Electronic Communications of the EASST
Volume 5 (2006)

Proceedings of the Sixth OCL Workshop OCL for (Meta-)Models in Multiple Application Domains (OCLApps 2006)

Integrating OCL and Model Transformations in Fujaba

Mirko Stölzel, Steffen Zschaler and Leif Geiger

16 pages

Guest Editors: Dan Chiorean, Birgit Demuth, Martin Gogolla, Jos Warmer
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Integrating OCL and Model Transformations in Fujaba

Mirko Stölzel[1], Steffen Zschaler[2] and Leif Geiger[3]

[1] s2729561@inf.tu-dresden.de, http://st.inf.tu-dresden.de/
Department of Computer Science
Dresden University of Technology, Germany

[2] steffen.zschaler@inf.tu-dresden.de, http://st.inf.tu-dresden.de/
Department of Computer Science
Dresden University of Technology, Germany

[3] leif.geiger@uni-kassel.de, http://www.se.eecs.uni-kassel.de/se/
Universität Kassel, Wilhelmshöher Allee 73, 34121 Kassel

Abstract: This paper discusses the integration of the Dresden OCL Toolkit into the Fujaba Tool Suite. The integration not only adds OCL support for class diagrams but also makes OCL usable in Fujaba's model transformations. This makes Fujaba's model transformations more powerful, completely platform independent and easier to read for developers who are already familiar with OCL. By using the code generator of the Dresden Toolkit, we are able to generate executable Java code from Fujaba's model transformations including the OCL constraints.

Keywords: Object Constraint Language, Fujaba, Dresden OCL Toolkit, Model Transformation, Story Diagrams

1 Introduction

The Fujaba Tool Suite [Zün99] is a CASE tool which supports Model Driven Development (MDD) [KWB03]. Within MDD model transformations play an important role. Fujaba offers special interaction diagrams to specify model transformations. Within these diagrams most of the transformations are specified graphically. Nevertheless, some expressions have to be specified textually, like complicated constraints, return values, etc. Since Fujaba generates Java source code from model transformations, these textual statements have been specified using Java expression. Currently, no syntax-checking is done for these expressions, so an erroneous expression results in a compile error after code generation. Newer work adds C++ code generation to Fujaba. Note, that if a developer wants to use C++ code generation, the constraints have to be written in C++ syntax.

So, it would be helpful to have a platform-independent constraint language, which makes syntax checking possible within Fujaba's model transformations, and adds code completion and code generation for the different target languages Fujaba offers. This paper suggest to use the Object Constraint Language (OCL) [Obj03a] for this task. To this end, we have integrated the Dresden OCL Toolkit [OCL99] into the Fujaba Tool Suite. This allows us to use OCL as the constraint language for Fujaba's model transformations.

This paper is structured as follows: Section 2 briefly describes how model transformations are specified using Fujaba, Section 3 describes the integration of OCL into Fujaba's model transformations, Section 4 describes the implementation of the Dresden OCL Toolkit integration into Fujaba4Eclipse, Section 5 discusses code generation. In Section 6, related work is presented and Section 7 concludes.

2 Story Diagrams – A Short Overview

The Fujaba Tool Suite [Zün99] uses Unified Modelling Language (UML) [Obj03b] class diagrams to model the structure of an application. In [Stö05] we have integrated the Dresden OCL Toolkit [OCL99] for use in Fujaba's class diagrams. For behaviour specification, model transformations are specified by using graph transformations within Fujaba. This is done by modelling specialized UML interaction diagrams for the method bodies—so-called story diagrams [FNT98, Zün01b]. From such diagrams Fujaba can then generate executable Java source code.

Figure 1 shows such a story diagram. It is an activity diagram into which graph transformation rule have been embedded. The activity diagram models the control flow. The graph transformations within the activities model the behaviour. The first activity of Figure 1 shows such a graph transformation. Here, starting from the object `this`, which is the object the method `nameExists()` is called on, a child is searched via the `children` association. This child's `name` attribute should equal the passed `name` parameter. If such a child is found, it is stored in a local variable called `child`.

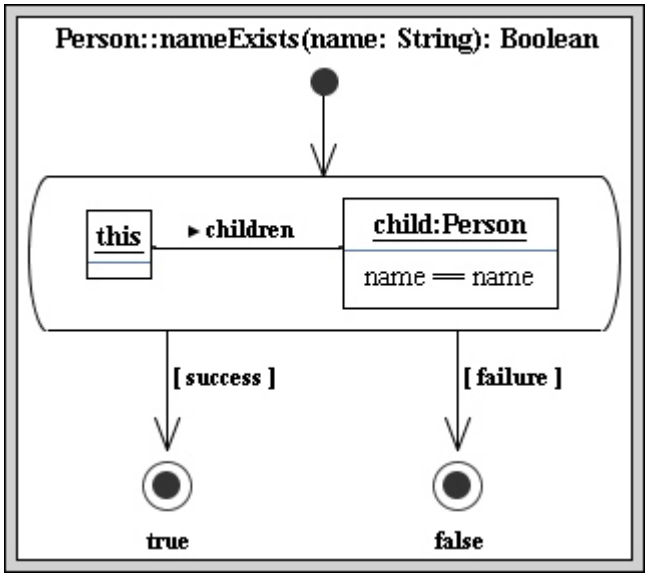

Figure 1: Story diagram

Afterwards, the activity is left. If the graph transformation was applied, it is left via the `success` transition. So the method returns `true`. Otherwise, the `failure` transition is taken,

thus `false` is returned.

