The Edapt Solution for the GMF Model Migration Case

Markus Herrmannsdoerfer

Institut für Informatik, Technische Universität München
Boltzmannstr. 3, 85748 Garching b. München, Germany
herrmama@in.tum.de

Abstract. This paper gives an overview of the Edapt solution to the GMF model migration case of the Transformation Tool Contest 2011.

1 Edapt in a Nutshell

Edapt[1] is a transformation tool tailored for the migration of models in response to metamodel adaptation.

Modeling the Coupled Evolution. As depicted by Figure 1, Edapt specifies the metamodel adaptation as a sequence of operations in an explicit history model. The operations can be enriched with instructions for model migration to form so-called coupled operations. Edapt provides two kinds of coupled operations according to the automatability of the model migration [1]: reusable and custom coupled operations.

Reuse of recurring migration specifications allows to reduce the effort associated with building a model migration [2]. Edapt thus provides *reusable coupled operations* which make metamodel adaptation and model migration independent of the specific metamodel through parameters and constraints restricting the applicability of the operation. An example for a reusable coupled operation is *Enumeration to Sub Classes* which replaces an enumeration attribute with subclasses for each literal of the enumeration. Currently, Edapt comes with a library of over 60 reusable coupled operations [3]. By means of studying real-life metamodel histories, we have shown that, in practice, most of the coupled evolution can be covered by reusable coupled operations [2, 4].

Migration specifications can become so specific to a certain metamodel that reuse makes no sense [2]. To express these complex migrations, Edapt allows the user to define a custom coupled operation by manually encoding a model migration for a metamodel adaptation in a Turing-complete language [5]. By softening the conformance of the model to the metamodel within a coupled operation, both metamodel adaptation and model migration can be specified as in-place transformations, requiring only to specify the difference. A transaction mechanism ensures conformance at the boundaries of the coupled operation.

[1] http://www.eclipse.org/edapt

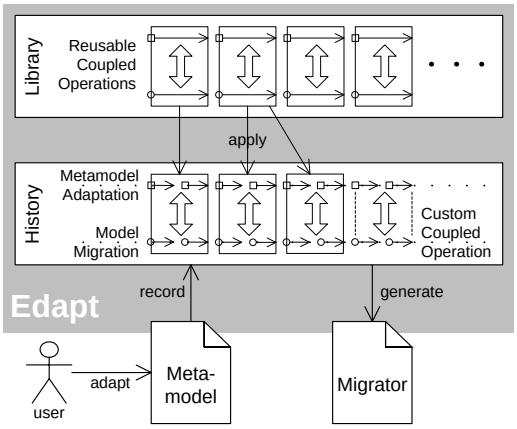

Fig. 1. Overview of Edapt

Recording the Coupled Evolution. To not lose the intention behind the metamodel adaptation, Edapt is intended to be used already when adapting the metamodel. Therefore, Edapt's user interface which is depicted in Figure 2 is directly integrated into the existing EMF *metamodel editor*. The user interface provides access to the *history model* in which Edapt records the sequence of coupled operations. An initial history can be created for an existing metamodel by invoking *Create History* in the *operation browser* which also allows the user to *Release* the metamodel.

The user can adapt the metamodel by applying reusable coupled operations through the *operation browser*. The operation browser allows to set the parameters of a reusable coupled operation, and gives feedback on its applicability based on the constraints. When a reusable coupled operation is executed, its application is automatically recorded in the history model. Figure 2 shows the operation Sub Classes to Enumeration being selected in the operation browser and recorded to the history model.

The user needs to perform a custom coupled operation only, in case no reusable coupled operation is available for the change at hand. First, the metamodel is directly adapted in the metamodel editor, in response to which the changes are automatically recorded in the history. A migration can later be attached to the sequence of metamodel changes. Figure 2 shows the *migration editor* to encode the custom migration in Java.

2 GMF Model Migration Case

The complete solution is available through the repository of the Eclipse Edapt project[2]. Here, we only briefly describe the main characteristics of the solution.

