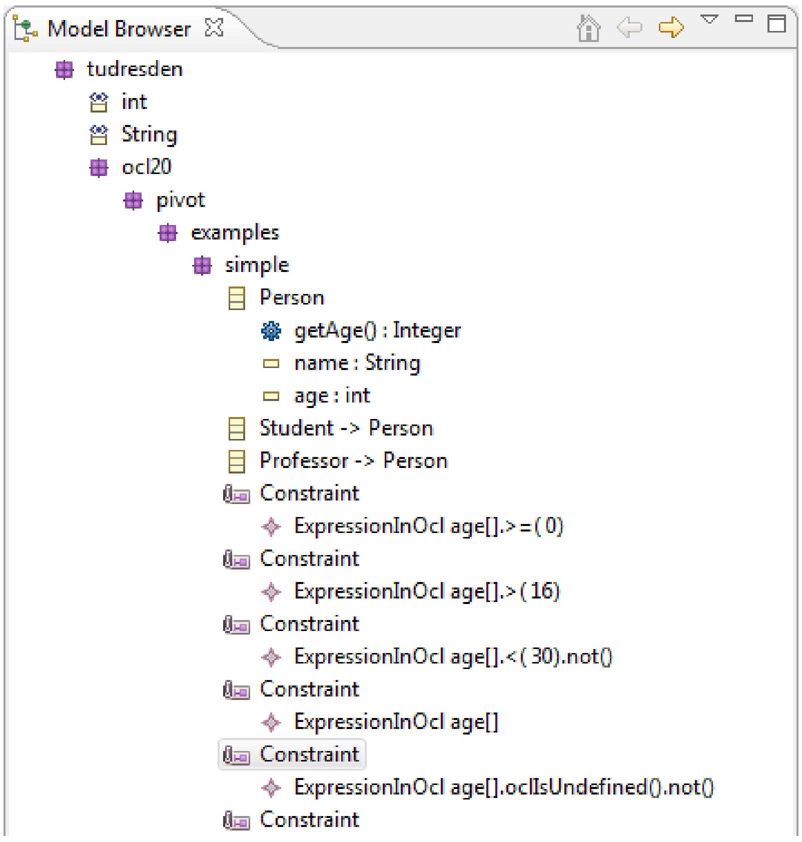

```
Model Browser
tudresden
    int
    String
  ocl20
    pivot
      examples
        simple
          Person
            getAge() : Integer
            name : String
            age : int
          Student -> Person
          Professor -> Person
          Constraint
            ExpressionInOcl age[].>=(0)
          Constraint
            ExpressionInOcl age[].>(16)
          Constraint
            ExpressionInOcl age[].<(30).not()
          Constraint
            ExpressionInOcl age[]
```

Figure 2.8: 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.

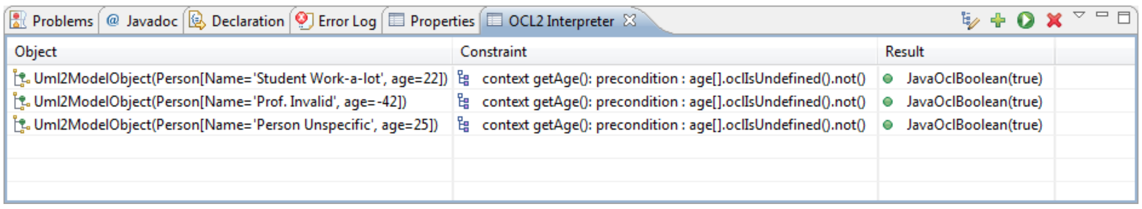
We select the constraint in the *Model Browser* (see Figure 2.9) and click the *Interpret* button. The result of the interpretation is shown in figure 2.10.

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



```
Model Browser
  tudresden
    int
    String
    ocl20
      pivot
        examples
          simple
            Person
              getAge() : Integer
              name : String
              age : int
            Student -> Person
            Professor -> Person
            Constraint
              ExpressionInOcl age[].>=(0)
            Constraint
              ExpressionInOcl age[].>(16)
            Constraint
              ExpressionInOcl age[].<(30).not()
            Constraint
              ExpressionInOcl age[]
            Constraint
              ExpressionInOcl age[].oclIsUndefined().not()
            Constraint
```

Figure 2.9: 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)

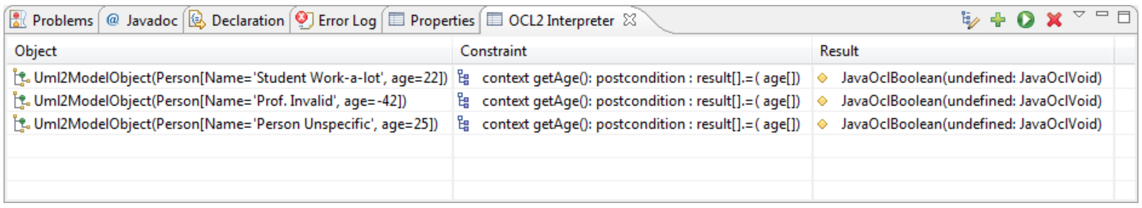
Figure 2.10: The results of the precondition for all model instances.

2.3.3 Adding Variables to the Environment

By preparing the definition constraint we added some information to the *Interpretation Environment* that 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.

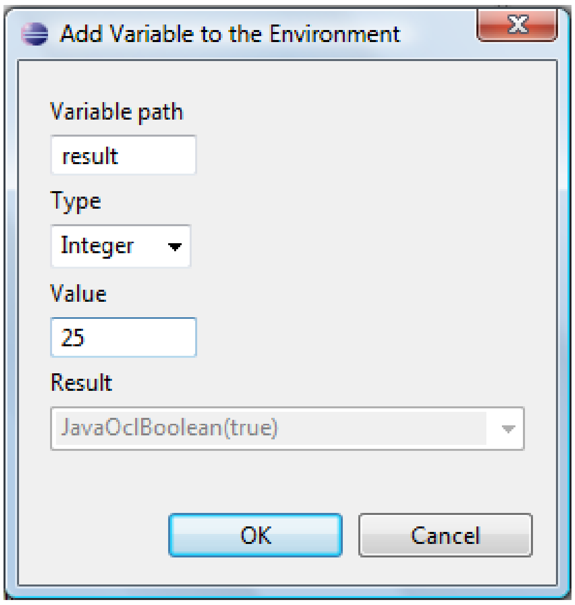
If we interpret the postcondition constraint (the sixth and last constraint in the *Model Browser*) without setting the `result` variable, the constraint results in a `undefined` result for all three model instances (see Figure 2.11).



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 2.11: The results of the postcondition without preparing the result variable.

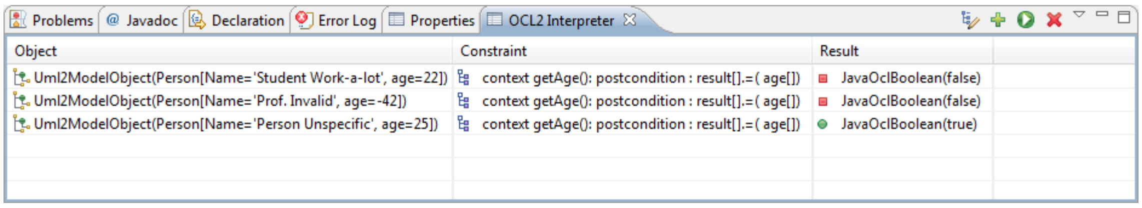
To prepare the variable we click on the button to add new variables to the Interpreter Environment (the second button from the left) and a new window opens which we can use to specify new variables. We enter the name `result`, select the variable type `Integer` and enter the value 25. Then we press the *OK* button (see Figure refpic:interpret:interpret09. The result variable has now been added to the Interpreter Environment.



Variable path: result
Type: Integer
Value: 25
Result: JavaOclBoolean(true)

Figure 2.12: The window to add new variables to the environment.

Now we can interpret the postcondition again. The result is shown in Figure 2.13. 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 2.13: The results of the postcondition with result variable preparation.

2.4 SUMMARY

This Chapter described how OCL 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.

3 ASPECTJ CODE GENERATION

Chapter written by Claas Wilke

This Chapter describes how the Java Code Generator *OCL22Java* provided with Dresden OCL2 for Eclipse can be used. A general introduction into Dresden OCL2 for Eclipse can be found in Chapter 1. A detailed documentation of the development of OCL22Java can be found in the Minor Thesis (Großer Beleg) of Claas Wilke [Wil09].

In addition to the general Eclipse installation the *AspectJ Development Tools (AJDT)* are required to execute the code generated with OCL22Java. The AJDT plug-ins can be found at the AJDT website [URL09b].

3.1 CODE GENERATOR PREPARATION

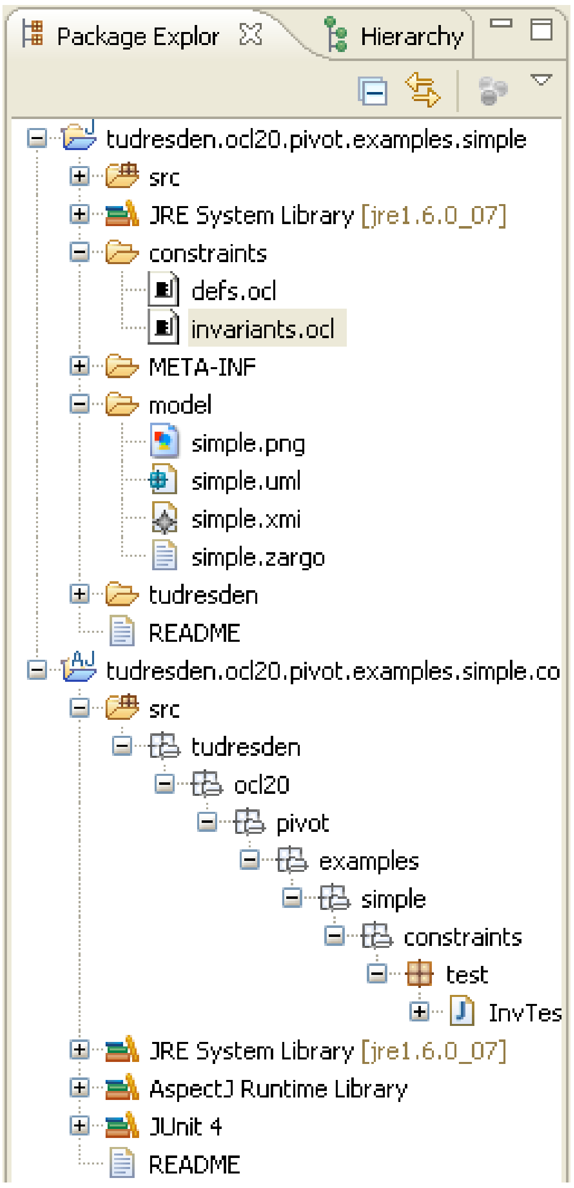
This Chapter uses the *Simple Example* which is provided with Dresden OCL2 for Eclipse and has been introduced in Subsection 1.2.1 To import the Simple Example into our Eclipse workspace we have to create a new Java project into our Workspace (here called `tudresden.ocl20.pivot.examples.simple`) and use the import wizard *General -> Archive File* to import the example provided as a JAR archive. In the following window we select the directory were the JAR file is located (eventually the `plugins` or `dropins` directory inside the Eclipse root folder). 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 AJDT first). Figure 3.1 shows the *Package Explorer* containing both imported projects.

