Abstract and Concrete Syntax Migration of Instance Models

Antonio Cicchetti[1], Bart Meyers[2], and Manuel Wimmer[3]

[1] Mälardalen University, MRTC, Västerås, Sweden
antonio.cicchetti@mdh.se
[2] University of Antwerp, Belgium
bart.meyers@ua.ac.be
[3] Vienna University of Technology, Austria
wimmer@big.tuwien.ac.at

Abstract. In this paper, we present a solution for the TTC 2010 model migration case study. Firstly, we present a modular approach to migrate the instance models' abstract syntax. Secondly, the problem of co-evolution of diagrammatical information such as icon positions and bend points of edges is identified and a solution specific to this case study is presented. Our solution implemented using ATL and Java.

Key words: Metamodel evolution, model co-evolution, inplace model transformations, model merging

1 Goal and approach

This paper presents a solution to the TTC 2010[4] model migration case study. This case study explores the consequences an evolution of a modelling language can have with respect to its instance models. In particular, these models must be migrated so that (i) they conform to the new language, and (ii) their semantics are preserved. The presented language evolution is UML 1.4 Activity Graphs to UML Activity Diagrams 2.2. The main goal of the case study is shown in the diagram of Figure 1. In our approach, a structured migration process is pursued by first modelling the evolution Δ itself into manageable parts, followed by a migration of each part resulting in a migration transformation M. The explicit breakdown of Δ helps us in understanding the evolution and finding a correct, semantically preserving migration M.

After migrating a model m using M, the diagrammatical information of m is lost. Specific to this case study, this results in a model m' for which the original positions of the icons are lost. This makes the migrated models less readable, as people tend to arrange the model icons in a way that is natural to them. Therefore, we decided to migrate the concrete syntax information or diagram model as well. Figure 2 shows the new diagram that includes this second goal. Apart from migrating models m as in Figure 1, the diagram model m_{CS} must

[4] http://planet-research20.org/ttc2010/

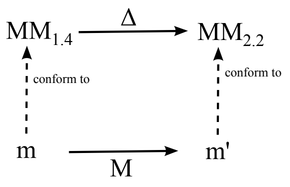


Fig. 1: The model migration case study with the evolution Δ and the migration M.

also be migrated by a migration transformation M_{CSinst}. The concrete syntax can be obtained by so-called rendering, which can be considered a transformation in which the user adds diagrammatical information, such as icons (at the level of the language concepts) and positions of these icons (at the level of the language concept instances). As suggested in Figure 2, we can make some assumptions for this case study: (i) the metamodels of the diagram model m_{CS} and its migrated counterpart m'_{CS} are the same: $MM_{GMF-notation}$ – GMF notation 1.0.2, and (ii) the rendering transformation itself must not be migrated, as in this case study, it is part of the GMF-based graphical editor which has evolved itself too (captured as the evolution Δ_{CS}, from $render_{1.4}$ to $render_{2.2}$). So in this case study, the migration of the concrete syntax MM_{CSinst} can be derived from the evolution Δ and Δ_{CS} (which in practice evolved together), and does not have to take a change of the metamodel $MM_{GMF-notations}$ into account. In conclusion, in this case study the concrete syntax co-evolution is simplified. However, this solution can start a discussion about the general topic of concrete syntax co-evolution.

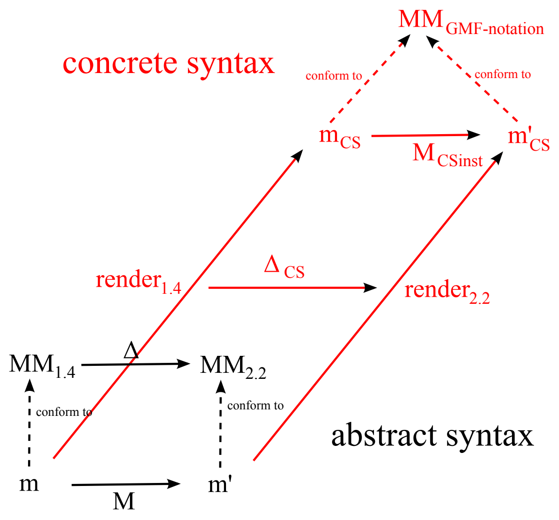


Fig. 2: MM_{CSinst}, The co-evolution of the concrete syntax instances.

