UML1.4 to 2.1 Activity Diagram Model Migration with Fujaba - a Case Study

Andreas Koch  
SE, Kassel University  
Wilhelmshöher Allee 73  
34121 Kassel  
email@andreaskoch.net

Ruben Jubeh  
SE, Kassel University  
Wilhelmshöher Allee 73  
34121 Kassel  
ruben.jubeh@uni-kassel.de

Albert Zündorf  
SE, Kassel University  
Wilhelmshöher Allee 73  
34121 Kassel  
albert.zuendorf@uni-kassel.de

ABSTRACT
We have modeled a UML1.4 to UML2.2 Activity Diagram model transformation for the TTC2010 Transformation Tool Contest with the Fujaba Tool Suite. The solution uses core fujaba feature: the whole application is modeled using Story Driven Modelling [4].

1. INTRODUCTION
This paper reports on our case study with the Fujaba environment, cf. [www.fujaba.de], on building a UML1.4 to UML2.2 migration transformation for the TTC2010 Transformation Tool Contest. The transformation reads XMI files as input and can write a corresponding XMI2 file. The transformation is specified using Story Driven Modeling (SDM). It is similar to last years BPMN2BPEL transformation solution, see [1].

A common task in model driven software development is to migrate models when their corresponding meta models evolve. In this case study, Activity Diagrams specified in UML1.4 are migrated to the equivalent ones specified in UML2.2. This is a very exciting task, as Fujabas metamodel itself is heavily based on UML1.4 and Story Diagrams rely on activity diagram constructs. We want to migrate Fujaba to an up-to-date UML2.x metamodel in the future, so this case study will examine one possible approach to do that.

2. META MODELS AND FRAMEWORK
For getting started with the Meta Models, one solution within our Tool Suite is to import existing class binaries as .class-files (preferably available as EMF-conforming model), which would have been possible for the Eclipse UML2 plugins. As the Case Study gives .ecore-Files as reference, we imported those with the roundtrip/reverse engineering Tool UMLab1, which has an adapter to Fujaba. As the given models were simplified not very comprehensive, minor extensions like container classes were applied by hand. Figure 1 and 2 show the used models within the class diagram editor of Fujaba.

3. STORY DRIVEN MODELING
Story diagrams are graph rewrite rules embedded in activity diagrams that allow to query and modify the application’s object graph on a high level of abstraction. Each Story diagram is associated with a method, so Story diagrams call each other by invoking other Story diagram methods. A graph rewrite rule consists of an object graph that is used as a query pattern to be matched in the current runtime object graph. It is similar to an object diagram, but adds also graph/object modifications. Our graph pattern distinguish bound objects and unbound objects. Bound objects are already matched to runtime objects and need not to be searched any more. In our notation, bound objects show only their name and omit their type. At execution time, unbound objects are matched onto runtime objects such that the overall match conforms to the search pattern graph. In addition to querying the runtime graph with a pattern graph, a graph rewrite rule also allows to model modifications of the matched elements.

Figure 3 shows a story diagram method. It’s the Constructor method of the class ModelMigration taking an ActivityGraph instance (from UML1.4) as parameter (see Figure 4 for associations of this class).

This method contains three graph rewrite steps. The first one looks for a state instance associated with the ActivityGraph (UML1.4) parameter, and creates, when found, an (UML2.2) Activity instance. Object or link creation is denoted by the ≪create≫ stereotype and green color. Furthermore, it assigns the same name via an attribute assignment expression and sets all required links between the

1http://www.umlab.de/
4. THE TRANSFORMATION

Figure 4 shows the class diagram of our transformation. The central class `ModelMigration` refers to both the input graph as UML1.4 `ActivityGraph` and the output graph instance (UML2.2).

Graph rewrite rules are an excellent means to model an applications behavior in terms of operations on its underlying object graph. The graph patterns allow to express complex graph queries that are (with some exercise) easy to read and to understand. Thereby, the programs are easier to extend and to maintain.

Figure 5 shows the input model, either constructed as separate graph construction rule or imported via XML1.2 file, as object model in the heap memory, visualized by the eDOBS runtime debugger tool. Figure 6 in the appendix shows the same model with partitions, equal to the model given in the case study. In the following, we will show some of the methods performing the migration transformation, and the corresponding runtime object graphs. We start the transformation by calling `migrate(StateHandler)` (cf. figure 7) on a model migration instance. As a result of it’s constructor referenced by it’s parent class `Activity`. To distinguish between the different operation modes, as required by the extension discussed in 5.1, it delegates some migration work to a `StateHandler`. Furthermore, it remembers already migrated states via a `StateMapping` pointing to elements of both meta models.