After importing the second plug-in, we have to add the JUnit4 library to the project's build path. Right click on the project in the *Package Explorer* and select the menu item *Properties* (see Figure 3.2). In the new opened window select the sub-menu *Java Build Path -> Libraries* and click the button *Add Library...* (see Figure 3.3). In the following window select *JUnit*, click *Next*, select *JUnit 4* and click *Finish*. Click *OK* to close the project's properties. The project should not contain any compile errors anymore.

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 in Subsection 1.2.1 (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 3.1 shows one invariant that is contained in the constraint file for which we want to generate code. The invariant declares, 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 that checks, whether or not the mentioned constraint is enforced during run-time. 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 run-time exception is thrown, if the constraint is violated.



```
tudresden.ocl20.pivot.examples.simple
  src
  JRE System Library [jre1.6.0_07]
  constraints
    defs.ocl
    invariants.ocl
  META-INF
  model
    simple.png
    simple.uml
    simple.xmi
    simple.zargo
  tudresden
  README
tudresden.ocl20.pivot.examples.simple.co
  src
    tudresden
      ocl20
        pivot
          examples
            simple
              constraints
                test
                  InvTes
  JRE System Library [jre1.6.0_07]
  AspectJ Runtime Library
  JUnit 4
  README
```

Figure 3.1: The package explorer containing the two projects which are needed to run this tutorial.

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 3.4). To fulfill the test case we have to generate the ApsectJ code for the constraint which enforces the constraint's condition. How to generate such code will be explained in the following.

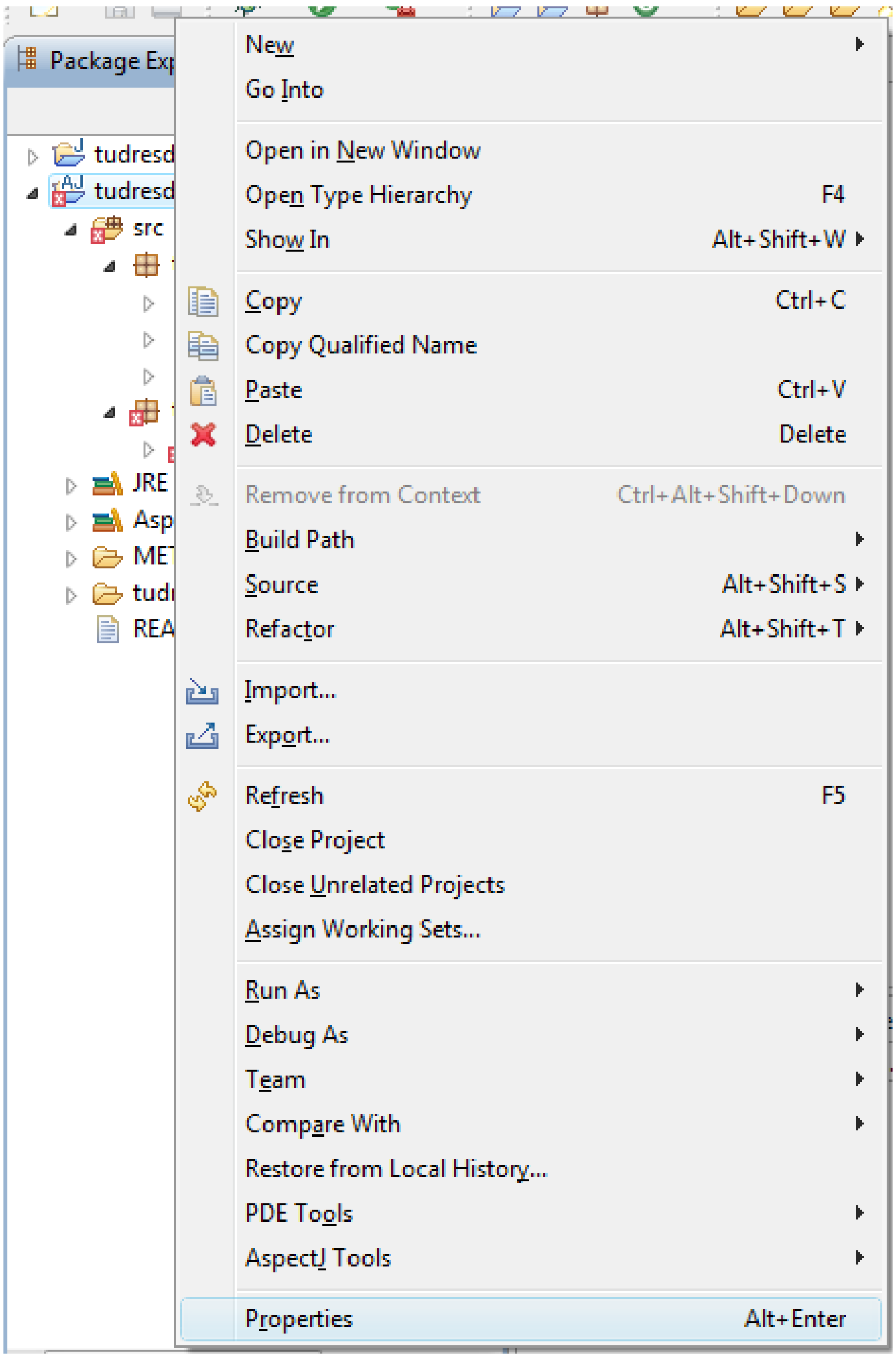
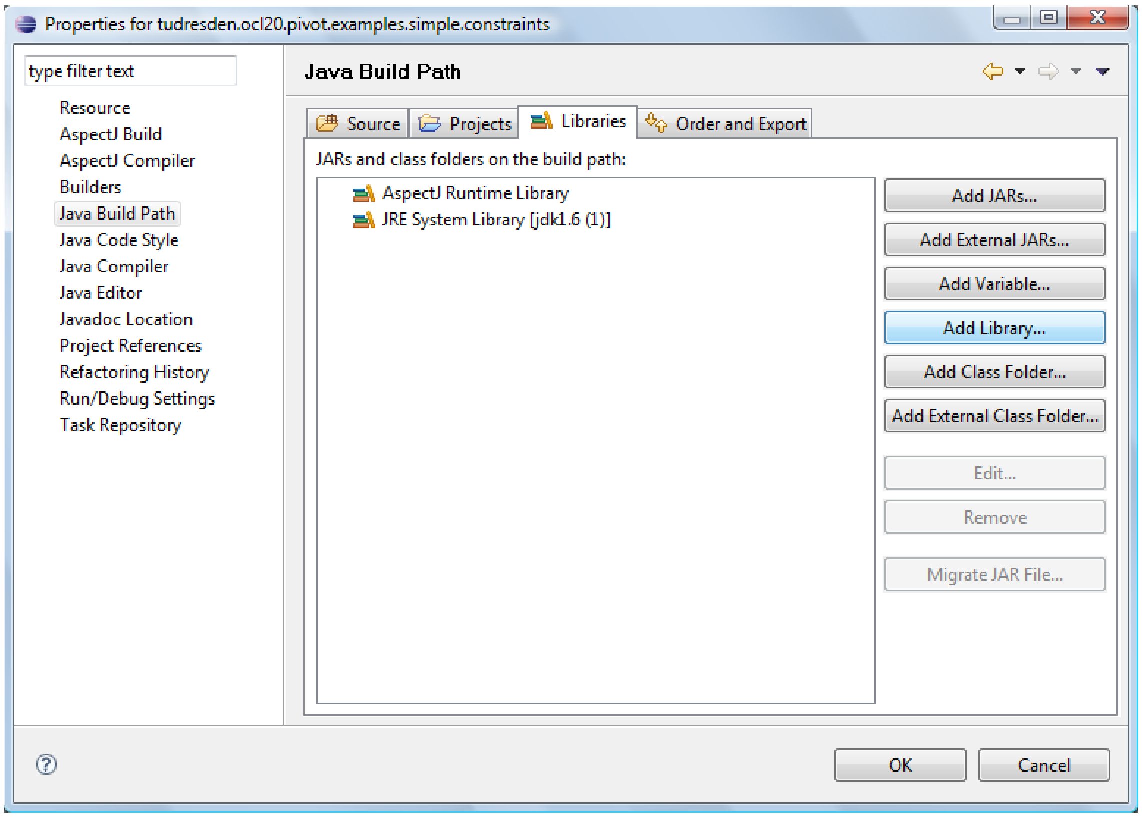


Figure 3.2: Selecting the properties settings.



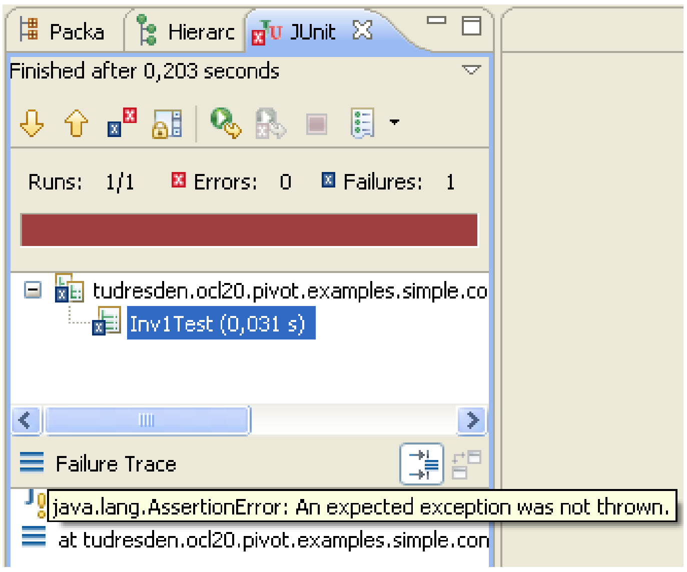
JARs and class folders on the build path:

AspectJ Runtime Library
JRE System Library [jdk1.6 (1)]

Figure 3.3: Adding a new library to the build path.

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

Listing 3.1: A simple invariant.