2 Solution

In this section, we present an overview of our solution for the case study, consisting of the two parts: migration of instance models and migration of diagram models.

2.1 Instance Model Migration

As stated before, we structurally tackle the problem of instance model migration by starting off from the metamodel evolution and model co-evolution, as in the following:

metamodel evolution detection that is the old and new versions of the metamodel are compared in order to synthesise the evolution it has been subject to. This will result in a breakdown into manageable evolutionary steps;

modular migration creation for every evolutionary step, a migration activity is created (typically in the form of a transformation rule).

Our solution is based on the manual declaration of evolutionary steps; such choice guarantees that the intentions of the metamodel developer are fully captured. It is worth noting that the same result could be obtained by a tool recording the changes made by the user. In any case, in general the metamodels are remarkably smaller and more manageable than model instances, which makes even the manual specification of the evolution an acceptable effort if compared to verifying the migration correctness of existing instances.

We broke down the evolution into eleven evolutionary steps. For each step, the ATL migration rule is given. In the first two steps, a short explanation is given for the migration action; for the other rules, the migration is just a straightforward replacement of corresponding instances:

1. ActivityGraph, StateMachine, and CompositeState are merged into Activity. Subvertex and partition containers are merged into groups container to the new class ActivityGroup;
   migrate_TopCompositeState, *migrate_SubCompositeStates*
2. Transition becomes ActivityEdge with two subclasses, ControlFlow and ObjectFlow. In case of a surrounding ObjectFlowState, the transition becomes ObjectFlow, otherwise ControlFlow;
   migrate_Transition_ControlFlow, *migrate_Transition_ObjectFlow*
3. StateVertex and State are merged into ActivityNode;
   migrate_State
4. Pseudostate(kind:initial) becomes InitialNode;
   migrate_PseudoState_Initial
5. Pseudostate(kind:join) becomes JoinNode;
   migrate_PseudoState_Join
6. Pseudostate(kind:fork) becomes ForkNode;
   migrate_PseudoState_Fork

7. Pseudostate(kind:junction) is split into DecisionNode and MergeNode; *migrate_PseudoState_Junction*
8. FinalState becomes ActivityFinalNode; *migrate_FinalState*
9. ActionState becomes OpaqueAction; *migrate_ActionState*
10. Partition becomes ActivityPartition as subclass of ActivityGroup; *migrate_Partitions*
11. Guard becomes ValueSpecification and the contained BooleanExpression becomes the subclass of ValueSpecification called OpaqueExpression. *migrate_Guard*

2.2 Concrete Syntax Migration

The concrete syntax of UML models is not standardized like the abstract syntax. In our solution we used the Eclipse UML 2 tool suite[5], which includes a graphical editor for UML 2 models. The diagrammatical information this editor can read and write will be the target platform for migrating the concrete syntax of a model. On the other hand however, the Activity Graph 1.4 source models cannot be read by the UML 2 tool. Because we only want to use Eclipse, we built a simple UML 1.4 editor using GMF, that allows creating a model and position icons. The Activity Graph 1.4 example visualized by our simple editor is shown in Figure 3. After migration, the diagram model should be visualized in Eclipse as in Figure 4.

In order to obtain the correct result, the icon positions and bend points of edges must be preserved after migrating the abstract syntax as explained in Section 2.2. This requires a simple copying of the coordinates and bend points. However, it must be made sure that the copying of this information is done to the right element, a not obvious task given the exploitation of UUIDs. In other words, the graphical arrangement is saved in a XMI-like document by referring to model elements through UUIDs, whereas the ATL transformation is agnostic of them as working at the metamodel level. Therefore, trace information has to be created in order to link elements of the old diagram and the corresponding migrated ones of the new diagram. In particular, when the co-adapting ATL transformation is executed, a simple trace model is filled in with a list of (source, target) pairs storing the links between elements of the old and new metamodel instances.