Note, that in Fujaba's story diagrams, there are several places, where Java source code can be used. For example, the return value for a stop activity can be any Java expression. This code is directly copied into the source code during code generation.

3 Integrating OCL into Story Diagrams

This section discusses the integration of OCL into Fujaba's story diagrams. First, we give a short overview of the places where OCL can be used in story diagrams. Then some special characteristics of Fujaba's story diagrams, which must be considered to use OCL in story diagrams, are presented. Finally a possible solution which considers these special characteristics will be shown.

3.1 Where to Integrate

In this section, we will present a short overview of all possibilities to use OCL in Fujaba's story diagrams. On the left side of the Figures 2–6 one can see some examples with the original notation of Fujaba while on the right the same example is illustrated using OCL:

Attribute expressions can be used to assign new attribute values to an attribute of an object and to define some additional attribute conditions which must be fulfilled by an object. In the example of Figure 2 the value of the `name` attribute of the `this`-object is assigned to the `name` attribute of the `child`-object by calling the `getName()` method of the `this`-object. On the right side of Figure 2 one can see that the `name` attribute of the `this`-object can be directly referenced using OCL.

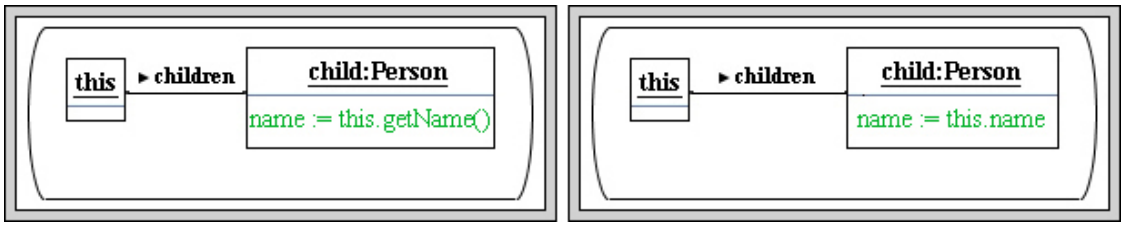

Figure 2: Attribute assertion and attribute constraints

Collaboration statements are used to execute methods, to define new variables or to assign new values to variables. These operations can be combined using the sequential, if- or while composition. In the following example (Figure 3) a collaboration statement is used to define the variable `count` of type Integer. The `sizeOfChildren()` method has been automatically generated by Fujaba for the to-n association `children`. It returns the number of `Person` instances which are assigned to the `this`-object as a child. Using OCL, one can reference the children association directly and can call the `size()` method of OCL-Set to get the number of children of the `this`-object.

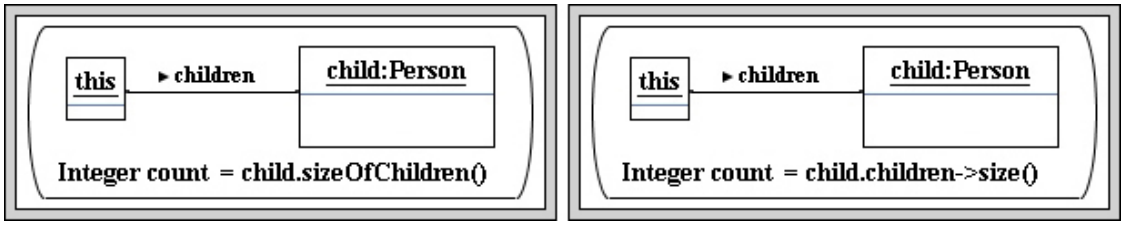

Figure 3: Collaboration statements

Additional constraints are boolean constraints which can be assigned to a story pattern so that the story pattern is applicable if the constraint evaluates to true. In the example of Figure 4 the additional constraint defines that the `this`-object must have exactly five children.

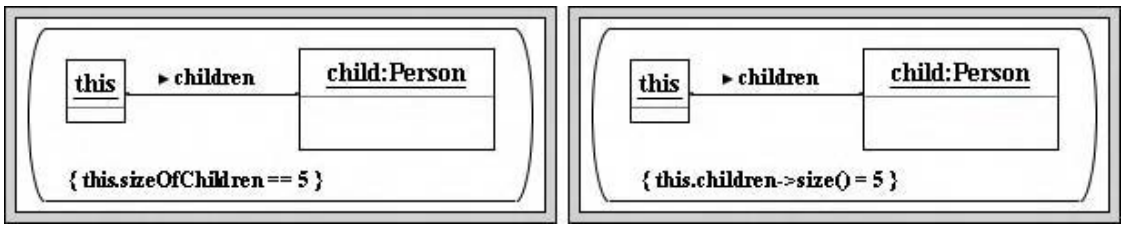

Figure 4: Additional constraints

Boolean transition guards can be used to realize a if- or while-composition in the activity diagram part of the story diagrams. In the following example the variable `found` will be set to true if the `child`-object was successfully bound in the previous story pattern. As one can see on the right side of this example the `oclIsUndefined()` method can be used to formulate the boolean condition with OCL.