Finished after 0,203 seconds

Runs: 1/1 Errors: 0 Failures: 1

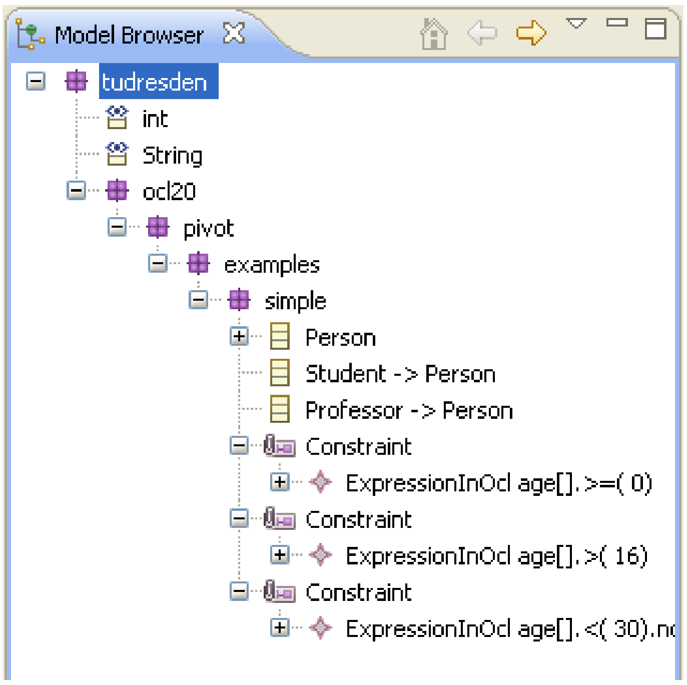
tudresden.ocl20.pivot.examples.simple.co
Inv1Test (0,031 s)

java.lang.AssertionError: An expected exception was not thrown.
at tudresden.ocl20.pivot.examples.simple.con

Figure 3.4: The result of the JUnit test case.

3.2 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 introduction in Chapter 1. Afterwards, we have to import the constraint file `constraints/invariant.ocl` which is done by an import wizard again. After the importation, the *Model Browser* should look like illustrated in Figure 3.5. Now we can start the code generation.



```
tudresden
  int
  String
  ocl20
    pivot
      examples
        simple
          Person
          Student -> Person
          Professor -> Person
          Constraint
            ExpressionInOcl age[].>=( 0)
          Constraint
            ExpressionInOcl age[].>( 16)
          Constraint
            ExpressionInOcl age[].<( 30).nc
```

Figure 3.5: The model browser containing the simple model and its constraints.

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

3.2.1 Selecting a Model

A wizard opens and we have to select a model for code generation (see Figure 3.6). We select the `simple.uml` model and click the *Next* button.

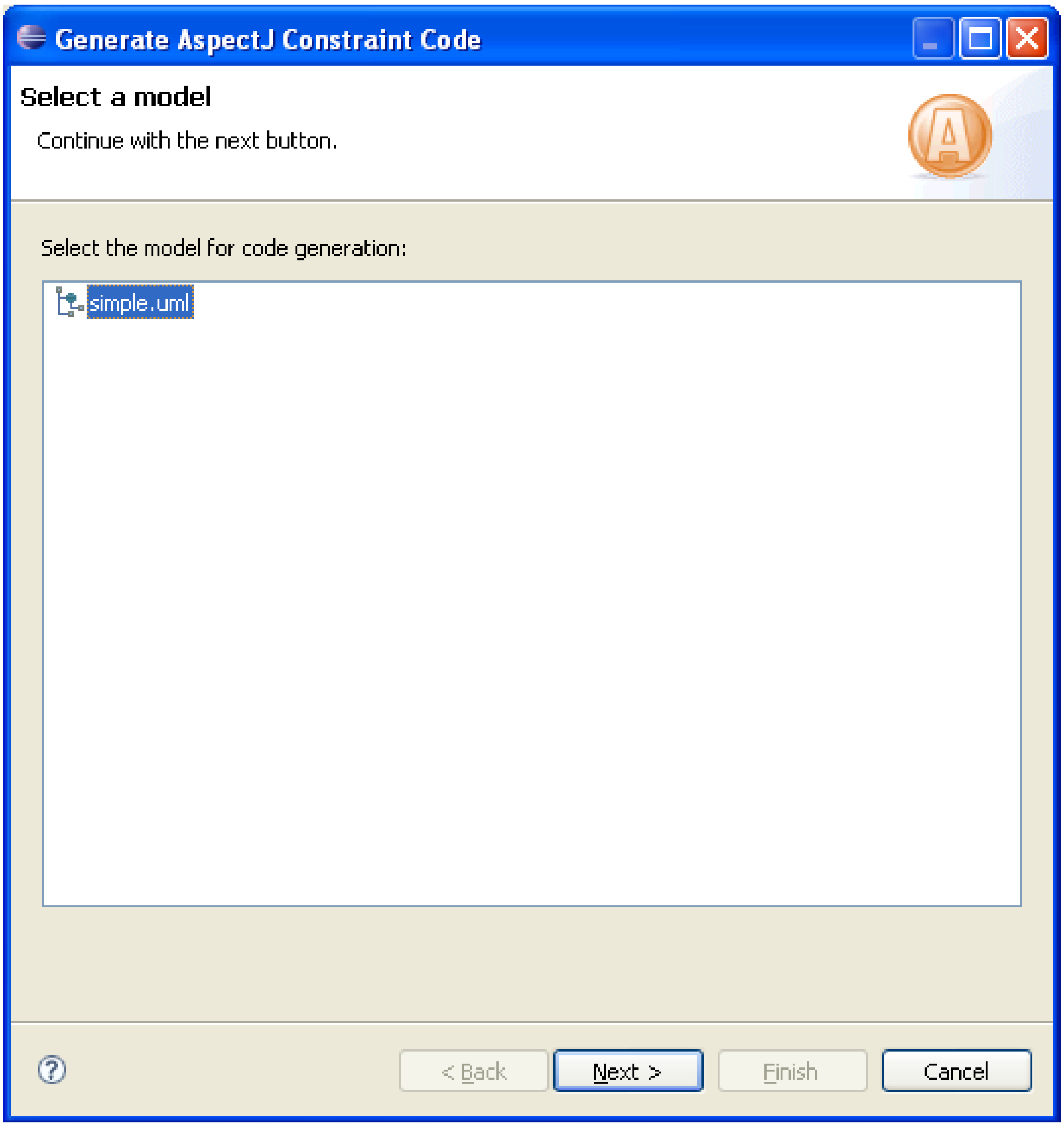
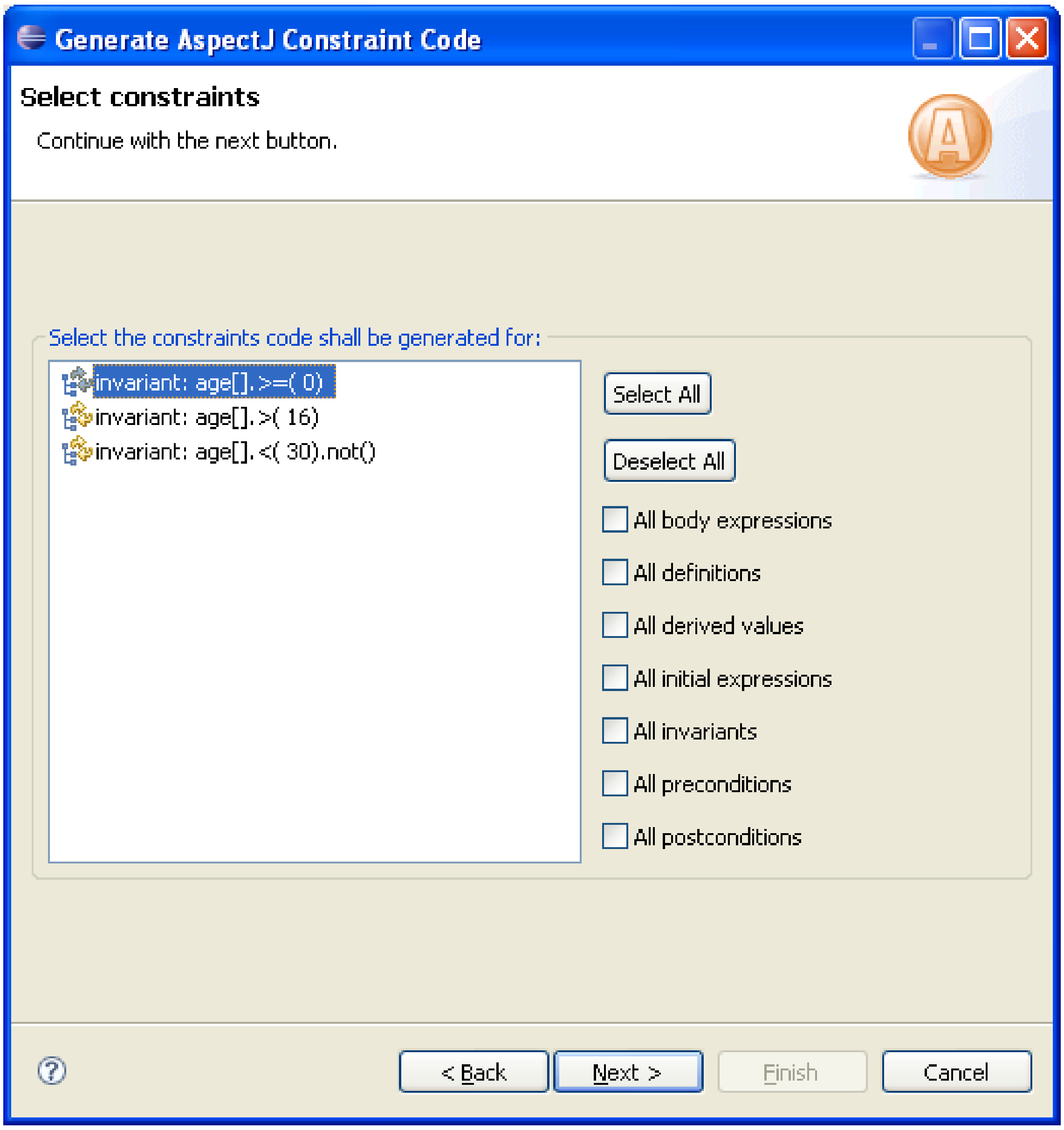

Generate AspectJ Constraint Code
Select a model
Continue with the next button.
Select the model for code generation:
simple.uml
< Back
Next >
Finish
Cancel

Figure 3.6: The first step: Selecting a model for code generation.

3.2.2 Selecting Constraints

As a second step we have to select the constraints for that we want to generate code. We only select the constraint that enforces that the `age` of any `Person` must be equal to or greater than zero and click the *Next* button (see Figure 3.7).



Generate AspectJ Constraint Code

Select constraints

Continue with the next button.