[2] `http://dev.eclipse.org/svnroot/modeling/org.eclipse.emft.edapt/trunk/examples/ttc_gmf`

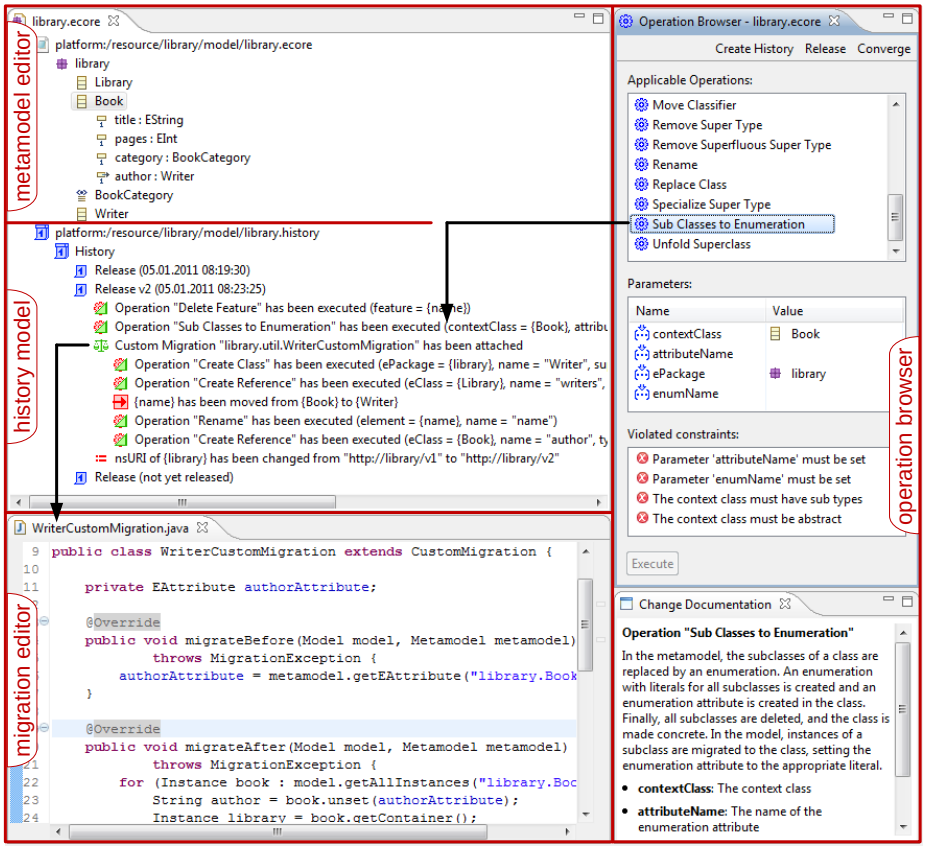

Fig. 2. User interface of Edapt

Core Task. To build the history model for the GMF Graph metamodel, we took the intermediate metamodel versions from the case resources and applied coupled operations to get from one metamodel version to the next. To know which coupled operations to apply, we used the difference model obtained with EMF Compare[3] between two subsequent metamodel versions. Edapt provides the so-called *Convergence View* which automatically updates the difference model after each application of an operation [6].

Figure 3 shows the screenshot of the history model from release 1.0 to 2.1. The history model also contains markers for the different metamodel versions. Table 1 lists the employed coupled operations together with their number of applications. Most of the evolution can be covered by reusable coupled operations. For more information about the reusable coupled operations, we refer the reader to [3]. Only two custom coupled operations are necessary to specify a correct model migration. The first custom migration is necessary to initialize

[3] `http://www.eclipse.org/emf/compare/`

Table 1. GMF Graph

Operation	Kind	Number
Add Super Type	Reusable	10
Change Namespace URI	Reusable	1
Create Attribute	Reusable	2
Create Class	Reusable	4
Create Reference	Reusable	5
Delete Feature	Reusable	4
Delete Operation	Reusable	1
Document Metamodel Element	Reusable	12
Drop Attribute Identifier	Reusable	1
Extract Super Class	Reusable	3
Generalize Attribute	Reusable	1
Generalize Reference	Reusable	1
Inline Super Class	Reusable	2
Make Class Abstract when Interface	Reusable	14
Make Reference Containment	Reusable	1
Not Changeable to Suppressed Set Visibility	Reusable	1
Push Down Feature	Reusable	4
Remove Super Type	Reusable	10
Specialize Reference Type	Reusable	4
Specialize Super Type	Reusable	6
Suppressed Set Visibility to Not Changeable	Reusable	1
Unfold Super Class	Reusable	1
Initialize FigureAccessor.typedFigure	Custom	1
Decouple FigureHandle.referencingElements	Custom	1

a reference that was made mandatory. The second custom migration decouples the diagram elements from the figures to make the figures reusable. The custom migrations which are implemented in Java can be found in the appendix of this paper.

Multi-File Models. Edapt can migrate multi-file models, as it preserves the modularization into files throughout the migration. If one file refers to the other, the second file is automatically loaded when resolving references. However, a user can also explicitly define that several files are migrated together.

