Integrating OCL and Model Transformations in Fujaba

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

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

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

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.

1 Introduction

The Fujaba Tool Suite [1] is a CASE tool which supports Model Driven Development (MDD) [2]. 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, adds code completion and code generation for the different target languages Fujaba offers. This work suggest to use the Object Constraint Language (OCL) [3] for this task. We have integrated the Dresden OCL Toolkit [4] into the Fujaba Tool Suite. So, we use OCL as constraint language for Fujaba's model transformations.

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 discusses code generation and Section 6 concludes.

2 Story Diagrams – A Short Overview

The Fujaba Tool Suite [1] uses Unified Modelling Language (UML) [5] class diagrams to model the structure of an application. A previous work [6] has already integrated the Dresden OCL Toolkit [4] for use in Fujaba's class diagrams. For behavior specification, model transformation 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 [7, 8]. From such diagrams Fujaba can then generate executable Java source code.

Figure 1 show such a story diagram. The activity diagram models the control flow. The graph transformations within the activities model the behavior. 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 search 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`.

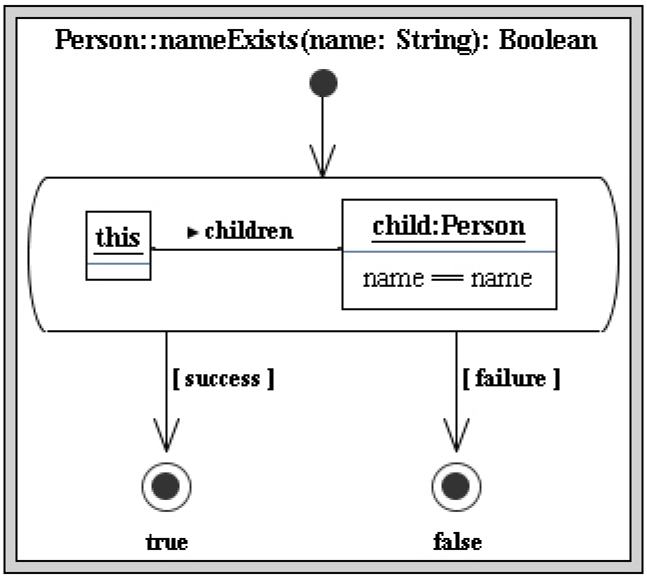


Fig. 1. Story diagram

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, e.g. the return value for a stop activity can be any Java expression and 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. At first it will give you a short overview of the possibilities to use OCL 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 actual 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.

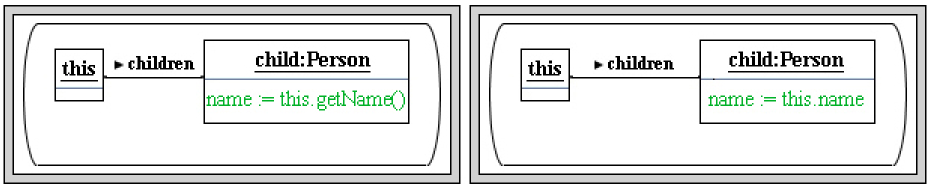


Fig. 2. Attribute assertion and attribute constraints

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 is an automatic generated method of 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.

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.

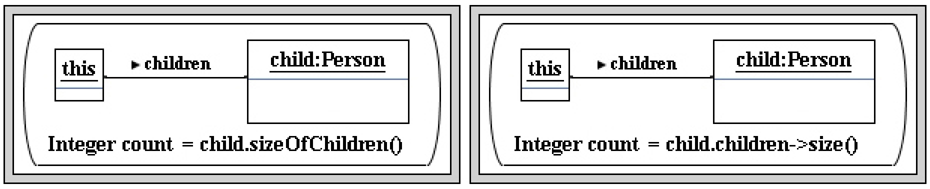

this
▸ children
child:Person
Integer count = child.sizeOfChildren()
this
▸ children
child:Person
Integer count = child.children->size()