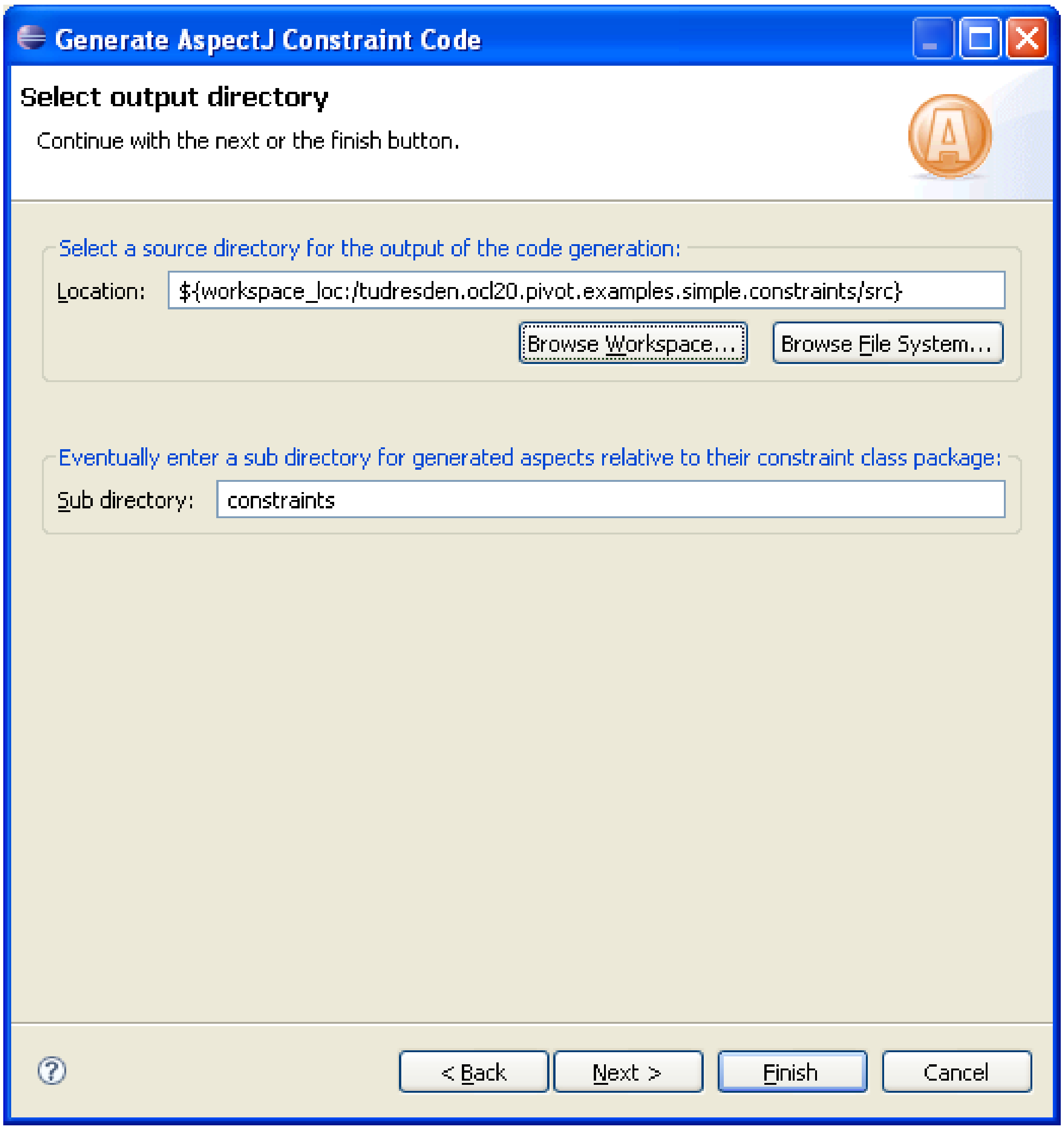
Select the constraints code shall be generated for:

- invariant: age[].>=(0)
- invariant: age[].>(16)
- invariant: age[].<(30).not()

Figure 3.7: The second step: Selecting constraints for code generation.

3.2.3 Selecting a Target Directory

Next, we have to select a target directory to that 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 3.8). 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 that the constraint code shall be generated relatively to the package of the constrained class. By default this is a sub directory called `constraints`. We don't want to change this setting and click the *Next* button.



Select output directory

Continue with the next or the finish button.

Select a source directory for the output of the code generation:

Location: ${workspace_loc:/tudresden.ocl20.pivot.examples.simple.constraints/src}

Eventually enter a sub directory for generated aspects relative to their constraint class package:

Sub directory: constraints

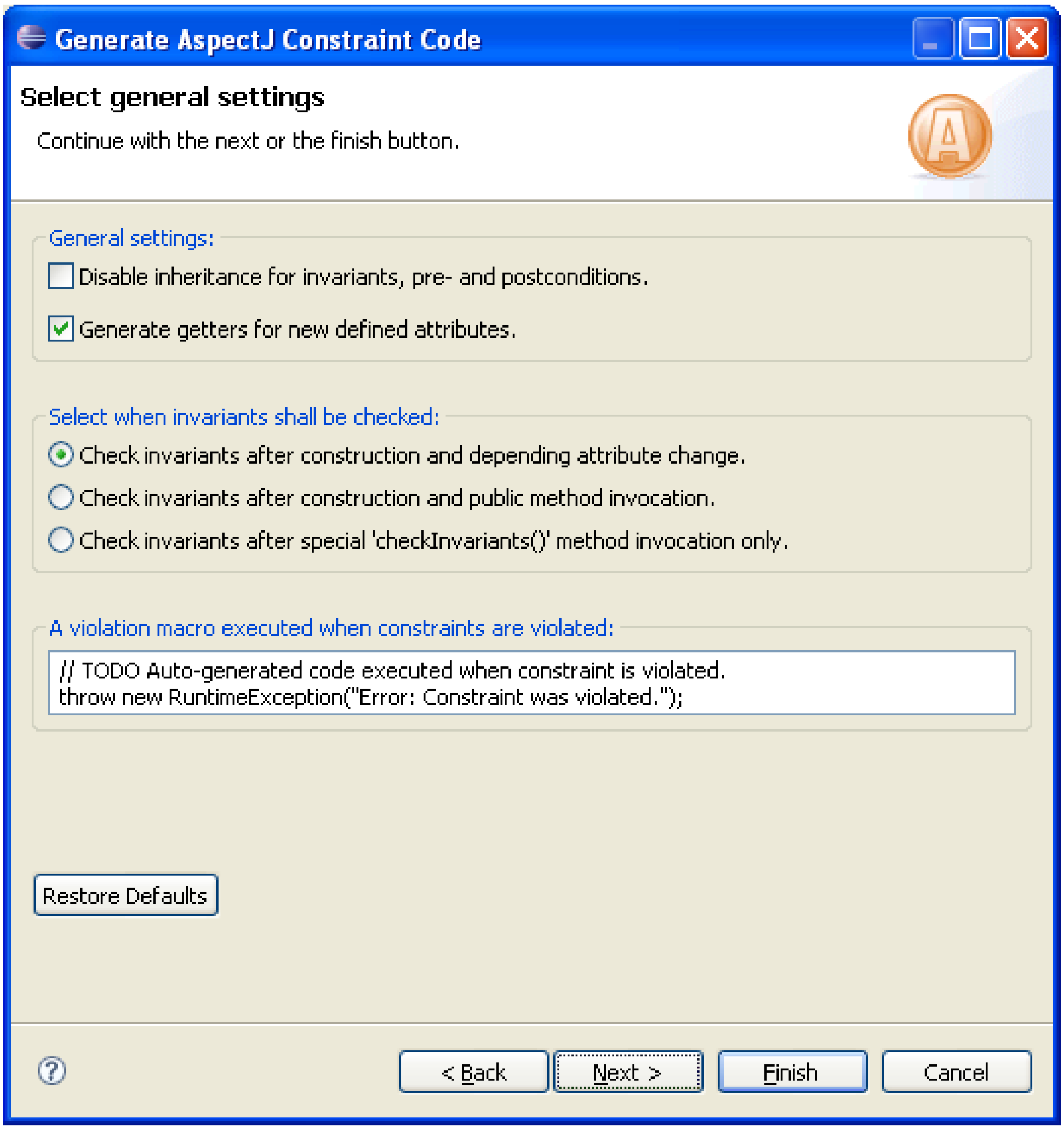
Figure 3.8: The third step: Selecting a target directory for the generated code.

3.2.4 Specifying General Settings

On the following page of the wizard we can specify general settings for the code generation (see Figure 3.9).

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 newly defined attributes of `def` constraints. More interesting is the possibility to select one of three provided strategies, when invariants shall be checked during runtime:

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*).


Generate AspectJ Constraint Code

Select general settings

Continue with the next or the finish button.

General settings:

- [ ] Disable inheritance for invariants, pre- and postconditions.
- [✓] Generate getters for new defined attributes.

Select when invariants shall be checked:

- ● Check invariants after construction and depending attribute change.
- ○ Check invariants after construction and public method invocation.
- ○ Check invariants after special 'checkInvariants()' method invocation only.

A violation macro executed when constraints are violated:

```
// TODO Auto-generated code executed when constraint is violated.
throw new RuntimeException("Error: Constraint was violated.");
```

Figure 3.9: The fourth step: General settings for the code generation.

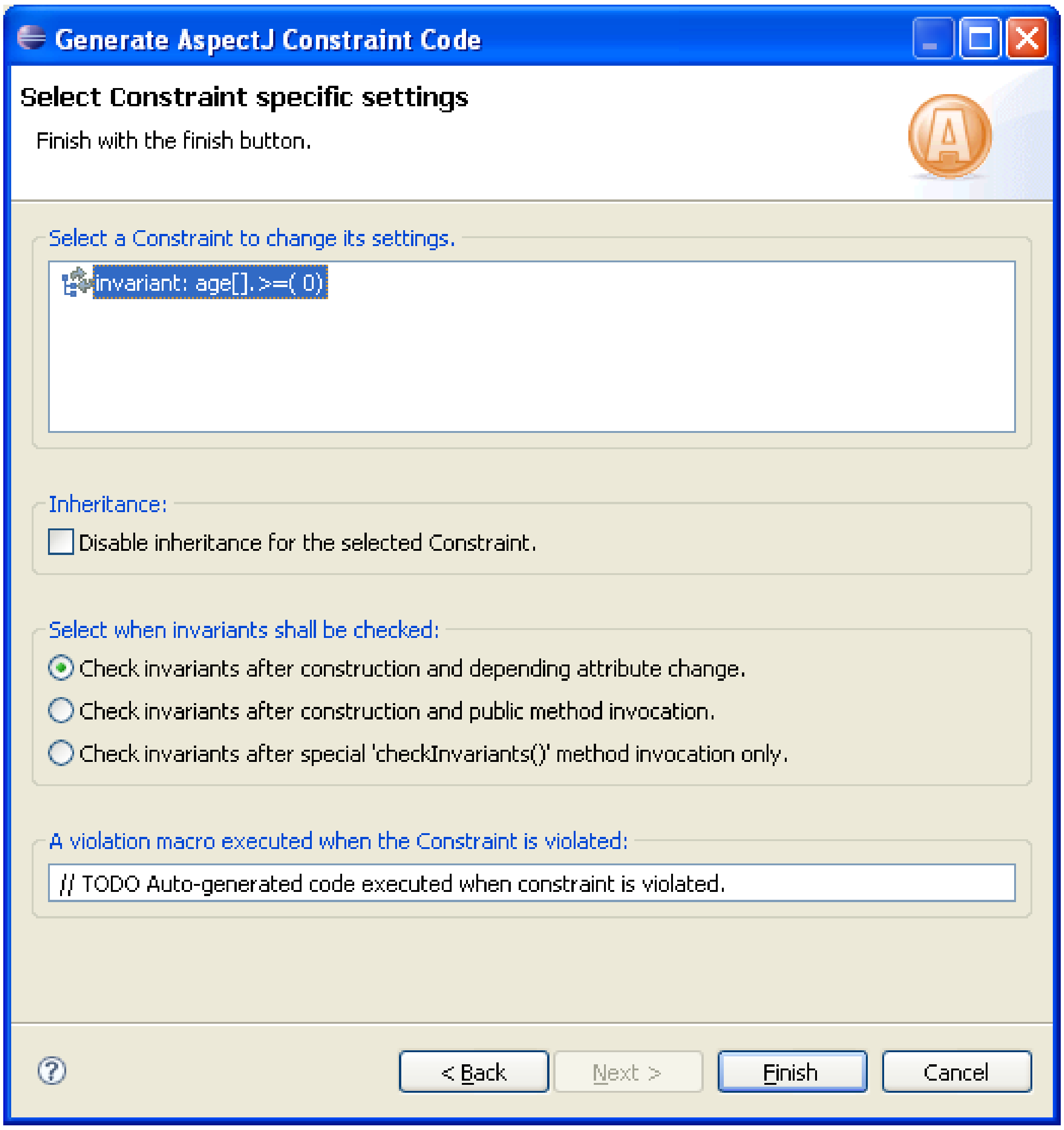
2. Invariants can be checked after construction of an object and before or after the execution of any public method of the constrained class (*Weak Verification*).
3. And finally, invariants can only be checked if the user calls a special method at runtime (*Transactional Verification*).

