Solving the TTC 2011 Reengineering Case with Edapt

Markus Herrmannsdoerfer

Institut für Informatik, Technische Universität München

`herrmama@in.tum.de`

This paper gives an overview of the Edapt solution to the reengineering case [6] 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. Edapt is an official Eclipse tool derived from the research prototype COPE.

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 [3]: reusable and custom coupled operations.
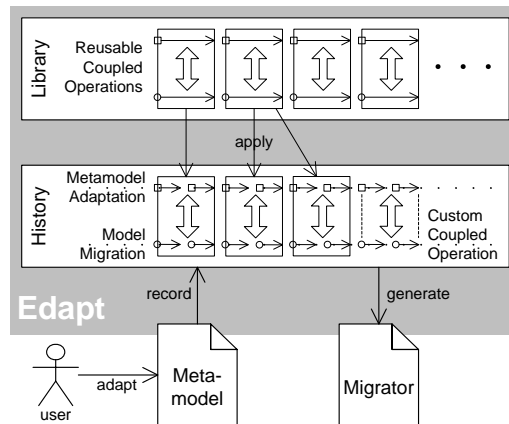
Figure 1: Overview of Edapt

Reuse of recurring migration specifications allows to reduce the effort associated with building a model migration [1]. 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 [5]. 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 [1, 4].

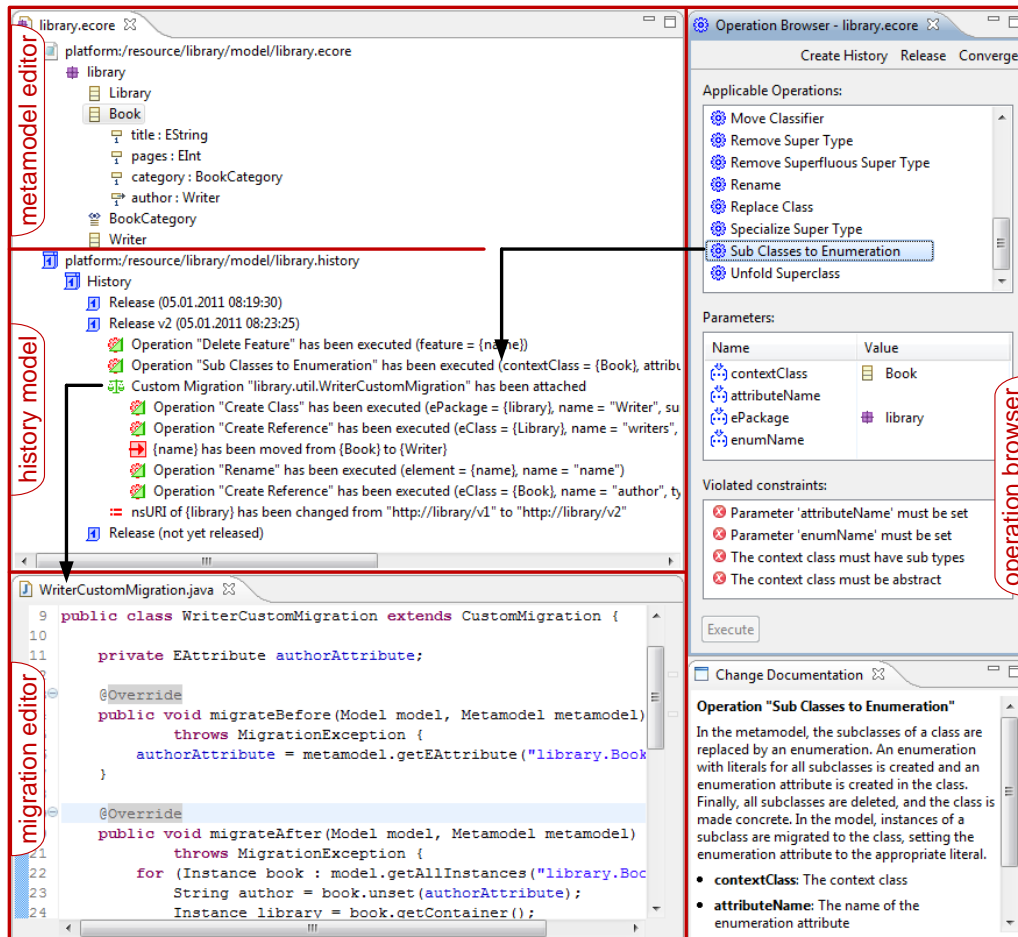
[1] `http://www.eclipse.org/edapt`

Figure 2: User interface of Edapt

Migration specifications can become so specific to a certain metamodel that reuse does not make sense [1]. 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 [2]. By softening the conformance of the model to the metamodel within a coupled operation, both meta-model 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.