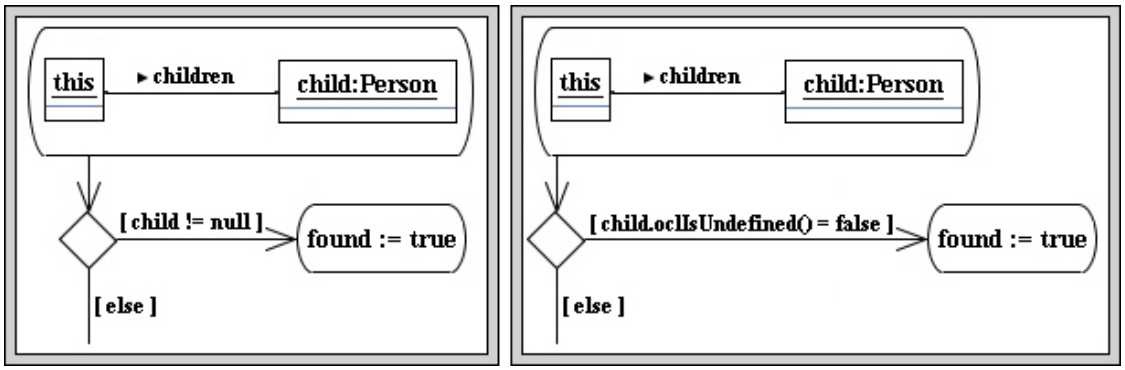

Figure 5: Boolean if-condition

Method return value The last possible use of OCL in Fujaba's story diagrams is represented in Figure 6. At every stop activity, the operation's return value can be defined using a constraint.

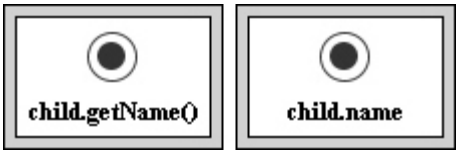

Figure 6: Stop activity

3.2 Resolving Scoping

To integrate OCL in Fujaba's story diagrams we use the OCL parser of the Dresden OCL Toolkit. It checks the syntax and the consistency of an OCL constraints in the context of the containing story diagram. To perform a consistency check the parser of the Dresden OCL Toolkit tries to find all variables which are referenced within an OCL constraint in its story diagram. To do so, the parser has to know which variables and objects are defined in the corresponding story diagram. This information is contained in the so-called context of an OCL constraint.

When generating the context of OCL constraints in Fujaba's story diagrams we have to consider some special characteristics of story diagrams:

- In each story diagram the `this`-object and the method parameters are predefined bound objects. Those can always be referenced in OCL constraints.

- A story diagram in general contains many execution paths. Every path visits different story activities and so different variables and objects can be bound. It can, for example, occur that one variable is not initialized on one special path leading to a story activity and initialized on another one. For this reason, only these variables and objects can be used in an OCL constraint of a story activity which are defined on every path leading to that story activity.

- An object of a story diagram is initialized with a valid value if the corresponding story pattern is applicable. So the objects of a story pattern can only be referenced by the OCL constraints of the next story activity if the activities are connected by a `success` or `eachtime` transition. An `eachtime` transition is used in combination with a so called `foreach` activity. This special activity basically represents repeated matching. It is not left after the first object was found, but the specified transformations are executed for every valid object allocation. In the example of Figure 1 we could have used a `foreach` activity to count all children where the `name` attribute equals the passed `name` parameter.

In the following we present an algorithm which considers these characteristics and can be used to generate the context of an OCL constraint in a story diagram. To obtain this context, an environment is assigned to every element in the story diagram, beginning with the start activity.

An environment encapsulates a set of name–type bindings representing the variables accessible under this environment. When a name lookup occurs, the environment first checks whether it contains a corresponding binding itself. If this is not the case, the environment can delegate the lookup to its parent environments (other environments linked to it via a parent association). If all parent environments agree on the result of the lookup, this will be returned. If they do not agree, the lookup fails. As we will see, parent–child relations can, thus, be used to represent the control flow in a story diagram. Note that story diagrams allow the deletion of objects from the object graph. Therefore, after deletion of an object its name will be no longer bound. To represent this, environments distinguish different types of bindings; one of them is used to mark deleted objects.

In order to clarify the context generation algorithm an example story diagram is represented in Figure 7. There one can see that first an initial environment `e1` is assigned to the start activity of the story diagram and that the `this`-object and the method parameter `var1` are added to this environment. In the next step, the outgoing transition of the start activity is traversed and the first activity is visited. In addition, the environment `e2` is assigned to the activity as input environment. Since the variables `this` and `var1` of the environment `e1` can also be used within the first activity, the parent–child relationship between `e1` and `e2` is created. In the first activity the variable `var2` is created and it is added to the outgoing environment `e3` of the activity.