These 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 run-time exception. We also want to have a run-time exception thrown when our constraint is violated. Thus, we do not change the violation macro and continue with the *Next* button.

3.2.5 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 3.10). We don't want to adapt the settings, thus we can finish the wizard and start the code generation by clicking the *Finish* button.


Generate AspectJ Constraint Code

Select Constraint specific settings

Finish with the finish button.

Select a Constraint to change its settings.

invariant: age[].>=(0)

Inheritance:

☐ Disable inheritance for the selected Constraint.

Select when invariants shall be checked:

● Check invariants after construction and depending attribute change.

○ Check invariants after construction and public method invocation.

○ Check invariants after special 'checkInvariants()' method invocation only.

A violation macro executed when the Constraint is violated:

// TODO Auto-generated code executed when constraint is violated.

Figure 3.10: The fifth step: Constraint-specific settings for the code generation.

3.3 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.InvAspect01.aj` (see Figure 3.11). Now we can rerun our JUnit test case. The test case finishes successfully because the expected run-time exception is thrown (see Figure 3.12).

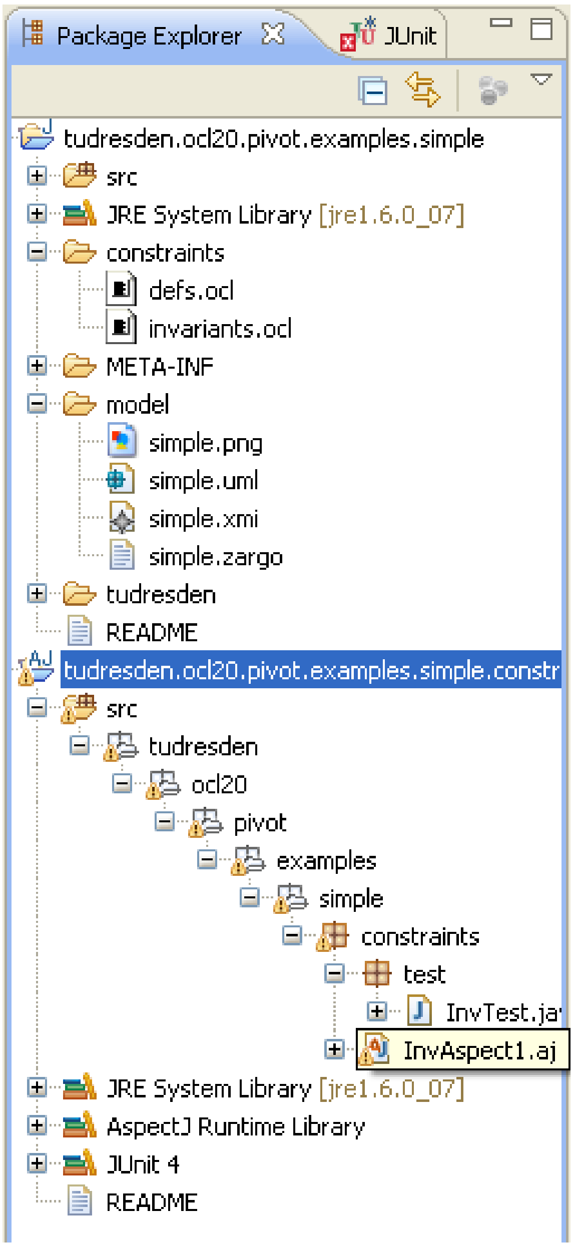
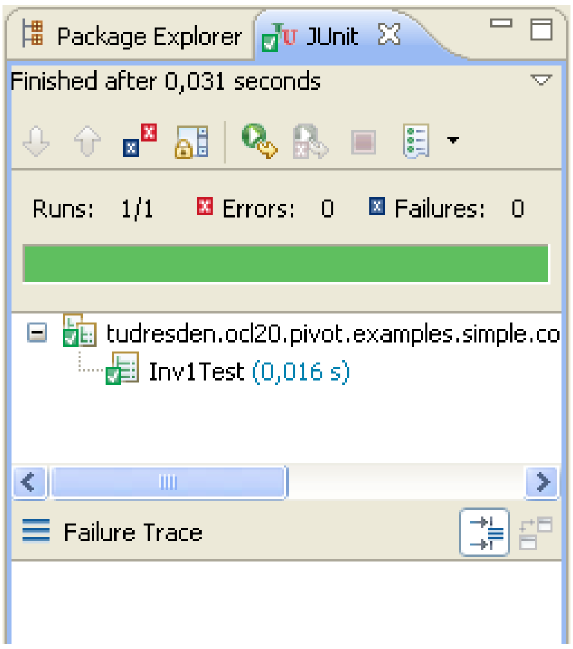


Figure 3.11: The package explorer containing the new generated aspect file.



Finished after 0,031 seconds

Runs: 1/1 Errors: 0 Failures: 0

tudresden.ocl20.pivot.examples.simple.co
Inv1Test (0,016 s)

Figure 3.12: The successfully finished jUnit test case.

3.4 SUMMARY

This Chapter described how to generate AspectJ code using the *OCL22Java* code generator of Dresden OCL2 for Eclipse. A more detailed documentation of the OCL22Java code generator can be found in the Minor Thesis (Großer Beleg) of Claas Wilke [Wil09].

II DEVELOPMENT WITH DRESDEN OCL2 FOR ECLIPSE

4 THE ARCHITECTURE OF DRESDEN OCL2 FOR ECLIPSE

Chapter written by Claas Wilke

This chapter shortly introduces into the architecture of Dresden OCL2 for Eclipse. Before the architecture is presented, some theoretic background is shortly presented.

4.1 THE GENERIC THREE LAYER METADATA ARCHITECTURE

The Object Constraint Language is a language that is always based on another modeling language (usually the UML). Without another language used for modeling, it does not make any sense to define constraints because OCL is used for constraint specification but not for modeling itself. Thus, besides OCL, a modeling language is required to define a model on that OCL constraints can be specified.

Each modeling language is defined in another language, its meta-modeling language. For example, the Unified Modeling Language is defined using the *Meta Object Facility (MOF)* [OMG06], the standardized meta-meta language of the OMG. The MOF is used to describe the UML meta-model that can be used to model UML models. Generally spoken, each model requires a meta-model that is used to describe the model. The model can be instantiated by model instances (for example a UML class diagram could be instantiated by a UML object diagram). The model can be enriched with OCL constraints that are defined on the model (using an OCL meta-model) and can then be verified for model instances of the model.

The OMG introduced the *MOF Four Layer Metadata Architecture* [OMG06][OMG09, p. 16ff] that is used to arrange and structure the meta-model, the model, and its model instances into a layered hierarchy (see Figure 4.1; the layout of the illustration is oriented at the conceptual framework introduced by Matthias Bräuer [Brä07, p. 28]). Generally, four layers exist, the *Meta-Meta-Model Layer (M3)*, the *Meta-Model Layer (M2)*, the *Model Layer (M1)*, and the *Model Instance Layer (M0)*.

OCL constraints can be defined on both, meta-models and models to verify models respectively model instances. Thus, the four layer metadata architecture can be generalized to a *Generic*

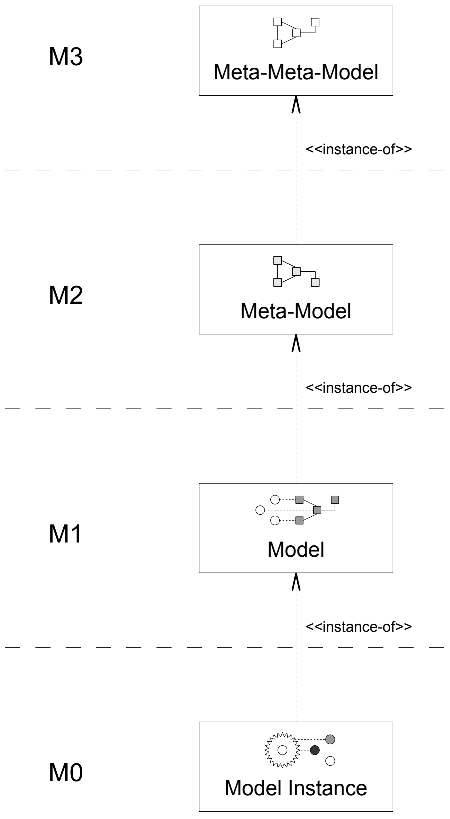


Figure 4.1: The MOF Four Layer Metadata Architecture.

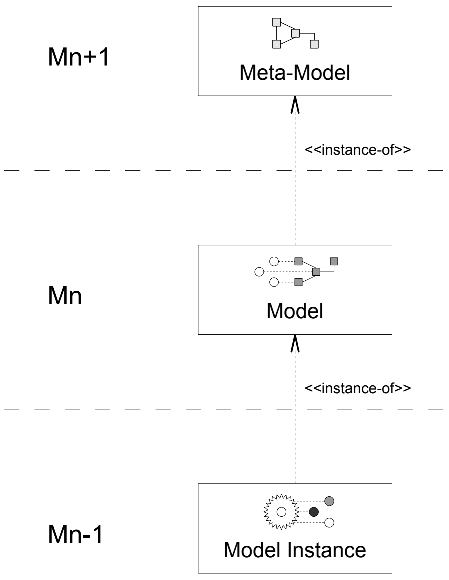


Figure 4.2: The Generic Three Layer Metadata Architecture.