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 the operation's applicability based on the constraints. When a reusable coupled operation
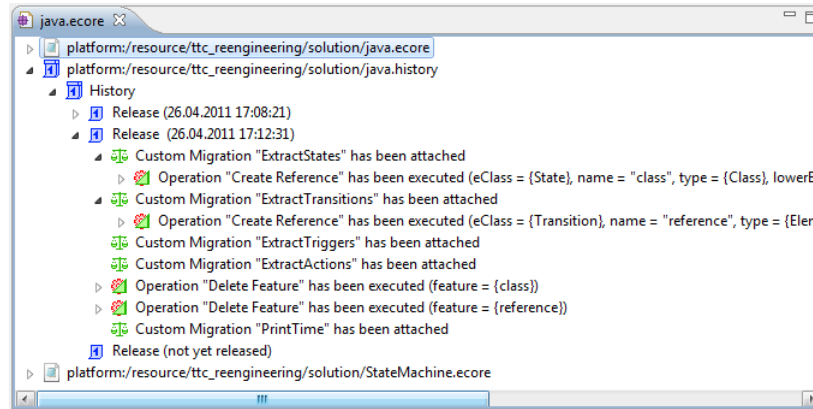
Figure 3: History model for the reengineering case

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 Reengineering Case

Figure 3 shows the history model that solves the reengineering case. The history model was initialized for both the Java and the statemachine metamodel. The history model is modularized into custom coupled operations for extracting states, transitions, triggers and actions from the Java model. In the first steps, trace associations from the statemachine to the Java metamodel are added, which are deleted at the end by reusable coupled operations. In the following, we briefly explain the different steps. The complete solution is available in the appendix, through a SHARE demo [6] and in the repository of the Eclipse Edapt project[2].

Core Task. To extract the states from the Java model (see Listing A.2), we first create the statemachine in a separate model resource (lines 27-43). Then we search for the abstract base class that denotes states in the Java model (lines 58-66). From this class, we navigate opposite to inheritance associations to obtain all non-abstract subclasses (lines 68-103). In Edapt, opposite navigation can be performed very efficiently using the helper method getInverse (e. g. lines 83/84). Finally, we create a state for each non-abstract subclass (lines 45-56). We also associate a state to its implementing class through the association class created for this purpose (line 53).

To extract transitions from the Java model (see Listing A.3), for each class implementing a target state, we search for ElementReferences to it in the classes implementing source states (lines 18-39). This can again be performed very efficiently using the inverse navigation of Edapt. We also have to check whether the ElementReferences are part of a call to the activate method before creating a transition (lines

---

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

41-57). We use the association class created in the last custom coupled operation to navigate from a state in the statemachine model to its implementing class in the Java model (e. g. line 22), and vice versa (e. g. lines 29/30). We associate a transition to its implementing ElementReference through the association reference created for this purpose (lines 60-69).

Extension 1: Triggers. We implemented the logic as explained by the case description to extract triggers from the Java model for each transition in the statemachine model (see Listing A.4). We use the association reference to navigate from a transition in the statemachine model to its implementing ElementReference in the Java model.

Extension 2: Actions. We implemented the logic as explained by the case description to extract actions from the Java model for each transition in the statemachine model (see Listing A.5). Again, we use the association reference to navigate between transitions and their implementations.

3   Conclusion

We discuss the solution with respect to the evaluation criteria defined in the case description.

Understandability. We use Edapt's mechanism to compose coupled operations to modularize the solution into a number of understandable steps. Moreover, each of the custom migrations can again be split up into a number of methods to increase understandability. However, the implementation in Java makes the migration slightly more difficult to understand than a specialized DSL.

Conciseness. The solution is quite concise, since we can rely on Java's inheritance mechanism to reuse recurring methods—e. g. getContainerOfType in the super class ReengineeringCustomMigration to obtain the direct or indirect parent of a certain type. Moreover, the API with which migrations are implemented provides methods that foster a concise implementation—e. g. methods that resolve elements from the metamodel by their fully qualified name. Even though the implementation in Java is quite concise and understandable, a specialized DSL could further improve conciseness and understandability. However, we can rely on the strong Java tooling to implement, refactor and debug the solution.