GMF Map metamodel. To build the history model for GMF Map metamodel, we applied the same procedure as in case of the GMF Graph metamodel. The screenshots of the resulting history model are depicted in Figures 4 and 5: Figure 4 shows the coupled operations from release 1.0 to 2.0, and Figure 5 the operations from release 2.0 to 2.1. Table 2 lists the employed coupled operations together with their number of applications. The complete coupled evolution can be covered by reusable coupled operations. For more information about the reusable coupled operations, we refer the reader to [3].

Fig. 3. History model for the GMF Graph metamodel

Table 2. GMF Map

Operation	Kind	Number
Change GMF Constraint	Reusable	5
Change Namespace URI	Reusable	3
Create Annotation	Reusable	1
Create Attribute	Reusable	3
Create Class	Reusable	2
Create Enumeration	Reusable	3
Create GMF Constraint	Reusable	6
Create Reference	Reusable	3
Delete Annotation	Reusable	2
Document Metamodel Element	Reusable	12
Extract Subclass	Reusable	1
Extract Super Class	Reusable	1
Inheritance to Delegation	Reusable	1
Make Class Abstract when Interface	Reusable	10
Make Feature Volatile	Reusable	1
Move Annotation	Reusable	1
Not Changeable to Suppressed Set Visibility	Reusable	5
Push down Feature	Reusable	2
Replace Enumeration	Reusable	2
Suppressed Set Visibility to Not Changeable	Reusable	5

As one can see, the history model is modularized into different releases. Based on the namespace URI, Edapt automatically detects of which release a model is and applies the coupled operations from this release to the newest release. Moreover, the history model can be recorded for metamodels that refer to other metamodels. However, to be able to load the models, the other metamodels need to be available during migration. In the solution, we ensured this by starting the migration in an Eclipse workbench where the other metamodels are available through plugins.

3 Conclusion

Expressiveness. The language provided by Edapt is expressive enough to specify the migration. Most of the migration can be covered by reusable coupled operations. For more complex migrations, custom coupled operations can be specified in Java. The API provided to specify the custom coupled operations turned out to be expressive enough to cover the complex migrations involved in the GMF Graph metamodel evolution.

Correctness. To ensure correctness, we used the test models from the case resources. We modified the history model until there are no differences between the migrated model and the expected model. To obtain the differences, we used

Fig. 4. History model for the GMF Map metamodel (release 1.0 - 2.0)

Fig. 5. History model for the GMF Map metamodel (release 2.0 - 2.1)

EMF Compare. We ignored differences in the order of references, since we believe that the order is not important for the GMF Graph and Map models.

Conciseness. The migration specification is rather concise, since most of the evolution can be covered by reusable coupled operations. Only two custom coupled operations are required: one with 21 and another one with 90 lines of custom migration code. Considering the complexity of the second migration, we consider the migration specification as concise. However, conciseness could be improved by providing a DSL instead of a Java API to implement migrations.

Maintainability. The history model is easy to understand, since it is modularized into rather small coupled operations. Intermediate metamodel versions can be reconstructed from the history model, thus simulating the evolution. Moreover, coupled operations can be undone to revert changes that have been performed erroneously, and custom migrations can be removed, edited and added to fix error-prone migrations. Finally, existing history models can be extended by recording new coupled operations that are applied to the metamodel.

Acknowledgments. This work was funded by the German Federal Ministry of Education and Research (BMBF), grants "SPES2020, 01IS08045A" and "Quamoco, 01IS08023B".

References

1. Herrmannsdoerfer, M., Benz, S., Juergens, E.: COPE - automating coupled evolution of metamodels and models. In: ECOOP 2009 - Object-Oriented Programming. Volume 5653 of Lecture Notes in Computer Science., Springer Berlin / Heidelberg (2009) 52–76

2. Herrmannsdoerfer, M., Benz, S., Juergens, E.: Automatability of coupled evolution of metamodels and models in practice. In Czarnecki, K., Ober, I., Bruel, J.M., Uhl, A., Völter, M., eds.: Model Driven Engineering Languages and Systems (MODELS 2008). Volume 5301/2008 of Lecture Notes in Computer Science., Springer Berlin / Heidelberg (October 2008) 645–659
3. Herrmannsdoerfer, M., Vermolen, S., Wachsmuth, G.: An extensive catalog of operators for the coupled evolution of metamodels and models. In: Software Language Engineering. (2010)
4. Herrmannsdoerfer, M., Ratiu, D., Wachsmuth, G.: Language evolution in practice: The history of GMF. In: Software Language Engineering. Volume 5969 of Lecture Notes in Computer Science., Springer Berlin / Heidelberg (2009) 3–22
5. Herrmannsdoerfer, M., Benz, S., Juergens, E.: COPE: A language for the coupled evolution of metamodels and models. In: 1st International Workshop on Model Co-Evolution and Consistency Management. (2008)
6. Herrmannsdoerfer, M.: COPE - a workbench for the coupled evolution of metamodels and models. In: Software Language Engineering. (2010)