Three Layer Metadata Architecture in the scope of an OCL definition (see Figure 4.2) [DW09]. On the *Mn+1 Layer* lies the meta-model that is used to define the model that shall be constrained. On the *Mn Layer* lies the model that is an instance of the meta-model and can be enriched by the specification of OCL constraints. Finally, on the *Mn-1 Layer* lies the model instance on that the OCL constraints shall be verified. Please note, that in the context of such a generic layer architecture, a model instance can be both a model (like a UML class diagram) or a set of objects (like Java run-time objects).

4.2 THE TOOLKIT'S ARCHITECTURE

The architecture of Dresden OCL2 for Eclipse is shown in Figure 4.3. The architecture is the result of the work of Matthias Bräuer [Brä07] and can easily be extended. The architecture can be separated into three layers: The *Back-End*, the *Basis* and the *Tools Layer*.

The back-end layer represents the repository and the meta-model which can easily be exchanged because all other packages of Dresden OCL2 for Eclipse do not directly communicate with the meta-model but use the *Pivot Model* that delegates all requests to the meta-model instead. For example a possible meta-model is the UML2 meta-model of the *Eclipse Modeling Development Tools (Eclipse MDT) Project* [URL09c]. The second layer is the toolkit basis layer and contains the *Pivot Model*, *Essential OCL* and the *Model Bus*. The use of the pivot model was explained before. The package *Essential OCL* extends the pivot model and implements the *OCL Standard Library* to extend loaded models with OCL constraints. The package *Model Bus* loads, manages and provides access to models the user wants to work with. The third layer contains all tools that are provided by the toolkit. This layer contains the *OCL2 Parser* (which is essential, because the other tools require models that are enriched with already parsed and syntactically and semantically checked constraints), the *OCL2 Interpreter*, and the *OCL22Java Code Generator*. All the tools use the packages of the second layer to access models and model instances and to work on OCL constraints.

Dresden OCL2 for Eclipse has been developed as a set of Eclipse/OSGi plug-ins. All packages that are located in the basis and tools layer represent different Eclipse plug-ins. Additionally, Dresden OCL2 for Eclipse contains some plug-ins to provide GUI elements such as wizards and examples to run Dresden OCL2 for Eclipse with some simple models and OCL expressions.

Dresden OCL2 for Eclipse and The Generic Three Layer Metadata Architecture

The architecture of Dresden OCL2 for Eclipse was designed in respect to the Generic Three Layer Metadata Architecture (introduced in Section 4.1, see Figure 4.4). At the *Mn+1 Layer*, different meta-models and DSLs can be adapted to the toolkit by adapting them to the pivot model (as explained in Chapter 5. Afterwards, models defined with the elements of these meta-models can be loaded into the toolkit at the *Mn Layer*. The models can be enriched with OCL constraints that are described using their own meta-model, Essential OCL, which is described at the *Mn+1 Layer* by extending the pivot model. At the *Mn-1 Layer* model instances of models loaded before can be imported into the toolkit and OCL constraints can be verified on these model instances by using the OCL2 Interpreter or the OCL22Java Code Generator. Please note that model instances can be both, models like UML class diagrams or model instances like Java objects.

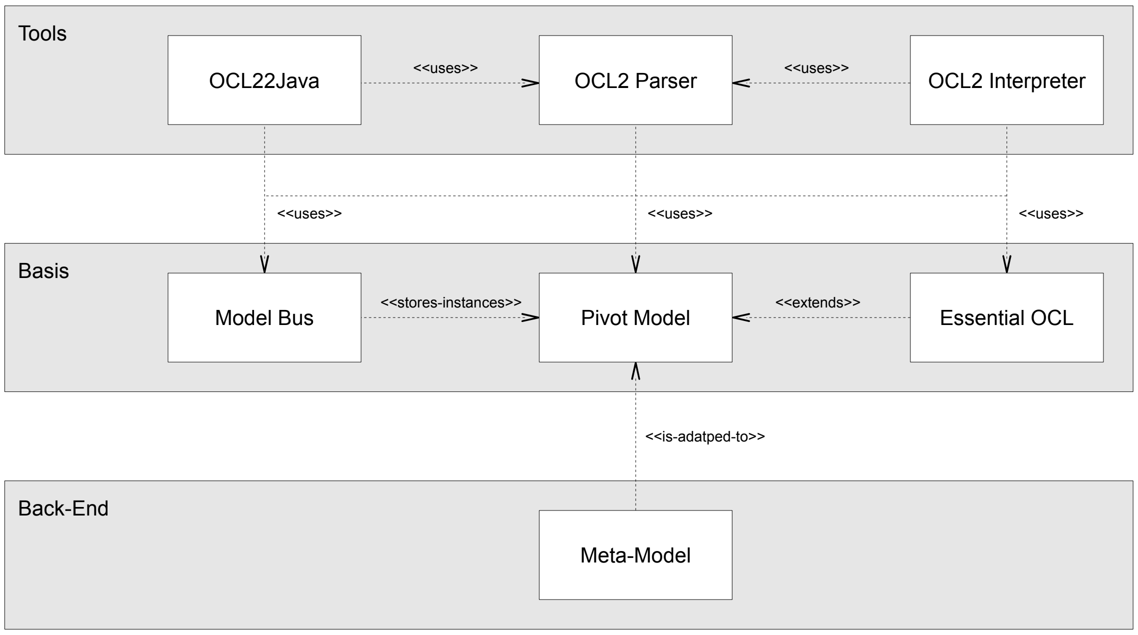


Figure 4.3: The architecture of Dresden OCL2 for Eclipse.

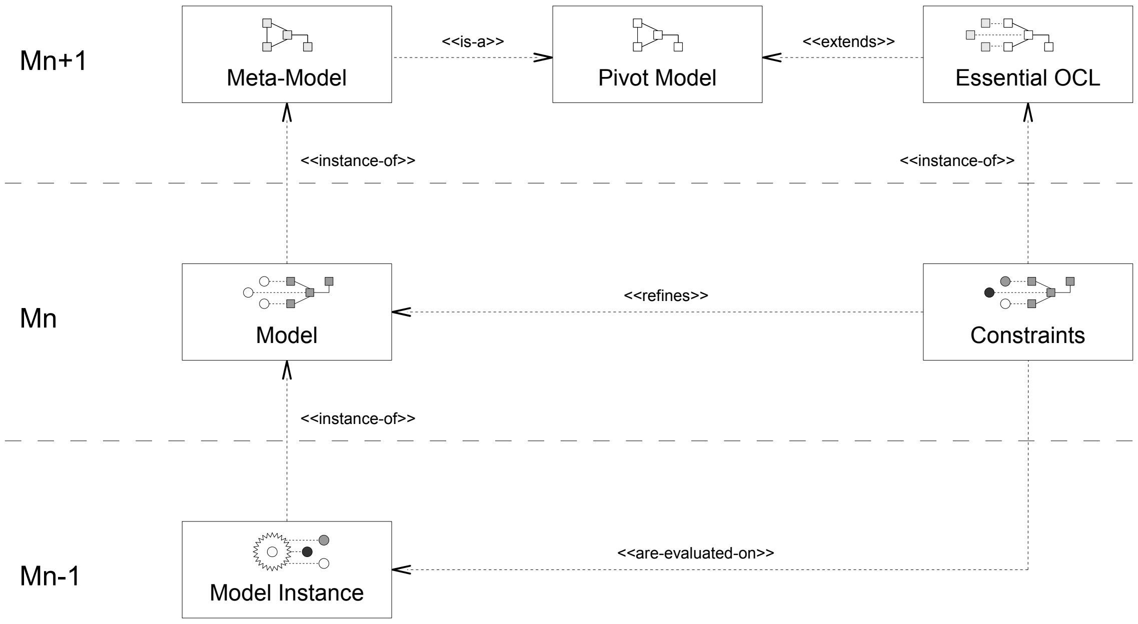


Figure 4.4: The architecture of Dresden OCL2 for Eclipse in respect to the Generic Three Layer Metadata Architecture.

5 ADAPTING A META-MODEL TO THE PIVOT MODEL

TODO: This chapter has not been written yet.

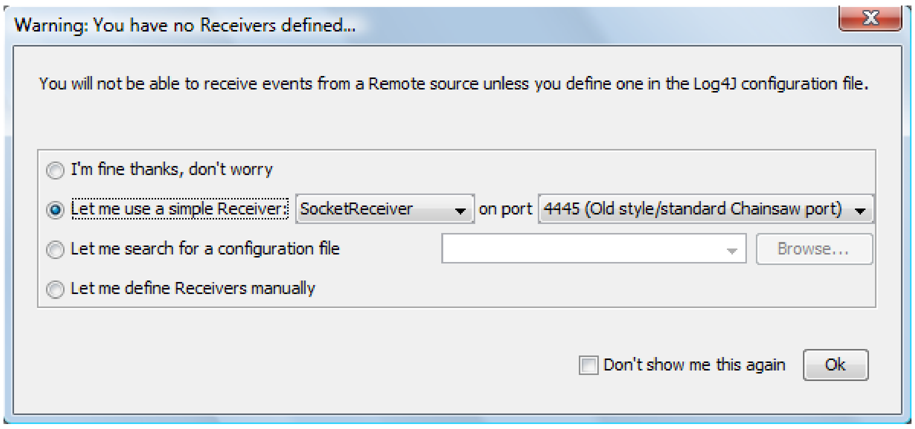
6 THE LOGGING MECHANISM OF DRESDEN OCL2 FOR ECLIPSE

Chapter written by Claas Wilke

Dresden OCL2 for Eclipse uses a *Log4j Logger* to log method entries, exits and errors during the toolkit's execution. If you run Dresden OCL2 for Eclipse as source code plug-ins from an Eclipse workspace, you might receive exceptions like following, although the toolkit works correctly.

```
log4j:ERROR Could not connect to remote log4j server at [localhost].
```

The reason is that the Log4j Logger tries to sent the logged events to a server running at `localhost`. To solve this problem (if you want to) you have to install and setup a logging server at your computer. One logging server you might use is called *Chainsaw* and available at the Apache Logging website [URL09a]. If you start Chainsaw, set up a *SocketReceiver* at port *4445 (Old Style/Standard Chainsaw Port)* (see figure 6.1).


Warning: You have no Receivers defined...

You will not be able to receive events from a Remote source unless you define one in the Log4J configuration file.

- I'm fine thanks, don't worry
- [✓ Let me use a simple Receiver:] SocketReceiver on port 4445 (Old style/standard Chainsaw port)
- Let me search for a configuration file
- Let me define Receivers manually

Figure 6.1: Setting up a simple SocketReceiver in Chainsaw.

7 THE GENERAL TEST SUITE OF DRESDEN OCL2 FOR ECLIPSE

TODO: This chapter has not been written yet.

8 THE GENERIC META-MODEL TEST SUITE

Chapter written by Claas Wilke

To test the adaptation of a meta-model to the pivot model of Dresden OCL2 for Eclipse, the toolkit provides a generic test suite that can be simply instantiated by each adapted meta-model. This chapter shortly presents, how the generic meta-model test suite can be instantiated to test an adapted meta-model.

