Model Migration with MOLA

Elina Kalnina, Audris Kalnins, Janis Iraids, Agris Sostaks, Edgars Celms
University of Latvia, IMCS, Raina bulvaris 29, LV-1459 Riga, Latvia
Elina.Kalnina@lumii.lv, Audris.Kalnins@lumii.lv, Edgars.Celms@lumii.lv,
Agris.Sostaks@lumii.lv, Janis.Iraids@lumii.lv

Abstract. This paper describes the activity diagram migration from UML 1.4 to UML 2.2 in MOLA transformation language. Transformations implementing the migration task are relatively straightforward and easily inferable from the task specification. The required additional steps related to model import and export are also described.

1 Introduction

In this paper we describe the solution to Model migration case [1] for TTC 2010 contest, as implemented in MOLA model transformation language. The core migration task is implemented. We take as input the UML 1.4 activity models built according to the provided minimal UML 1.4 metamodel (original minimal metamodel). The target models are created according to the provided (complete) UML 2.2 metamodel.

The main point of discussion in this task is how to migrate object flows, where the best semantics preserving transformation is not so unique. We have chosen to substitute object flow states in UML 1.4 by pins in UML 2.2. This seems to be the most natural choice according to UML 2.2 specifications [2], used also in several commercial UML tools. Thus, we convert an object flow state together with incoming and outgoing transitions into a pin with incoming and outgoing object flows respectively.

We build the target model in a way that it can be imported into Eclipse with the UML 2 plug-in and visualized as a diagram, in order to check the transformation correctness more easily. This feature requires some classes to be added to the model, therefore we have chosen the complete evolved metamodel for the target, but not the minimal one.

The migration task can be implemented in MOLA in a very straightforward way once the correspondence between metamodel elements is well understood. We describe in the paper the basic principles of the solution. Before that, the situation with metamodels is described in some details, namely, how the metamodels are imported and extended and what implications this creates for source model import and target model export. The whole migration process using MOLA is briefly described as well.

1 http://planet-research20.org/lttc2010/
2 MOLA Environment

MOLA [3] is a graphical transformation language developed at the University of Latvia. It is based on traditional concepts among transformation languages: pattern matching and rules defining how the matched pattern elements should be transformed. The formal description of MOLA and also MOLA tool can be downloaded in [4].

A MOLA program transforms an instance of a source metamodel (defined in the MOLA metamodelling language – MOLA MOF, close to EMOF) into an instance of a target metamodel.

Rule contains a declarative pattern that specifies instances of which classes must be selected and how they must be linked. Pattern in a rule is matched only once. The action part of a rule specifies which matched instances must be changed and what new instances must be created. The instances to be included in the search or to be created are specified using class elements in the MOLA rule. The traditional UML instance notation (instance_name:class_name) is used to identify a particular class element. Class elements may contain constraints and attribute assignments defined using simple OCL like expressions. Additionally, the rule contains association links between class elements. Class elements matched in one rule may be referenced in another one using the reference element (prefixed with "@" symbol).

In order to iterate through a set of the instances MOLA provides the foreach loop statement. The loophead is a special kind of a rule that is used to specify the set of instances to be iterated over. The pattern of the loophead is given using the same pattern mechanism used by an ordinary rule, but with an additional important construct. It is the loop variable—the class element that determines the execution of the loop. The foreach loop is executed for each distinct instance that corresponds to the loop variable and satisfies the constraints of the pattern. In fact, the loop variable plays the same role as an iterator in classical programming languages. The execution order in MOLA is specified in a way similar to UML activity diagrams.

MOLA has an Eclipse-based graphical development environment (MOLA tool [4]), incorporating all the required development support. A transformation in MOLA is compiled via the low-level transformation language L3 [5] into an executable Java code which can be run against a runtime repository containing the source model. For this case study Eclipse EMF is used as such a runtime repository, but some other repositories can be used as well (e.g. JGraLab [6]).

3 General Principles of Migration Case Solution with MOLA

The transformation development in MOLA starts with the development of metamodels. The MOLA tool has a facility for importing existing metamodels, in particular, in EMF (Ecore) format. Though MOLA metamodelling language (MOLA MOF) is very close to EMOF, and consequently Ecore, there are some issues to be solved. The current version of MOLA requires all metamodel associations to be navigable both ways (this permits to perform an efficient pattern matching using simple matching algorithms). Since a typical Ecore metamodel has many associations navigable one way, the import facility has to extend the metamodel. Another issue is
the variable coding of references to primitive data types (in this case the coding within source and target metamodels was also different).