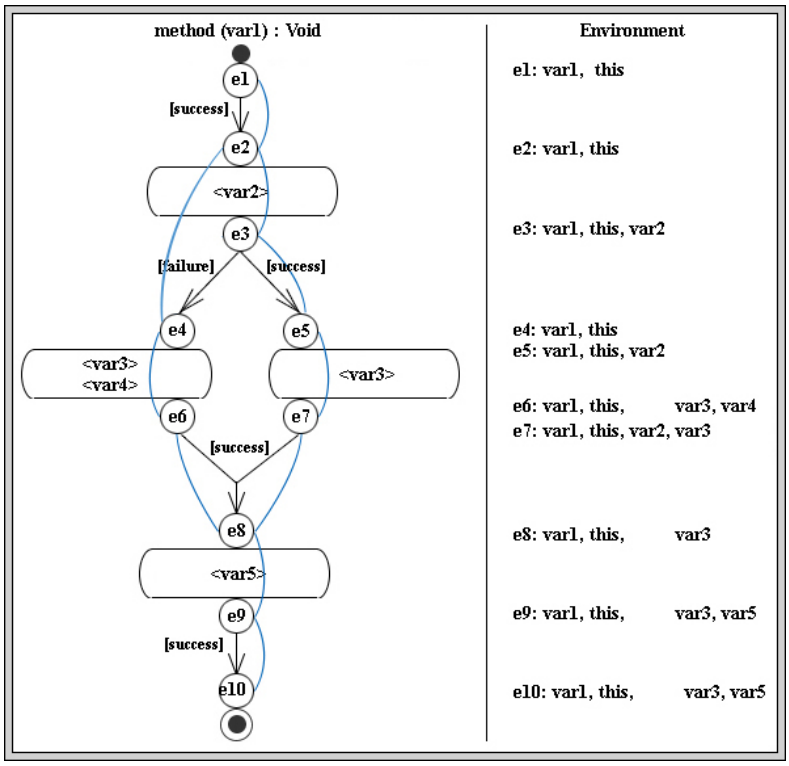

Figure 7: Generation of the OCL-Context

In the next step the two outgoing transitions of the second activity are traversed and the environments `e4` and `e5` are assigned to the corresponding story activities. It must be considered, that on the path following the `failure` transition the story pattern of the second activity was not applicable. Consequently, we cannot assume that the objects of the second activity were successfully bound. Therefore these objects cannot be used in following OCL constraints. That's the reason why the parent–child relationship is made between the environment `e4` and the environment `e2` and not to the environment `e3`. Similarly, the variable `var2` could successfully be bound when taking the `success` transition and can be used in following OCL constraints. So the parent–child relationship between the environment `e3` and `e5` is created. In the next steps the environment `e6` and `e7` are created, which contain the visible variables, and the outgoing transitions are traversed.

As result the environment `e8` is assigned to the next story activity and the parent–child relations between the environment `e8` and the environments `e6` and `e7` are created. At this point the second problem mentioned above must be considered. Since the variable `var4` is defined only on the left path, the environment `e8` does not contain this variable. The same problem applies to the variable `var2`. Because of the `failure` transition this variable can be used only in the right path and thus the variable `var2` is also not a part of the environment `e8`. The last step of the generation process is to generate the environments `e9` and `e10` which is assigned to the stop activity of the story diagram.

4 Fujaba and the Dresden OCL Toolkit

This section discusses the implementation of the OCL integration into Fujabas story diagrams within the scope of Fujaba4Eclipse [Zün01a] and the Dresden OCL Toolkit.

The section starts with a short discussion of the context declaration to be used for the OCL constraints defined in Fujaba's story diagrams. Every OCL constraint must be preceded by a context declaration that defines the context in which the constraint is to be evaluated. Next, we discuss the integration interface of the Dresden OCL Toolkit. This interface has already been used in [Stö05] to allow the specification of OCL constraints in Fujaba's class diagrams. Finally, we present our OCL editor for Eclipse. This allows creating and editing OCL constraints for a given story diagram—and actually for any model edited in an Eclipse-based application. The OCL parser of the Dresden OCL Toolkit is used to check syntax and consistency of the OCL constraints.

4.1 OCL constraint context declaration

The parser of the Dresden OCL Toolkit needs for a consistency check the context declaration of the OCL constraints. That's the reason why the context of the OCL constraints in Fujaba's story diagrams has to be specified.

As already mentioned in Section 3 an environment containing all visible variables is assigned to each story diagram element. Thus, the environment of a story diagram element represents the context of an OCL constraint defined for this element.

In order to illustrate the context declaration of OCL constraints in story diagrams two exam-

ples are shown below. In the first example one can see an OCL constraint which can be used for general constraints of a story activity, an attribute condition or for a transition guard. This constraint is defined in the context of the `cond()` method of the class `Environment` and the evaluation results in `true` if the OCL constraint is fulfilled. In the second example one can see an OCL constraint which can be used to specify an attribute assertion, a collaboration statement or to define the result clause of a method. The difference between these two examples is the result type of the `cond()` method which is equivalent to the type of the asserted value.

```
Environment::cond():Boolean
   post:result = (child.oclIsUndefined())

Environment::cond():String
   post:result = (this.name)
```

Figure 8: Examples

4.2 Integration interface of the Dresden OCL Toolkit

The Dresden OCL Toolkit was developed by the Department of Computer Science of the Technical University Dresden. It can be used to check syntax and consistency of OCL constraints against a UML model and to generate the corresponding Java code. The required model information are managed in an Metadata Repository (MDR) [Mat03] and can be accessed using so called Java Metadata Interfaces (JMI) [Inc06].

When integrating the Dresden OCL Toolkit in a CASE-Tool the problem occurs that the model information required for the consistency check are not part of the toolkit repository, but can be found in the repository of the CASE-Tool. Thus, the main goal of the integration interface was to find a way to allow the parser of the Dresden OCL Toolkit the direct access to the CASE-Tool repository.