A Solution

Initialize FigureAccessor.typedFigure

```
1  import org.eclipse.emf.edapt.migration.CustomMigration;
2  import org.eclipse.emf.edapt.migration.Instance;
3  import org.eclipse.emf.edapt.migration.Metamodel;
4  import org.eclipse.emf.edapt.migration.Model;
5
6  public class GMFGraphTypedFigureCustomMigration extends CustomMigration {
7
8      @Override
9      public void migrateAfter(Model model, Metamodel metamodel) {
10         for(Instance fa : model.getAllInstances("gmfgraph.FigureAccessor")) {
11             Instance tf = fa.getLink("typedFigure");
12             if(tf == null) {
13                 tf = model.newInstance("gmfgraph.CustomFigure");
14                 tf.set("qualifiedClassName", "org.eclipse.draw2d.IFigure");
15                 fa.set("typedFigure", tf);
16             } else {
17                 tf.set("name", null);
18             }
19         }
20     }
21 }
```

Decouple FigureHandle.referencingElements

```
1  import java.util.List;
2
3  import org.eclipse.emf.ecore.EReference;
4  import org.eclipse.emf.edapt.migration.CustomMigration;
5  import org.eclipse.emf.edapt.migration.Instance;
6  import org.eclipse.emf.edapt.migration.Metamodel;
7  import org.eclipse.emf.edapt.migration.MigrationException;
8  import org.eclipse.emf.edapt.migration.Model;
9
10 public class GMFGraphFigureDescriptorCustomMigration extends CustomMigration {
11
12     private EReference reference;
```

```
13
14      @Override
15      public void migrateBefore(Model model, Metamodel metamodel)
16              throws MigrationException {
17          reference = metamodel
18                  .getEReference("gmfgraph.FigureHandle.referencingElements");
19      }
20
21      @Override
22      public void migrateAfter(Model model, Metamodel metamodel)
23              throws MigrationException {
24          for (Instance handle : model.getAllInstances("gmfgraph.FigureHandle")) {
25              List<Instance> elements = handle.unset(reference);
26              if (!elements.isEmpty()) {
27                  Instance toplevel = getToplevel(handle);
28                  Instance descriptor = getOrCreateDescriptor(toplevel, model);
29                  for (Instance element : elements) {
30                      element.set("figure", descriptor);
31                  }
32                  if (toplevel != handle) {
33                      Instance access = getOrCreateAccess(descriptor, handle);
34                      for (Instance element : elements) {
35                          if (element.instanceOf("gmfgraph.DiagramLabel")
36                                  || element.instanceOf("gmfgraph.Compartment")) {
37                              element.set("accessor", access);
38                          }
39                      }
40                  }
41              }
42          }
43      }
44
45      public Instance getToplevel(Instance handle) {
46          while (handle.getContainer().instanceOf("gmfgraph.FigureHandle")) {
47              handle = handle.getContainer();
48          }
49          return handle;
50      }
51
52      public Instance getOrCreateDescriptor(Instance toplevel, Model model) {
53          Instance gallery = toplevel.getContainer();
54          if (gallery.instanceOf("gmfgraph.FigureDescriptor")) {
55              return gallery;
56          }
57          Instance descriptor = model.newInstance("gmfgraph.FigureDescriptor");
58          descriptor.set("actualFigure", toplevel);
59          gallery.remove("figures", toplevel);
60          gallery.add("descriptors", descriptor);
61          descriptor.set("name", toplevel.get("name"));
62          return descriptor;
63      }
64
65      public Instance getOrCreateAccess(Instance descriptor, Instance handle) {
66          Instance figure = null;
67          if (handle.instanceOf("gmfgraph.FigureAccessor")) {
68              figure = handle.getLink("typedFigure");
69          } else {
70              figure = handle;
71          }
72          Instance access = findAccess(descriptor, figure);
73          if (access == null) {
74              access = descriptor.getType().getModel()
75                      .newInstance("gmfgraph.ChildAccess");
76              access.set("figure", figure);
77              descriptor.add("accessors", access);
78          }
79          return access;
80      }
```

```
81
82      public Instance findAccess(Instance descriptor, Instance figure) {
83          for (Instance access : descriptor.getLinks("accessors")) {
84              if (access.get("figure") == figure) {
85                  return access;
86              }
87          }
88          return null;
89      }
90  }
```