Figure 2: UML2.2 Metamodel

Figure 3: migration entry point: the constructor

migration and the other object. This is a preparation step of the migration transformation, creating the parent instance (Activity) of the migrated model. The second rewrite rule just looks for all partitions associated with the input activity graph and creates, for each found (that is indicated by the double outer box) partition, an `ActivityPartition` instance (UML2.2).
Figure 7: main migration method: find initial node and continue

method, it is already connected to an Activity instance. Figure 7 shows the story diagram of this method. It searches for the Pseudostate declared as initial state and creates a corresponding UML2.2 InitialNode instance. The transformation continues after that by invoking another story pattern: migrateTransition(trans, node, boolean) is called via a collaboration statement on the ModelMigration instance itself (this).

Figure 8: Runtime objects during transformation: first recursion

The migrateTransition(trans, node, boolean) method is shown in Figure 10. Figure 8 shows an excerpt of the object graph for the sample model just before the method gets executed the first time. It will now transform the Actionstate a16.

Figure 9: Runtime objects during transformation: second recursion

As a16 still has successor(s) (cf. figure 5), the algorithm will continue and transform the next action state a11. Figure 9 shows the object graph when entering migrateTransition(trans, node, boolean) the second time. A state mapping remembers that a16 has been transformed already.

In the following, the input graph is now traversed following all transitions between state vertices in a depth-first manner. We remember already transformed states with state mapping instances, so a path joining an already traversed path will terminate the transformation for that branch. Of course, this approach relies on valid input models with exactly one initial node, but checking the input model beforehand was out of scope of this work.

Figure 10 shows the central transformation rule: the left part determines the concrete instance type of the state and calls the corresponding migrateState(...) method of the handler. Furthermore, the method creates the additional flow instance and links it to it’s source and target node. It converts also the transition guard. Finally it creates a state mapping. Transformations for the individual UML1.4 state types are handled in separate methods of the handler. As an example, figure 11 shows the transformation of an object flow.

Figure 11: migrate an object flow state

Figure 12 shows a more complex graph rewrite rule: It’s responsible for migrating the different Pseudostate kinds to a corresponding activity node.

5. EXTENSIONS
5.1 Alternative Object Flow State Migration

In contrast to the behavior specified in the StateHandler.migrateState(ObjectFlow...) method (cf. figure 11), our algorithm can also be run with the ExtendedStateHandler, which overrides this method and specifies a different transformation: Figure 13 shows the Story diagram of the alternative method. Figure 14 shows the object graph surrounding the "Delivered Order" object flow state. When executing the alternative transformation, o0 is mapped to the sourceState 'variable'. Its predecessor is the action state a8. The object graph shows that this state has been migrated already, indicated by the state mapping s10 to o4, the corresponding source UML2.2 opaque action. In this method, a new object flow edge instance gets created. The method skips the transition t1 by invoking migrateTransition(trans, node, true) directly, so the algorithm will continue with transition t3. Furthermore, the edge gets a new activity parameter node as type, preserving the source state's name. The result of this method can be seen in figure 15. As you can see by the mapping links, that the 'middle' object o0 is migrated by a typed object flow edge.

5.2 XMI Import and Export

Importing and exporting XML was implemented by hand using the JDOM library. The import/export code adapts to the Fujaba generated model elements and is as small as about 100 LOC each. Besides these classes, everything else was modeled using Fujaba Story diagrams. The Importer for XMI1.2 and XMI2.0 uses a reflective mechanism and is independent of the meta model to be loaded, although schema checking etc. has not been implemented. By using reflective mechanisms, it is able to instantiate both UML1.4 and UML2.2 models without introducing a compile time dependency to those. The XMLExporter simply traverses the activity graph structure and writes the corresponding XML tree to disk. See the online solution via SHARE [3] for full listing of the XMI import/export source code.

6. SUMMARY AND OUTLOOK

This case study was a nice exercise for our Fujaba Case Tool. The transformation rules are intuitive and the iterative approach is easy to understand. Our approach reduces the problem down to eight graph transformation rules (not counting helper methods), which take an input structure fragment and create the corresponding output. The Importer and additional object graph creators show that the migration transformation works correctly. The development of the transformation was greatly supported by the eDOBS [2] eclipse plugin in combination with the Design Level Debugging functionality, which visualizes the runtime object graphs, so one can follow the execution of the transformation in detail by stepping with the debugger and watch how the graph is modified.

7. REFERENCES


Figure 6: Sample input model/graph as eDOBS view
Figure 10: Migration transition method
Figure 12: Migration of an pseudo state: mapping an enum to different instances