We have presented an integration concept in [Stö05]. The main idea is to create representative elements in the toolkit repository which know their corresponding model elements in the CASE-tool repository. These representatives serve as proxies into the CASE-tool repository. Duplicating the CASE-tool repository is not required, all model information can remain in the CASE-tool repository.

The most important part of the integration interface is the class `ModelFacade`. An instance of this class is assigned to a UML model in the toolkit repository and manages the relationships between the representative elements and the corresponding elements in the CASE-Tool repository. Thus, this instance can be used to determine the actual attributes of a representative element related to the model element in the CASE-Tool repository. In order to allow using the integration interface not only for one CASE-Tool but for any CASE-Tools the class `ModelFacade` is defined as an abstract class and has to be implemented CASE-Tool specific.

In order to illustrate the use of the integration interface the consistency check process for the right hand side of Figure 2 is shown below. There one can see the executed steps to find the `this` object referenced in the OCL constraint of Figure 2.

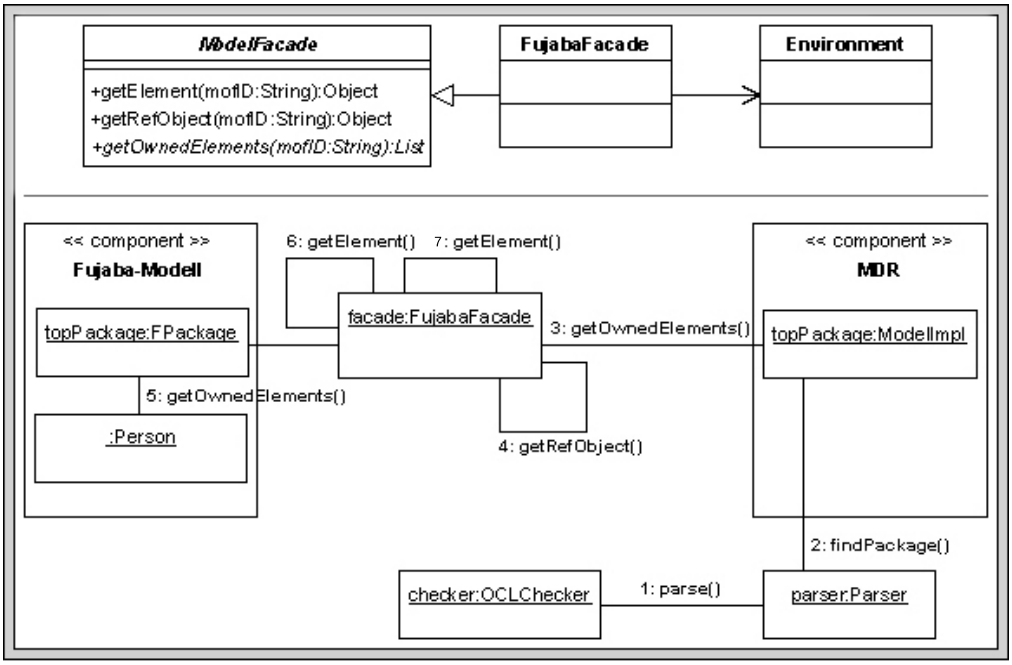

Figure 9: Integration interface of the Dresden OCL Toolkit

To find referenced model elements, the parser of the Dresden OCL Toolkit uses method `findClassifier()` of the `topPackage` representative model element which is the entry point to an UML model in the toolkit repository. Inside the `findClassifier()` method the `getOwnedElements()` method of the JMI interface `Namespace` is called to determine all model elements in the name space of the `topPackage` element.

In order to use the Fujaba specific implementation of the class `ModelFacade` the method `getOwnedElements()` was implemented in a custom way. Thus, the instance of the class `FujabaFacade` assigned to the UML model in the toolkit repository is determined and its `getOwnedElements()` method is called.

Inside this method the `getRefObject()` method is used to find out the corresponding element of the representative element the `getOwnedElement()` method was called for. This way, all model elements defined in the name space of the corresponding element can be determined and the `getElement()` method is called for each of these elements as a parameter. As result of this method a new representative element is created, or the still existing one is returned.

Since the class `Environment` is not part of the Fujaba UML model the representative element for the environment is not in the set of the determined elements. Thus, `getElement()` is invoked additionally with the `Environment` instance referenced by the `FujabaFacade`.

Finally all found representative elements including the environment element are returned as the result of the `getOwnedElements()` method. After this step the parser of the Dresden OCL Toolkit can use the representative element of the environment to search for the `this` variable. For this the `getFeatures()` method of the JMI interface `Classifier` is called and

the same procedure is started again.

In the last step the `getName()` method of the found element representing the `this` variable is used to check whether the `name` attribute exists.

The integration as presented has been implemented for class and story diagrams. Thus, input, consistency and syntax checking of OCL constraints in story diagram are possible, as it is possible for Fujabas class diagrams. We use the algorithm described in Section 3.2 to generate the context of OCL constraints in a story diagram which is used by the Fujaba specific implementation of the `ModelFacade`. Thus, the parser of the Dresden OCL Toolkit is able to check whether referenced variables within an OCL constraint are defined in the corresponding story diagram.