Fig. 3. Collaboration statements

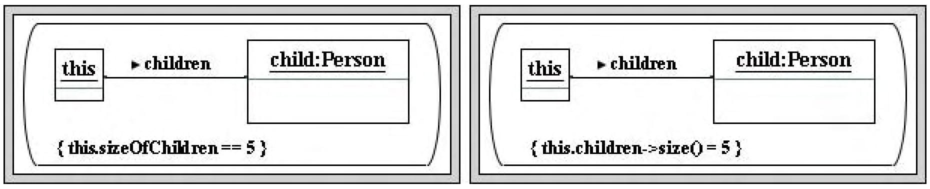


Fig. 4. Additional constraints

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.

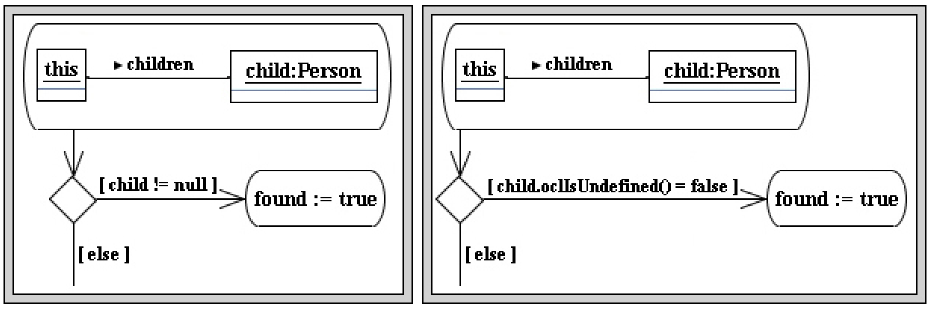


Fig. 5. Boolean if-condition

Method return value The last possible use of OCL in Fujaba's story diagrams is represented in Figure 6. There one can see that you have the possibility to provide a return clause for a ***stop activity*** of a story diagram.

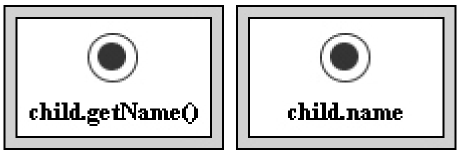


Fig. 6. Stop activity

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. So we have to generate the context of the OCL constraints in a story diagram.

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

– In story diagrams 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 differnt story activities and so different variables and objects can be bound. It can e.g. occur that one variable is not initialized on one special path leading to a story activity and initialized on another one. That's the reason why only these variables and objects can be used in an OCL constraint of a story activity which are defined on every paths 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 an story pattern can only be referenced by the OCL constraints of the next story activity if the story activity is connected by a success or eachtime transition. An eachtime transition is used in combination with a so called foreach activity. This special activity is not left after the first object was found, but the specified transformations are executed on every valid object allocation. In the example of Figure 1 we could have use 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. The context, called environment, of an OCL constraint contains all variables that are visible for the OCL expression. 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 to 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 the next figure. There one can see that first an initial environment e1