Metamodel import facilities in MOLA are able to perform all these adjustments automatically. This way both the source (original minimal) and target (evolved) metamodels were imported into MOLA tool. Efficient transformation development in MOLA for model mapping related tasks, as the current one is, requires additional metamodel elements for storing the mapping between the source and target model elements. These elements have to be added manually. In the given case, one association between the top classes (in the inheritance hierarchy) of both metamodels is sufficient. Now the transformation itself (MOLA procedures) can be developed. The key features of transformations are described in the next section. The development ends with MOLA compilation.

Since the metamodels have been modified during import, the original source model does not conform directly to the metamodel in the repository, mainly due to added association navigability. Therefore a source model import facility is required. MOLA execution environment (MOLA runner) includes a generic model import facility, which automatically adjusts the imported model to the modified metamodel. Now the migration transformation can be run on the model. Similarly, a generic export facility automatically strips all elements of the transformed model which does not correspond to the original target metamodel. Thus a transformation result is obtained which directly conforms to the target metamodel. The transformation user is not aware of these generic import and export facilities, he directly sees the selected source model transformed.

4 MOLA Transformations for the Migration

The given migration task is very adequate for implementation in MOLA. The key issue in the transformation design is to find out which source model elements must be converted to which target model elements. The best way is to define an informal mapping, such as:

ActivityGraph/CompositeState -> Activity
ActionState -> OpaqueAction
ObjectFlowState -> Pin
Pseudostate (kind=initial) -> InitialNode
...
Transition -> ObjectFlow (when connected to ObjectFlowState)
Transition -> ControlFlow (otherwise)
and so on.

Now simple MOLA procedures can be built which directly implement the mappings. The transformation process has to be started from the top elements in the containment hierarchy, in the given case from ActivityGraph. All elements of a container, here CompositeState, are transformed using a MOLA foreach loop, running over these elements (see Fig. 1). Certainly, we assume here that an activity graph contains just one CompositeState. Further, the procedure State is just a “dispatcher”
which finds out what kind of state is really represented by the current StateVertex and invokes the corresponding transformation.

The transformation of an individual state according to the selected mapping is very straightforward, for example, the transformation of ActionState is shown in Fig. 2. Note that along with the target element (OpaqueAction) its mapping to the corresponding source element (the link sourceElement/targetElement) is built.

When the nodes have been transformed, edges can be processed. Finding of end points of an edge to be created is based directly on mappings from the end points of the source edge. Finally, the partitions are created and transformed nodes are attached to the corresponding partitions (again using the mappings).

The complete set of transformation procedures is given in the appendix.
5 Conclusions

This migration case study has been very appropriate to be implemented in MOLA. The hierarchical model structure and relatively simple mappings from source to target model permitted a quite straightforward implementation by transformations in MOLA. The most complicated task was to define precisely this source-target element mapping, because of possible variations in understanding the correspondence between UML 1.4 and 2.2. Once defined, the mapping fits well into MOLA capabilities, the structure of the complete transformation given in the appendix corresponds directly to that mapping. The effort for transformation development was quite low. The required infrastructure for model management within the MOLA tool also was sufficient for the case though some less typical situations revealed a couple of deficiencies.

Certainly, a question can be raised whether such model migration task, easily to be specified by appropriate mappings, couldn’t be solved more formally on the basis of these mappings. There exist some mapping based approaches (see e.g. [7, 8]) where ATL transformations can be generated by higher order transformations from a sort of mappings. However, it seems that expressiveness of these approaches could be insufficient for natural and complete specification of mappings for this case. Therefore we preferred an informal definition of the mapping with manual implementation in MOLA.

References

2. OMG, Unified Modeling Language: Superstructure, version 2.2, formal/09-02-02, 2009
6. Universität Koblenz-Landau, Institute for Software Technology, Graph Laboratory http://www.uni-koblenz-landau.de/koblenz/itb4/institute/IST/AGEbert/MainResearch/ Graphentecnologie/graph-laboratory-grafalab
Appendix A: Transformation sources.

In this appendix transformation sources will be described.