4.3 OCL editor of the Dresden OCL Toolkit

In order to use the Dresden OCL Toolkit within Fujaba4Eclipse to check syntax and consistency of OCL constraints it had to be implemented as a Eclipse plugin. On basis of this plugin an OCL editor plugin was developed and the integration of the Dresden OCL Toolkit for Fujabas class diagrams has already been accomplished in [Stö05] using the Dresden OCL Toolkit integration interface. Presently, we extended the OCL editor plugin to integrate the Dresden OCL Toolkit also for Fujaba's story diagrams.

The OCL editor plugin allows other Eclipse plugins to create and edit OCL constraints, and check syntax and consistency of these constraints against a given model. To do this, the OCL editor provides an extension point consisting of the abstract class `IOCLEditorExtension` which is shown in Figure 10.

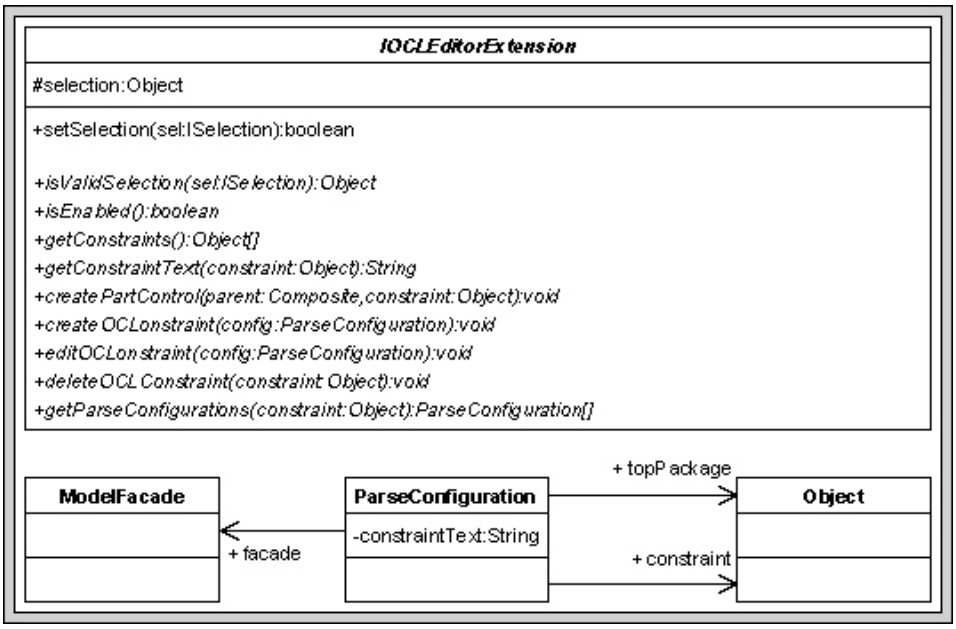

Figure 10: Extension point `IOCLEditorExtension`

In Figure 10 one can see that the class `IOCLEditorExtension` contains a method named `setSelection()` which is called by the OCL editor if the selection of the Eclipse IDE changed. This method is used to check whether the actual selection is a model element of an Eclipse plugin implementing the OCL editor extension point. Inside the `setSelection()` method the `isValidSelection()` method is called which should return a value unequal null if the selection is a model element of the plugin.

The OCL editor can search for all implementations of its extension point being responsible for the actual selection. Thus, all existing OCL constraints can be displayed, edited, deleted or new OCL constraints can be created. In order to realize this some other methods of the class `IOCLEditorExtension` have to be implemented:

- The method `isEnabled()` is called by the OCL editor to check whether OCL constraints can be defined for the selected model element.
- After the method `isEnabled()` returns `true` the method `getConstraints()` is used to get all existing constraints for the selected model element. To display these constraints in the OCL editor area the `getConstraintText()` is called.
- The method `createPartControl()` is called to create the graphical user interface which is used to create and edit OCL constraints
- After an OCL constraint was edited the `getParseConfiguration()` method is used to determine the `ModelFacade` instance, the TopPackage element and the textual OCL constraint which are encapsulated by an instance of the class `ParseConfiguration`. The `ModelFacade` instance and the TopPackage element are used by the OCL editor to check consistency and syntax of the textual constraint.
- If the syntax and the consistency check for an OCL constraints was successful the methods `createOCLConstraint()` or `editOCLConstraint()` are used to create a new model element or edit the existing model element representing an OCL constraint on CASE-Tool side.
- The method `deleteOCLConstraint()` is called to delete an existing OCL model element.

Finally, a screen shot of the OCL editor plugin can be seen in Figure 11. In the left lower part of this figure you can see the OCL-Editor for Eclipse which allows you to create and edit OCL constraints for a given story diagram. Additionally, you can use the OCL parser of the Dresden OCL Toolkit to check syntax and consistency of the OCL constraints against the story diagram. In the example shown in Figure 11 one can see, that an error message is shown in the problems view of eclipse, since the variable `var4` is not defined on the left path of the example story diagram.