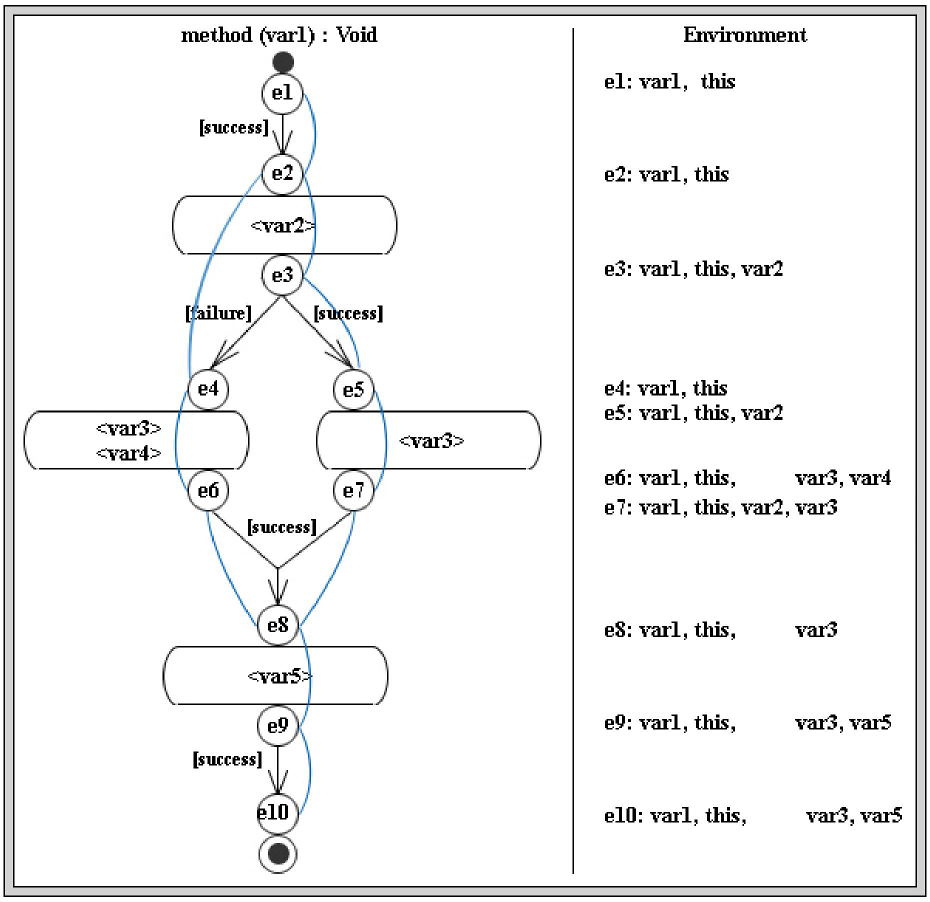


Fig. 7. Generation of the OCL-Context

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.

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 also. Because of the success 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.

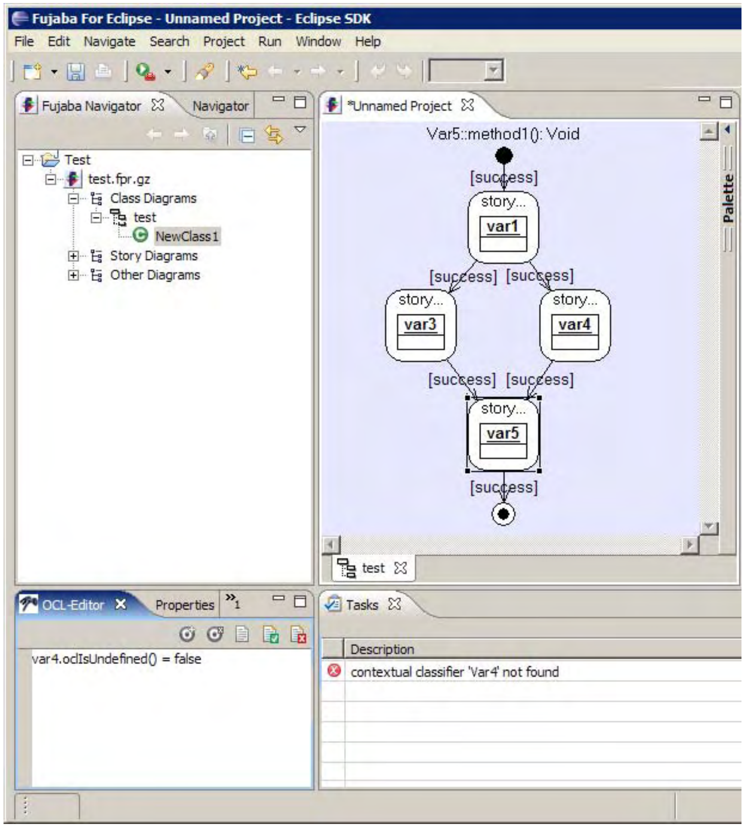
The last step of the generation process is to generate the environments e9 and e10 which is assigned to the stop activity of the story diagram.