In our solution, this is done by simply adding the code that creates the trace model in the ATL transformation. In fact, the addition of such code can also be done automatically, by using a transformation that adapts the ATL transformation. Because the input and output models of this transformation are transformations themselves, such a transformation is called a higher order transformation. Due to time limitations, we did not implement this higher order

[5] `http://www.eclipse.org/projects/project_summary.php?projectid=modeling.emf`

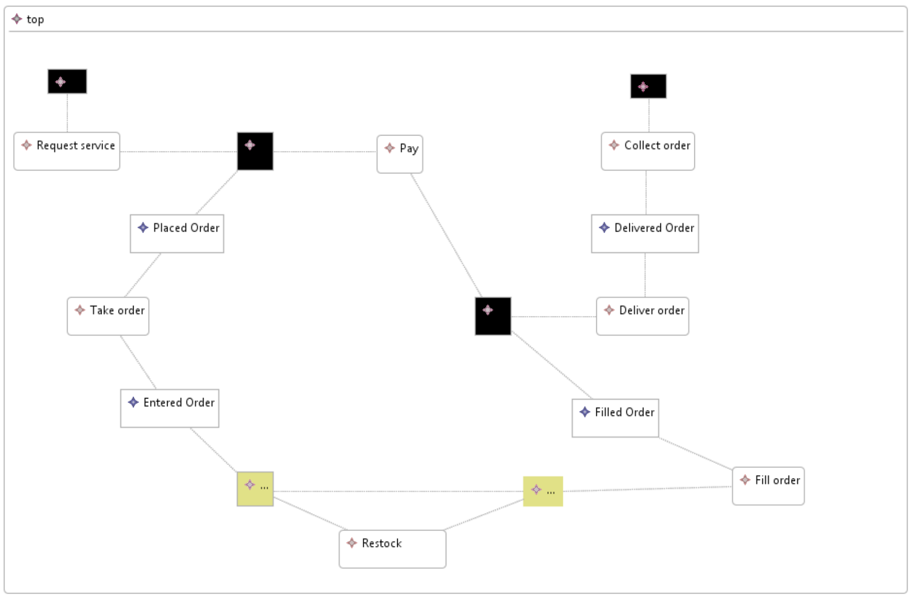


Fig. 3: The original model visualized by the simple GMF editor.

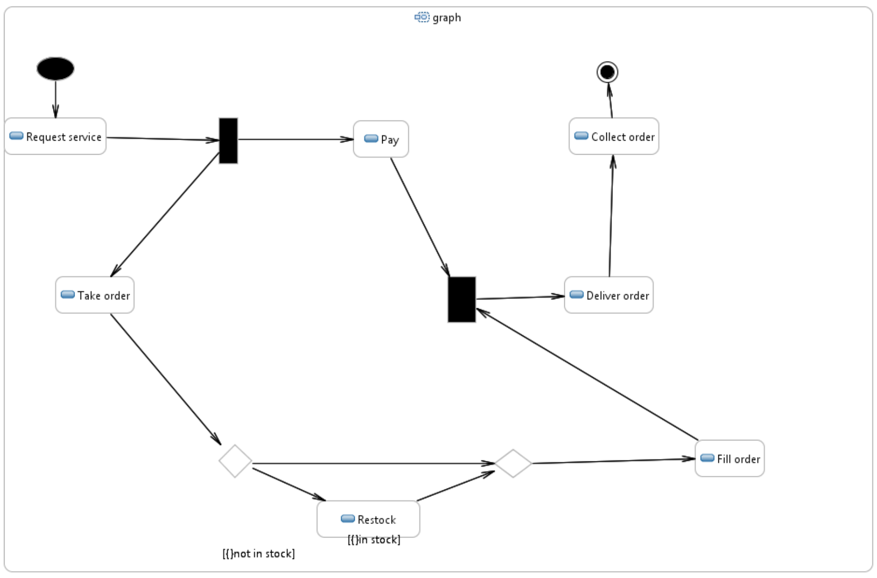


Fig. 4: The migrated model visualized by Eclipse UML 2.

transformation yet. It would improve automation, as well as clearly separating the code for creating the traceability model from the code for creating the migrated model. This lowers accidental complexity, increasing the overall quality (readability, maintainability, etc.) if the transformation models.