5 Generating Code

As already mentioned, Fujaba generates executable Java code from class diagrams and model transformations. The code that would be generated for the left hand side of Figure 4 is shown

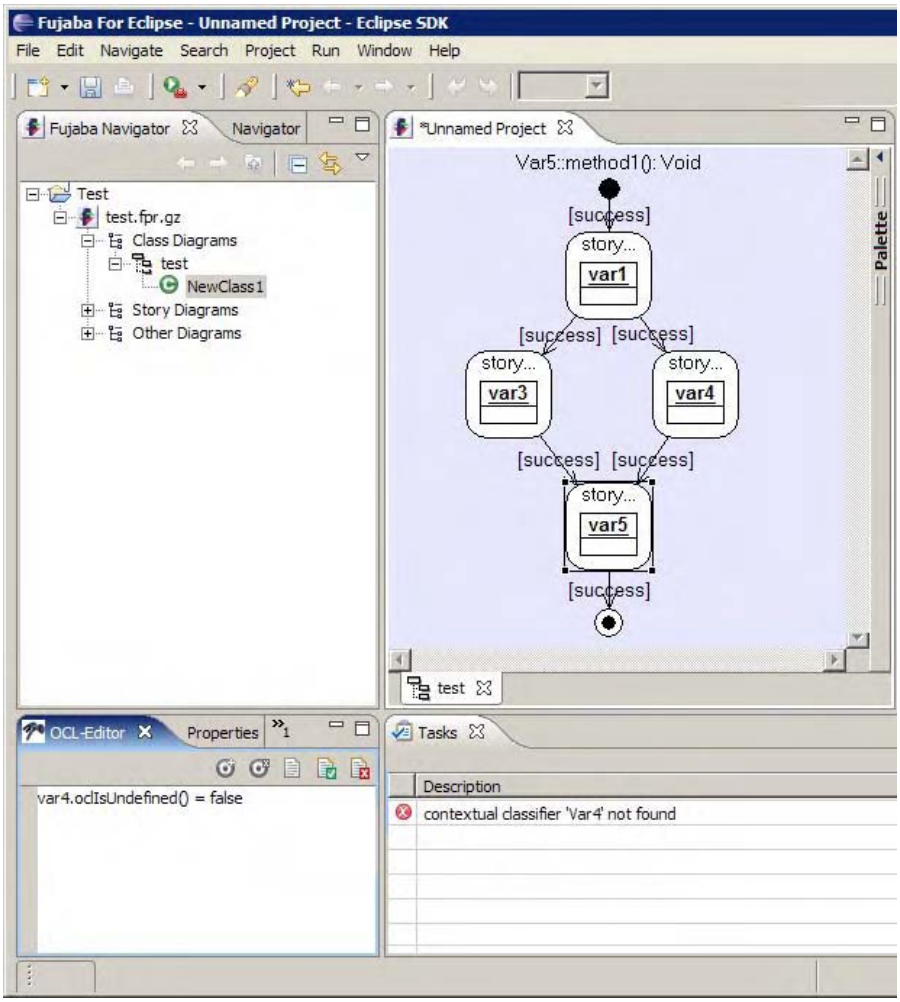

Figure 11: OCL editor Eclipse plugin

below.

```
01 // bind child: Person
02 Iterator iter = this.iteratorOfChildren ();
03 while ( !(fujaba__Success) && iter.hasNext () )
04 {
05    try
06    {
07       child = (Person) iter.next ();
08       // check isomorphic binding
09       JavaSDM.ensure ( !(this.equals (child)) );
10       // constraint
```

```
11       JavaSDM.ensure ( child.sizeOfChildren() == 5 );
12       fujaba__Success = true;
13    }
14    catch ( JavaSDMException e ) {}
15 }
```

To search through all children of the `this` object, a `Iterator` is created in line 02. The while loop from line 04 to line 15 is repeated till one child has been found, that matches all conditions (`fujaba__Success == true`) or till no more child exists in the list. In this loop, in line 07 the current `child` object is fetched from the list. Since the `this` object, and the `child` object are both of class `Person`, it is possible to make a person its own child. Fujaba's semantics forbids such matches (unless explicitly allowed in the story diagram), so this is checked in line 09. Note, that Fujaba provides the library method `JavaSDM.ensure(boolean)` which simply does nothing, when passed true and throws a `JavaSDMException` otherwise. So, if `this` equals `child`, this would end the checks for the current object and continue with the next one. Otherwise the additional constraint is checked in line 11. Note, that the text from the constraint is directly copied into the code surrounded by another `JavaSDM.ensure`. If this test is also passed, `fujaba__Success` is set to true, to indicate that a valid child has been found. The loop is terminated in that case.

If the additional constraint is now specified in OCL, as done in the right hand side of Figure 4, the code generation has to be adapted. We have integrated the code generation of the Dresden OCL Toolkit into Fujaba4Eclipse. The modified code generation leaves most of the code above untouched, but changes the check of the condition in line 11. The source code below shows the code which is now generated.

```
01 //bind child: Person
02 Iterator iter = this.iteratorOfChildren ();
03 while ( !(fujaba__Success) && iter.hasNext () )
04 {
05    try
06    {
07       child = (Person) iter.next ();
08       // check isomorphic binding
09       JavaSDM.ensure ( !(this.equals (child)) );
10       //******************constraint***************
11       OclAny self =
12            (OclAny) Ocl.getOclRepresentationFor(this);
13       OclBoolean constraintValid=
14           self.getFeatureAsCollection("children").
15              size().isEqualTo( new OclInteger(5) );
16       JavaSDM.ensure ( constraintValid.isTrue() );
17       //******************constraint***************
18       fujaba__Success = true;
19    }
```

```
20    catch ( JavaSDMException e ) {}
21 }
```

Within the Dresden OCL Toolkit the OCL Standard Library is implemented by some Java classes, which are used by the Java code, created by the Java code generator of the Dresden OCL Toolkit, to evaluate an OCL constraint. To evaluate the OCL constraint of the right hand side of Figure 4 an instance of the class `OCLAny` is created as one can see in line 11 of the code example shown above. This instance is used in line 13 to get an instance of the class `OCLCollection` which represents the children association end of the `this`-object. Afterwards the number of the elements in this collection is determined using the `size()` method of the `OCLCollection` instance. This results in an instance of the class `OCLInteger` of which the `isEqualTo()` method is used to evaluate whether the number of collection elements equals 5. As the result of the `isEqualTo()` method call an instance of the class `OCLBoolean` is created of which the `isTrue()` method returns the result of the comparison. So the result of this method can be used as input of the `JavaSDM.ensure()` method call as one can see in line 15.