In Figure 3 the main transformation procedure is demonstrated. Since in the source (minuml1) metamodel there is no Package hierarchy we simply created one “root” package in the target metamodel. To include a possibility to copy the source Package hierarchy we should change this procedure by adding a support for hierarchy.

This procedure calls the procedure Activity (see Fig. 4). It iterates through all ActivityGraphs (though most probably the source model will contain just one graph). For each ActivityGraph one UML 2.2 Activity in the root package is created. Then the transformation finds the Composite state with top association. This state contains all states of the Activity diagram. The first step is to copy all states (see Fig. 5.). The second step is to copy all Transitions (see Fig. 11.). Finally Partitions are created and states are added to appropriate Partitions (see Fig. 13.).

![Fig. 3. Main transformation procedure. Root package is created and activity cloning is invoked.](image)
Fig. 4. Procedure Activity iterating through all ActivityGraphs in UML 1.4 and creating Activities in UML 2.2.

Copying of States is done rather special way. A general state processing mechanism is used for implementing the specified mapping between state kinds in UML 1.4 and 2.2. At first the state kind is determined (see Fig. 5). The type of the top node is CompositeState. For it the procedure processing composite states is called (Fig. 6). In this example it is assumed that a composite state can be used only as a top node. Therefore all nodes within a composite state are copied and placed directly in the Activity. For each node its type is determined (see Fig. 5) and the appropriate copier is called. While copying nodes the mapping association between the old and new nodes is created. The FinalState corresponds to ActivityFinalNode (Fig. 7). The ActionState corresponds to OpaqueAction (Fig. 8).
Fig. 5. Procedure State performing the state cloning. The state type is determined and the appropriate transformation is called.
Fig. 6. Procedure CompositeState. The transformation copies all nodes contained in a Composite state and places them in the Activity.

Fig. 7. Procedure FinalState transforming final state to ActivityFinalNode.
In Figure 9 the processing of Pseudostates is described. Depending on Pseudostate kind the Node type to be created is determined. If the kind is initial an InitialNode is created. If the kind is join a JoinNode is created. If the kind is fork a ForkNode is created. If the kind is junction a DecisionNode is created. UML 2.2 has also a MergeNode but since the role of a junction is not differentiated in UML 1.4 always a DecisionNode is created for a Junction.

An ObjectFlowState is transformed to a Pin (see Fig. 10). In the updated task specification it was recommended to transform an ObjectFlowState to an ActivityParameterNode. As it was already stated in the introduction a Pin represents the semantics of ObjectFlow state more precisely. Visualization in the Eclipse UML tool is also much better.
Fig. 9. Procedure PseudoState. Type of the node created depends on the Pseudostate kind. If kind is initial an InitialNode is created. If kind is join a JoinNode is created. If kind is fork a ForkNode is created. If kind is junction a DecisionNode is created.
Fig. 10. Procedure ObjectFlow transforming an ObjectFlowState to a Pin.

When all states are copied the transitions are processed. The transformation iterates through all transitions in an activity graph (see Fig. 11.). It processes each transition and determines the Edge type which should be created (see Fig. 12 for transition processing). If source or target of a Transition is an ObjectFlowState then an ObjectFlow is created. Otherwise a ControlFlow is created. Source and target nodes of the new edge are determined using the mapping association between nodes in UML 1.4 model and UML 2.2 model. The same mapping association is created between transitions and edges. Edges are also linked to the Activity.

If the transition has a guard the appropriate guard is added also to the edge in UML 2.2 (see the final rule in Fig. 12.).
Fig. 11. Procedure Transitions finds all transitions in an ActivityGraph and processes them.
Fig. 12. Procedure transition processing one Transition. If one end of the Transition is ObjectFlow state an ObjectFlow is created. Otherwise a ControlFlow is created.

The final step of the transformation is to process partitions. The procedure in Fig. 13 iterates through all Partitions in the ActivityGraph and creates a new ActivityPartition in the Activity for each. In Fig. 14 a loop iterates through all StateVertex (and Transitions) instances in this Partition, finds the appropriate Node (Edge) in the target model and adds it to the new ActivityPartition.
Fig. 13. Procedure Partition iterates through all partitions in the UML 1.4 model and creates an appropriate ActivityPartition in UML 2.2.
Fig. 14. Procedure PartitionState adding States and Transitions to appropriate Partitions.