8.1 THE TEST SUITE PLUG-IN

The generic meta-model test suite is located in the plug-in `tudresden.ocl20.pivot.metamodels.test`. The test suite provides a set of JUnit tests, that check the functionality of all operations that must be adapted by an meta-model that shall be adapted to the pivot model. The adaptation of a meta-model to the pivot model is explained in Chapter 5. The test suite contains about 150 Junit tests.

To instantiate the generic test suite for a new adapted meta-model, only two resources must be provided: (1) a model modeled in the newly adapted meta-model that contains instances of all pivot model types that shall be tested, (2) a Java class that instantiates the test suite with the modeled model. During test execution, the generic test suite uses the provided model modeled using the meta-model to test the meta-model (see Figure 8.1). Both, the model and the Java class are shortly presented in the following sections.

8.2 THE REQUIRED MODEL TO TEST A META-MODEL

Figure 8.2 shows the test model that must be implemented in a meta-model that shall be tested with the generic meta-model test suite. At a first sight, the model seems to be very complex. But many of the contained features are optional, because some data structures and types of the pivot model could be (but do not have to be) implemented by a meta-model. E.g., a meta-model

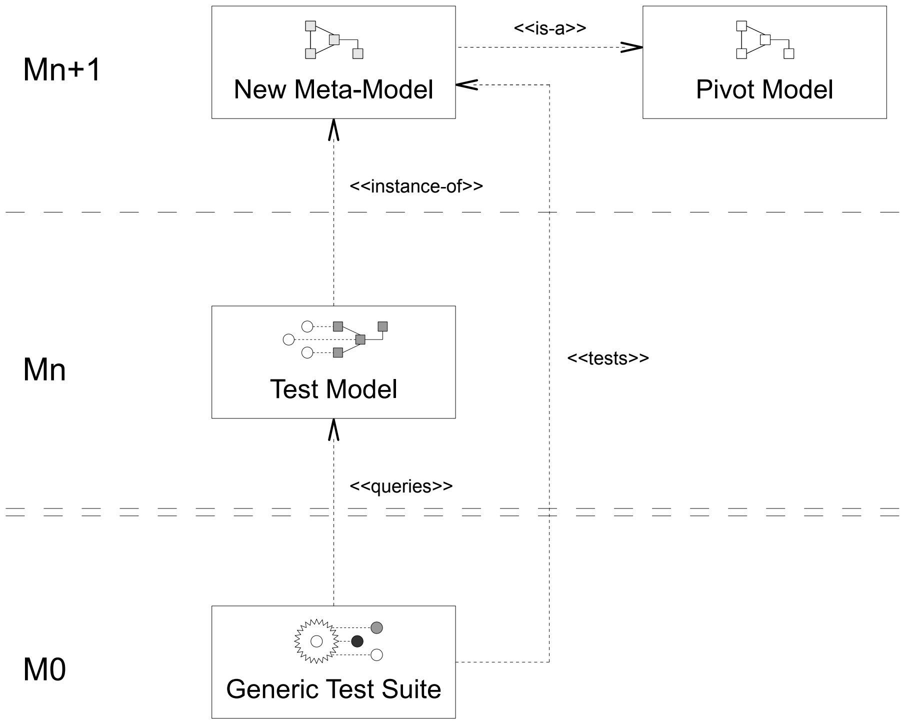


Figure 8.1: The Generic Meta-Model Test Suite in respect to the Generic Three Layer Architeture

can but has not to provide a enumeration type. If a structure is not provided by a test model, the test suite will print a warning during test execution that the expected structure has not been found. If the structure is not adapted, these warnings can be ignored. In the following, all types and relations of the model are explained shortly.

8.2.1 TestTypeClass1 and TestTypeClass2

As their name already tells us, the classes `TestTypeClass1` and `TestTypeClass2` are used to test the adaptation of types. Each meta-model has to provide types, thus, these classes are required. Both classes provided an operation and a property. The association between the two classes is optional because not all meta-models contain associations. But the generalization between `TestTypeClass2` and `TestTypeClass1` is required.

8.2.2 TestTypeInterface1 and TestTypeInterface2

Besides classes, some meta-models also provide a second type that must be mapped to the pivot model type `Type`, which is the interface type. To test the adaptation of the `Type` element for meta-models that have both, classes (or types) and interfaces, the test model contains two interfaces `TestTypeInterface1` and `TestTypeInterface2`. They are optional and can be used to test the adaptation of interfaces. A meta-model that adapts both classes and interfaces is the UML2 meta-model located in the plug-in `tudresden.ocl20.pivot.metamodels.uml2`.

8.2.3 TestEnumeration

To test the adaptation of a enumeration type, the class `TestEnumeration` can be used. Because enumerations are not part of every meta-model, this class of the test model is optional.

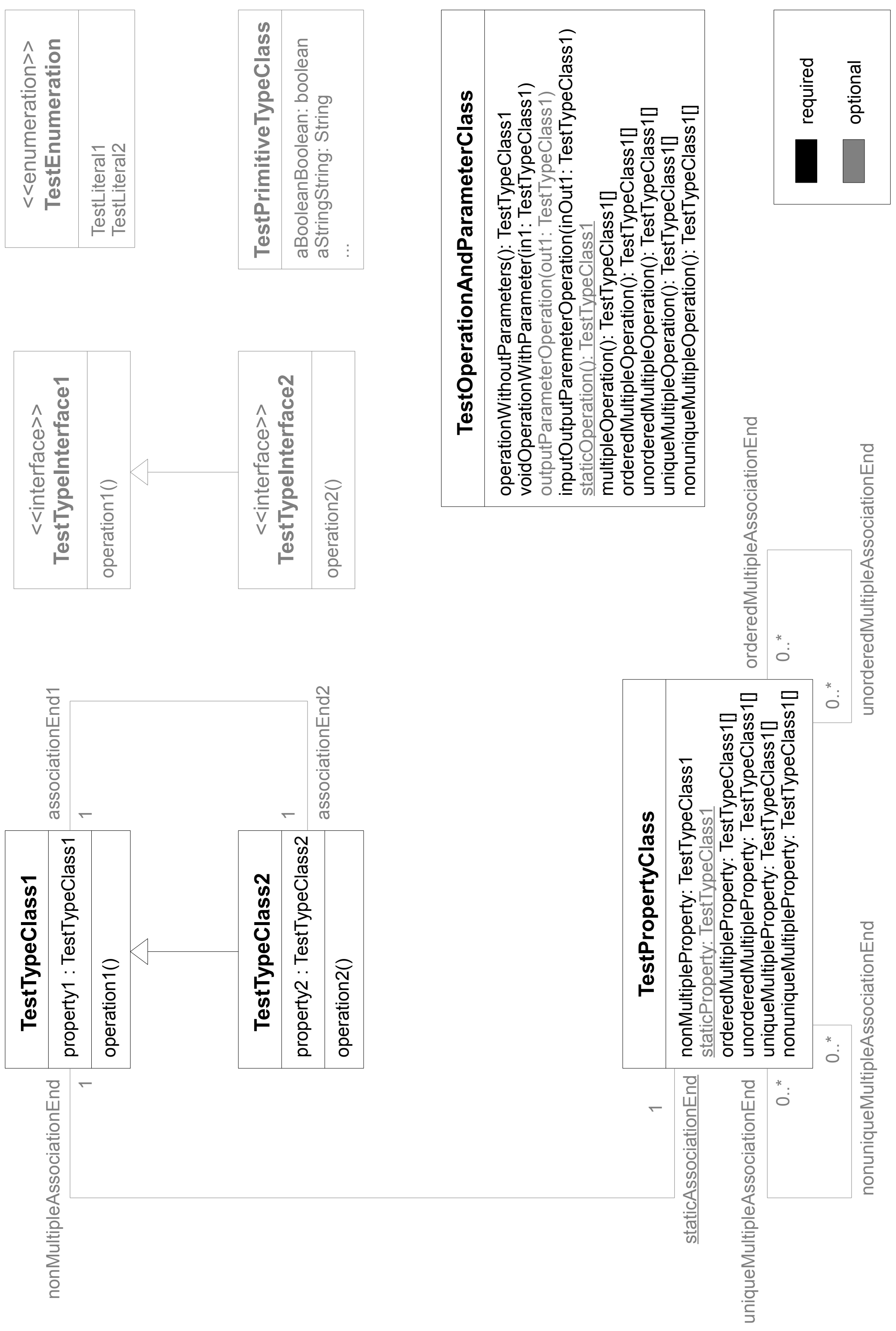

TestTypeClass1
property1 : TestTypeClass1
operation1()
TestTypeClass2
property2 : TestTypeClass2
operation2()
associationEnd1
associationEnd2
1
1
nonMultipleAssociationEnd
1
<<interface>>
TestTypeInterface1
operation1()
<<interface>>
TestTypeInterface2
operation2()
<<enumeration>>
TestEnumeration
TestLiteral1
TestLiteral2
TestPrimitiveTypeClass
aBooleanBoolean: boolean
aStringString: String
...
TestOperationAndParameterClass
operationWithoutParameters(): TestTypeClass1
voidOperationWithParameter(in1: TestTypeClass1)
outputParameterOperation(out1: TestTypeClass1)
inputOutputParemeterOperation(inOut1: TestTypeClass1)
staticOperation(): TestTypeClass1
multipleOperation(): TestTypeClass1[]
orderedMultipleOperation(): TestTypeClass1[]
unorderedMultipleOperation(): TestTypeClass1[]
uniqueMultipleOperation(): TestTypeClass1[]
nonuniqueMultipleOperation(): TestTypeClass1[]
TestPropertyClass
nonMultipleProperty: TestTypeClass1
staticProperty: TestTypeClass1
orderedMultipleProperty: TestTypeClass1[]
unorderedMultipleProperty: TestTypeClass1[]
uniqueMultipleProperty: TestTypeClass1[]
nonuniqueMultipleProperty: TestTypeClass1[]
staticAssociationEnd
1
uniqueMultipleAssociationEnd
0..*
0..*
nonuniqueMultipleAssociationEnd
orderedMultipleAssociationEnd
0..*
0..*
unorderedMultipleAssociationEnd
required
optional

Figure 8.2: The required Test Model to test a Meta-Model adaptation. The grey parts are optional.

8.2.4 TestPrimitiveTypeClass

A special class in the test model is the class `TestPrimitiveTypeClass`. This class contains a property for each primitive type of the adapted meta-model that shall be tested. Each property has the type of the `PrimitiveType` whose adaptation shall be tested. Important is the name of the property. If the property's name starts with `aBoolean`, the type is tested as adapted to a pivot model's `PrimitiveType` of the kind `Boolean`. If the name starts with `anInteger` instead, the types is tested as an `Integer`. E.g., the example property `aStringString` shown in `TestPrimitiveTypeClass` in Figure 8.2 is tested as adapted to a `String`. Table 8.1 shows the adaptation of the different property name prefixes to the different `PrimitiveTypeKinds`.