From the migrated instance model, the original instance model, the original diagram model and the trace model, the new diagram model can be automatically generated. For each element in the migrated instance model, its concrete syntax is generated by tracing back to the original element(s) using the trace model. The corresponding concrete syntax information which is obtained by using the `href` fields that point to the instance model elements are copied in the migrated diagram model. This approach is shown in Figure 5, where the links from the original diagram model to the migrated diagram model are shown from top to bottom.

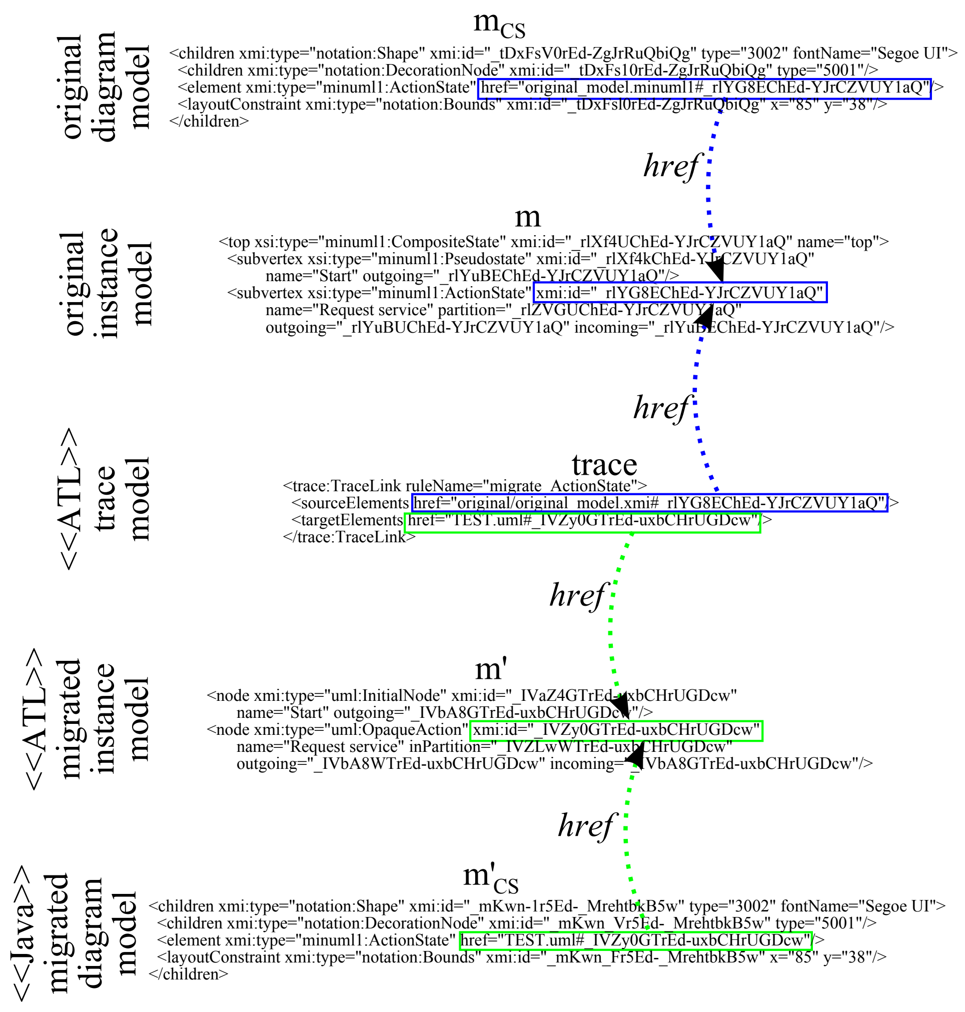
3 Conclusion and Discussion

In our solution we presented a structured way of migrating the instance model by splitting up the evolution into manual parts. For every evolution step that emerges, (a) simple migration rule(s) can be created. The total migration transformation is implemented in ATL.