3.3 Fujaba and the Dresden OCL Toolkit

Fujaba4Eclipse[9] is a Eclipse Plugin that among other functions integrates Fujaba's story diagrams into eclipse to specify methods. On the basis of Fujaba4Eclipse the integration of the Dresden OCL Toolkit[4] for Fujaba's class diagrams has already been accomplished in [6].

We are, presently, extending this integration to also cover story diagrams. Thus, input, consistency and syntax checking of OCL constraints in story diagram should be possible, as it is possible already for Fujaba's class diagrams. We use the algorithm described in Section 3.2 to generate the context of OCL constraints in a story diagram. The generated context is used by the parser of the Dresden OCL Toolkit to check whether referenced variables within an OCL constraint are defined in the corresponding story diagram.

Figure 8 shows a screen shot of the tool. 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. Additional 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 8 one can see, that an error message is shown in the problems view of eclipse, since the variable var4 is not defined on the right path of the example story diagram.



var4.oclIsUndefined() = false

contextual classifier 'Var4' not found

Fig. 8. Generation of the OCL-Context

4 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 below.

```
// bind child: Person
Iterator iter = this.iteratorOfChildren ();
while ( !(fujaba__Success) && iter.hasNext () )
{
   try
   {
      child = (Person) iter.next ();
      // check isomorphic binding
      JavaSDM.ensure ( !(this.equals (child)) );
      // 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. Our semantics forbids that (if not stated different), 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 are currently integrating the code generation of the Dresden OCL Toolkit in the presented work. The modified code generation would leave most of the code above untouched, but changes only the check of the condition in line 11. The source code below shows the code which is 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` which `isEqualTo()` method is used to evaluate whether the number of the collection elements equals 5. As result of the `isEqualTo()` method call an instance of the class `OCLBoolean` is created which `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.

5 Related work

Many CASE tools offer OCL support for class diagrams. The Dresden OCL Toolkit e.g. was also integrated in Together and ArgoUML. But those tools have no support for model transformation and since no integration of OCL in other diagrams. The EMFT project [10] 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 [11] 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. Since now, complete tool support for QVT is still missing.

6 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 behavior 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.

In Section 4 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 3, 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 is nearing completion.

References

1. Zündorf, A.: The fujaba toolsuite. http://www.fujaba.de/ (1999)
2. Kleppe, A., Warmer, J., Bast, W.: MDA Explained: The Model-Driven Architecture–Practice and Promise. Addison-Wesley (2003)
3. Object Management Group: UML 2.0 OCL specification. OMG document ptc/2003-10-14 (2003)
4. OCL Toolkit Team: Dresden OCL Toolkit homepage. http://dresden-ocl.sourceforge.net/ (1999)
5. Object Management Group: UML resource page. http://www.omg.org/uml/ (2003)
6. Stölzel, M.: OCL für Fujaba. Großer Beleg, Technische Universität Dresden (2005) In German.
7. Fischer, T., Niere, J., Torunski, L.: Konzeption und Realisierung einer integrierten Entwicklungsumgebung für UML, Java und story-driven modeling. Diplomarbeit, University of Paderborn (1998)
8. Zündorf, A.: Rigorous object oriented software development, habilitation thesis (2001)
9. Zündorf, A.: Fujaba for Eclipse. http://wwwcs.uni-paderbord.de/cs/fujaba/projects/eclipse/ (2001)
10. The Eclipse Foundation: Emft - eclipse modeling framework technologies. http://www.eclipse.org/emft/projects/ocl/ (2006)
11. Object Management Group: Mof qvt final adopted specification. http://www.omg.org/docs/ptc/05-11-01.pdf (2006)