6 Related work

Many CASE tools offer OCL support for class diagrams. For example, the Dresden OCL Toolkit has been integrated with Together and ArgoUML. But those tools have no support for model transformation and no integration of OCL in other diagrams. The EMFT project [The06] supports OCL for constraints and queries. One can use OCL for constraints on the static model and for specification of querying behavior. This way e.g. derived attributes can be modeled. So EMFT uses OCL for some very basic behavior specification. But it has no support for model transformations.

The QVT standard [Obj06] by the OMG has some similar ideas. QVT defines a model transformation language which uses OCL. QVT extends the OCL with imperative expressions to make it more powerful. In this ImperativeOCL things like attribute assignments, link creation etc. can now be expressed. In our approach this imperative part is modeled using story diagrams. Currently, complete tool support for QVT is still missing.

7 Conclusions

The Fujaba Tool Suite is a CASE-Tool which supports the most important diagrams of the Unified Modelling Language with code generation for Java. To also specify the behaviour of a system modelled with Fujaba one can use so-called story diagrams.

As described in Section 2 story diagrams combine UML activity diagrams and collaboration diagrams for the specification of methods. Within story diagrams some expressions, like additional constraints, return values, etc are specified textually using Java expressions. These expressions are inserted identically in the code generated by Fujaba. If a developer wants to use another programming language than Java every constraint within the story diagrams have to be changed separately. So it is useful to specify the additional constraints using the Object Constraint Language.

Therefore, we discussed the possibilities to use OCL in Fujaba's story diagrams in Section 3 and described some special characteristics which must be considered to generate the context of OCL contraints within story diagrams. After that we explained an algorithm to generate the OCL context considering the special characteristics.

On basis of the context generation algorithm the integration of the Dresden OCL Toolkit into Fujaba4Eclipse using the Dresden OCL Toolkit integration interface is described in Section 4. Additional

In Section 5 we described the code generation for Fujaba's story diagrams and discussed how the generated code of a story diagram could look like using OCL.

As already mentioned in Section 4, we use the Dresden OCL Toolkit to integrate OCL in Fujaba's story diagrams. This enables using OCL in various places in Fujaba's story diagrams, while maintaining the ability to generate code. Development of a prototype implementation of the concepts discussed in this paper was completed in [Stö06].

Finally, the implemented integration has been modularly constructed from Eclipse plugins. Therefore, integration with other Eclipse-based CASE tools should be very straight forward.

Bibliography

[FNT98] T. Fischer, J. Niere, L. Torunski. Konzeption und Realisierung einer integrierten Entwicklungsumgebung für UML, Java und Story-Driven Modeling. Diplomarbeit, University of Paderborn, 1998.

[Inc06] S. M. Inc. Java Metadata Interface. Nov. 2006. http://java.sun.com/products/jmi/. http://java.sun.com/products/jmi/

[KWB03] A. Kleppe, J. Warmer, W. Bast. *MDA Explained: The Model-Driven Architecture–Practice and Promise*. Addison-Wesley, 2003.

[Mat03] M. Matula. NetBeans Metadata Repository. Mar. 2003. http://mdr.netbeans.org/MDR-whitepaper.pdf. http://mdr.netbeans.org/MDR-whitepaper.pdf

[Obj03a] Object Management Group. UML 2.0 OCL Specification. OMG document ptc/2003-10-14, Oct. 2003.

[Obj03b] Object Management Group. UML Resource Page. 2003. http://www.omg.org/uml/

[Obj06] Object Management Group. MOF QVT Final Adopted Specification. 2006. http://www.omg.org/docs/ptc/05-11-01.pdf

[OCL99] OCL Toolkit Team. Dresden OCL Toolkit homepage. 1999. http://dresden-ocl.sourceforge.net/

[Stö05] M. Stölzel. OCL für Fujaba. Großer Beleg, Technische Universität Dresden, 2005. In German.

[Stö06] M. Stölzel. Verwendung der OCL zur Formulierung von Bedingungen in Storydiagrammen. Diplomarbeit, Technische Universität Dresden, 2006. In German.

[The06] The Eclipse Foundation. EMFT - Eclipse Modeling Framework Technologies. 2006. http://www.eclipse.org/emft/projects/ocl/

[Zün99] A. Zündorf. The Fujaba Toolsuite. 1999. http://www.fujaba.de/

[Zün01a] A. Zündorf. Fujaba for Eclipse. 2001. http://wwwcs.uni-paderbord.de/cs/fujaba/projects/eclipse/

[Zün01b] A. Zündorf. Rigorous Object Oriented Software Development, Habilitation Thesis. University of Paderborn, 2001.