In order to keep the concrete syntax information of a model, the diagram model is also migrated. For this case study, this includes the preservation of icon positions and bend points. These can be simply preserved by copying them. However, in order to find the right elements for the copied information, a trace model is needed that links elements from the original instance model to the migrated instance model. This trace model is created by a part of the migration transformation.

The complexity of concrete syntax migration When looking at Figure 2, in this case study the concrete syntax migration was simplified in two ways: the concrete syntax metamodel remains the same (i.e., $MM_{GMF-notation}$) and the evolution of the editor does not have to be taken into account, as it is done manually. In the more general case of concrete syntax migration however, these assumptions cannot be made. In general, the migration of concrete syntax will entail two migration actions:

- the migration of the concrete instance models. We have done this for this case study. In the more general case however, the metamodels need not be the same. As a result, the diagram model will have to migrated across two dimensions: the conformance with the abstract syntax model, and the conformance with its metamodel;
- the migration of the rendering transformation. This can be considered the migration of the “editor”.



```xml
<children xmi:type="notation:Shape" xmi:id="_tDxFsV0rEd-ZgJrRuQbiQg" type="3002" fontName="Segoe UI">
  <children xmi:type="notation:DecorationNode" xmi:id="_tDxFs10rEd-ZgJrRuQbiQg" type="5001"/>
  <element xmi:type="minuml1:ActionState" href="original_model.minuml1#_rlYG8EChEd-YJrCZVUY1aQ"/>
  <layoutConstraint xmi:type="notation:Bounds" xmi:id="_tDxFsl0rEd-ZgJrRuQbiQg" x="85" y="38"/>
</children>
```

```xml
<top xsi:type="minuml1:CompositeState" xmi:id="_rlXf4UChEd-YJrCZVUY1aQ" name="top">
  <subvertex xsi:type="minuml1:Pseudostate" xmi:id="_rlXf4kChEd-YJrCZVUY1aQ"
        name="Start" outgoing="_rlYuBEChEd-YJrCZVUY1aQ"/>
  <subvertex xsi:type="minuml1:ActionState" xmi:id="_rlYG8EChEd-YJrCZVUY1aQ"
        name="Request service" partition="_rlZVGUChEd-YJrCZVUY1aQ"
        outgoing="_rlYuBUChEd-YJrCZVUY1aQ" incoming="_rlYuBEChEd-YJrCZVUY1aQ"/>
```

```xml
<trace:TraceLink ruleName="migrate_ActionState">
  <sourceElements href="original/original_model.xmi#_rlYG8EChEd-YJrCZVUY1aQ"/>
  <targetElements href="TEST.uml#_IVZy0GTrEd-uxbCHrUGDcw"/>
</trace:TraceLink>
```

```xml
<node xmi:type="uml:InitialNode" xmi:id="_IVaZ4GTrEd-uxbCHrUGDcw"
      name="Start" outgoing="_IVbA8GTrEd-uxbCHrUGDcw"/>
<node xmi:type="uml:OpaqueAction" xmi:id="_IVZy0GTrEd-uxbCHrUGDcw"
      name="Request service" inPartition="_IVZLwWTrEd-uxbCHrUGDcw"
      outgoing="_IVbA8WTrEd-uxbCHrUGDcw" incoming="_IVbA8GTrEd-uxbCHrUGDcw"/>
```

```xml
<children xmi:type="notation:Shape" xmi:id="_mKwn-1r5Ed-_MrehtbkB5w" type="3002" fontName="Segoe UI">
  <children xmi:type="notation:DecorationNode" xmi:id="_mKwn_Vr5Ed-_MrehtbkB5w" type="5001"/>
  <element xmi:type="minuml1:ActionState" href="TEST.uml#_IVZy0GTrEd-uxbCHrUGDcw"/>
  <layoutConstraint xmi:type="notation:Bounds" xmi:id="_mKwn_Fr5Ed-_MrehtbkB5w" x="85" y="38"/>
</children>
```

Fig. 5: The models are linked through trace links.