Property Name Prefix	**PrimitiveTypeKind**
`aBoolean`	`Boolean`
`anInteger`	`Integer`
`aReal`	`Real`
`aString`	`String`

Table 8.1: The adaptation of property names to PrimitiveTypeKinds.

8.2.5 TestPropertyClass

The class `TestPropertyClass` contains properties to test the right adaptation of the pivot model element `Property`. Additionally, the class has many associations that can be used to test the adaptation of a second property type for associations (like in the UML2 meta-model, plug-in: `tudresden.ocl20.pivot.metamodels.uml2`). Thus, all associations are optional and are not required to test a meta-model's adaptation. The names of the contained properties are self-explainable: The property `nonMultipleProperty` is used to test the adaptation of a `Property` that cannot contain multiple values. The property `staticProperty` represents a static `Property` and is optional because not all meta-models contain a `static` modifier. The other properties are used to test multiple `Properties` that are ordered, unordered, unique and non-unique.

8.2.6 TestOperationAndParameterClass

Similar to the class `TestPropertyClass` (presented above), the class `TestOperationAndParameterClass` contains operations to test the adaptation of all different kinds of `Operations`. Additionally, the class is also used to test the adaptation of `Parameters` of operations. Some of the operations are optional (like the static operation and the operation with an output value), others are required.

8.3 INSTANTIATING THE GENERIC TEST SUITE

As mentioned above, to initialize the generic meta-model test suite, only one Java class must be implemented, that instantiates the test suite with the test model implemented in the adapted meta-model, whose adaptation shall be tested. Listing 8.1 shows a Java class that instantiates the test suite to test the UML2 meta-model.

Important is that the class provides a JUnit test suite (according to JUnit 4 conventions), that contains the `MetaModelTestSuite` (line 4). Additionally, the class only has to provide a `setUp()`

```
1  import tudresden.ocl20.pivot.metamodels.test.MetaModelTestPlugin;
2  import tudresden.ocl20.pivot.metamodels.test.MetaModelTestSuite;
3
4  @Suite.SuiteClasses(value = { MetaModelTestSuite.class })
5  public class TestUML2MetaModel extends MetaModelTestSuite {
6
7    /** The id of the {@link IMetamodel} which shall be tested. */
8    private static final String META_MODEL_ID = UML2MetamodelPlugin.ID;
9
10   /** The path of the model which shall be tested. */
11   private static final String TEST_MODEL_PATH = "model/testmodel.uml";
12
13   /**
14    * <p>
15    * Prepares the {@link MetaModelTestSuite}.
16    * </p>
17    */
18   @BeforeClass
19   public static void setUp() {
20
21     MetaModelTestPlugin.prepareTest(UML2MetaModelTestPlugin.PLUGIN_ID,
22       TEST_MODEL_PATH, META_MODEL_ID);
23   }
24 }
```
Listing 8.1: An instantiation of the generic meta-model test suite.

method that can be used to setup the test suite before execution (lines 13 to 23). Inside the `setUp()` method, the operation MetaModelTestPlugin.prepareTest(String, String, String) must be invoked. The method initializes the environment of the generic test suite by setting three arguments:

1. The `ID` of the plug-in that contains the test model used for testing (e.g., `tudresden.ocl20.pivot.metamodels.uml2.test`,
2. The location of the test model relative to the plug-in's root folder (e.g., `model/testmodel.uml`),
3. And the `ID` of the meta-model that shall be tested (e.g. `tudresden.ocl20.pivot.metamodels.uml2`.

Afterwards, the implemented Java class can be executed as a *JUnit Plug-in Test* in Eclipse. The test suite should then be executed and should inform you (by failed test cases) which parts of your meta-model adaptation are wrong implemented or misssing.

III APPENDIX

Tables

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 Plug-in Development Environment (only to run the toolkit using the source code distribution)	http://www.eclipse.org/pde/

Table 2: Software needed to run Dresden OCL2 for Eclipse (**If not using the Eclipse MDT Distribution)**.

Feature	Plug-ins
Core	**Required:** org.apache.commons.lang tudresden.ocl20.pivot.logging tudresden.ocl20.pivot.essentialocl tudresden.ocl20.pivot.essentialcol.edit tudresden.ocl20.pivot.essentialocl.editor tudresden.ocl20.pivot.essentialocl.standardlibrary tudresden.ocl20.pivot.modelbus tudresden.ocl20.pivot.modelbus.ui tudresden.ocl20.pivot.pivotmodel tudresden.ocl20.pivot.pivotmodel.edit tudresden.ocl20.pivot.standardlibrary **Optional:** tudresden.ocl20.pivot.essentialocl.tests tudresden.ocl20.pivot.pivotmodel.tests
Examples	**Optional:** tudresden.ocl20.pivot.examples.living tudresden.ocl20.pivot.examples.pml tudresden.ocl20.pivot.examples.royalandloyal tudresden.ocl20.pivot.examples.royalandloyal.constraints tudresden.ocl20.pivot.examples.simple tudresden.ocl20.pivot.examples.simple.constraints
Interpreter	**Required (for interpretation):** tudresden.ocl20.interpreter tudresden.ocl20.interpreter.ui **Optional:** tudresden.ocl20.interpreter.test
Metamodels	**Required (at least one of the following):** tudresden.ocl20.pivot.metamodels.ecore tudresden.ocl20.pivot.metamodels.uml2
Model Instances	**Required (at least one of the following for interpretation):** tudresden.ocl20.pivot.modelinstancetype.ecore tudresden.ocl20.pivot.modelinstancetype.java
Ocl2Java	**Required (for code generation):** tudresden.ocl20.pivot.ocl2java tudresden.ocl20.pivot.ocl2java.ui **Optional (eventually for code execution):** tudresden.ocl20.pivot.ocl2java.types **Optional:** tudresden.ocl20.pivot.ocl2java.test
Dresden OCL2 for Eclipse Parser Feature	**Required:** tudresden.ocl20.pivot.ocl2parser tudresden.ocl20.pivot.parser tudresden.ocl20.pivot.parser.ui **Optional:** tudresden.ocl20.pivot.ocl2parser.test

Table 3: The plug-ins of Dresden OCL2 for Eclipse related to their feature.

Living Example	
Plug-in Package	tudresden.ocl20.pivot.examples.living
Meta-Model	TODO
Model	TODO
OCL Expressions	TODO
Model Instance	src/tudresden.ocl20.pivot.examples.living.ModelProviderClass.java
PML Example	
Plug-in Package	tudresden.ocl20.pivot.examples.pml
Meta-Model	EMF Ecore
Model	model/pml.ecore
OCL Expressions	expressions/wfr.ocl
Model Instances	modelinstances/goodModelInstance.pml, modelinstances/badModelInstance.pml
Royal and Loyal Example	
Plug-in Package	tudresden.ocl20.pivot.examples.royalandloyal
Meta-Model	EMF Ecore or UML 2.0
Models	model/royalsandloyals.ecore model/royalsandloyals.uml
OCL Expressions	expressions/*.ocl
Model Instance	src/tudresden.ocl20.pivot.examples.royalsandloyals/ ModelProviderClass.java
Simple Example	
Plug-in Package	tudresden.ocl20.pivot.examples.simple
Meta-Model	EMF Ecore or UML 2.0
Model	model/simple.ecore or model/simple.uml
OCL Expressions	constraints/*.ocl
Model Instance	src/tudresden.ocl20.pivot.examples.simple.ModelProviderClass.java

Table 4: The examples provided with Dresden OCL2 for Eclipse.

LIST OF FIGURES

LISTINGS

LIST OF ABBREVIATIONS

AJDT	AspectJ Development Tools
DOT	Dresden OCL Toolkit
DOT4Eclipse	Dresden OCL2 for Eclipse
DSL	Domain-Specific Language
Eclipse MDT	Eclipse Modeling Development Tools
EMF	Eclipse Modeling Framework
GUI	Graphical User Interface
JAR	Java Archive
JDK	Java Development Kit
JRE	Java Run-time Environment
MOF	Meta Object Facility
OCL	Object Constraint Language
OMG	Object Management Group
OSGi	Open Services Gateway initiative
SVN	Subversion
UML	Unified Modeling Language

BIBLIOGRAPHY

[Brä07] Bräuer, Matthias: *Models and Metamodels in a QVT/OCL Development Environment*, Technische Universität Dresden, Germany, Großer Beleg (Minor Thesis), May 2007

[DW09] Demuth, Birgit; Wilke, Claas: Model and Object Verification by Using Dresden OCL. In: *Proceedings of the Russian-German Workshop "Innovation Information Technologies: Theory and Practice", July 25-31, Ufa, Russia, 2009*, Ufa State Aviation Technical University, Ufa, Bashkortostan, Russia, 2009

[OMG06] Object Management Group (OMG), Needham, MA, USA: *Meta Object Facility (MOF) Core Specification, OMG Available Specification*. `http://www.omg.org/spec/MOF/2.0/`. Version: 2.0, January 2006

[OMG09] Object Management Group (OMG), Needham, MA, USA: *OMG Unified Modeling Language^{TM}(OMG UML), Infrastructure*. `http://www.omg.org/spec/UML/2.2/`. Version: 2.2, February 2009

[URL09a] *Apache Chainsaw*. Apache Logging Services. `http://logging.apache.org/chainsaw/`. Version: 2009

[URL09b] *AspectJ Development Tools (AJDT)*. Eclipse AJDT Project Website hosted by the the Eclipse Foundation. `http://www.eclipse.org/aspectj/`. Version: 2009

[URL09c] *Eclipse Model Development Tools*. Eclipse Project Website. `http://www.eclipse.org/modeling/mdt/`. Version: 2009

[URL09d] *Eclipse project Website*. Eclipse project Website. `http://www.eclipse.org/`. Version: 2009

[URL09e] *Polarion Software: Subversive*. Polarion.org Community. `http://www.polarion.com/products/svn/subversive.php`. Version: 2009

[URL09f] *Sourceforge Project Site of the Dresden OCL Toolkit*. Sourceforge project website. `http://sourceforge.net/projects/dresden-ocl/`. Version: 2009

[URL09g] *SVN of the Dresden OCL Toolkit*. Subversion Repository. `http://dresden-ocl.svn.sourceforge.net/svnroot/dresden-ocl`. Version: 2009

[URL09h] *Update Site of the Dresden OCL Toolkit*. Eclipse Update Site. `http://dresden-ocl.svn.sourceforge.net/svnroot/dresden-ocl/trunk/ocl20forEclipse/updatesite/tudresden.ocl20.updatesite_1.2.0`. Version: 2009

[URL09i] *Website of the Dresden OCL Toolkit*. Project website. `http://dresden-ocl.sourceforge.net/`. Version: 2009

[Wil09] WILKE, Claas: *Java Code Generation for Dresden OCL2 for Eclipse*, Technische Universität Dresden, Germany, Großer Beleg (Minor Thesis), February 2009