Completeness. We believe that the solution is complete, since it covers all cases specified in the case description. Moreover, we obtain the same statemachine model as the reference models for the test models.

Performance. Due to the efficient inverse navigation of Edapt which has the same performance as the forward navigation, the transformation has a good performance on the test models. Since all Java test models contain the same statemachine model, the transformation itself takes the same time on all test models: 0.5 to 1 s on the SHARE virtual machine. However, loading the models takes longer for the bigger ones.

References

[1] Markus Herrmannsdoerfer, Sebastian Benz & Elmar Juergens (2008): *Automatability of Coupled Evolution of Metamodels and Models in Practice*. In: *MoDELS '08*, doi:10.1007/978-3-540-87875-9_45.

[2] Markus Herrmannsdoerfer, Sebastian Benz & Elmar Juergens (2008): *COPE: A Language for the Coupled Evolution of Metamodels and Models*. In: *MCCM '08*.

[3] Markus Herrmannsdoerfer, Sebastian Benz & Elmar Juergens (2009): *COPE - Automating Coupled Evolution of Metamodels and Models*. In: *ECOOP '09*, doi:10.1007/978-3-642-03013-0_4.

[4] Markus Herrmannsdoerfer, Daniel Ratiu & Guido Wachsmuth (2009): *Language Evolution in Practice: The History of GMF*. In: *SLE '09*, doi:10.1007/978-3-642-12107-4_3.

[5] Markus Herrmannsdoerfer, Sander Vermolen & Guido Wachsmuth (2010): *An Extensive Catalog of Operators for the Coupled Evolution of Metamodels and Models*. In: *SLE '10*, doi:10.1007/978-3-642-19440-5_10.

[6] Tassilo Horn (2011): *Program Understanding: A Reengineering Case for the Transformation Tool Contest*. In Pieter Van Gorp, Steffen Mazanek & Louis Rose, editors: *TTC 2011: Fifth Transformation Tool Contest, Zürich, Switzerland, June 29-30 2011*, EPTCS.

A Solution

A.1 Base Class for Custom Migrations of the Reengineering Case

```
1  import org.eclipse.emf.edapt.migration.CustomMigration;
2  import org.eclipse.emf.edapt.migration.Instance;
3
4
5  public abstract class ReengineeringCustomMigration extends CustomMigration {
6
7      /**
8       * Get the direct or indirect container of an element that is instance of a
9       * certain type.
10      */
11     protected Instance getContainerOfType(Instance instance, String type) {
12         Instance container = instance;
13         while (container != null && !container.instanceOf(type)) {
14             container = container.getContainer();
15         }
16         return container;
17     }
18 }
```

A.2 Extract States

```
1  import java.util.ArrayList;
2  import java.util.List;
3
4  import org.eclipse.emf.common.util.URI;
5  import org.eclipse.emf.edapt.migration.CustomMigration;
6  import org.eclipse.emf.edapt.migration.Instance;
7  import org.eclipse.emf.edapt.migration.Metamodel;
8  import org.eclipse.emf.edapt.migration.MigrationException;
9  import org.eclipse.emf.edapt.migration.Model;
10 import org.eclipse.emf.edapt.migration.ModelResource;
11
12 public class ExtractStates extends CustomMigration {
13
14     @Override
15     public void migrateBefore(Model model, Metamodel metamodel)
16             throws MigrationException {
17         PrintTime.start = System.currentTimeMillis();
18     }
19
20     @Override
21     public void migrateAfter(Model model, Metamodel metamodel)
22             throws MigrationException {
23         Instance stateMachine = createStateMachine(model);
24         createStates(model, stateMachine);
25     }
26
27     /** Create the state machine. */
28     private Instance createStateMachine(Model model) {
29         ModelResource resource = createResultResource(model);
30         Instance stateMachine = model.newInstance("statemachine.StateMachine");
31         resource.getRootInstances().add(stateMachine);
32         return stateMachine;
33     }
34
35     /** Create the resource to store the state machine in. */
36     private ModelResource createResultResource(Model model) {
37         URI uri = model.getResources().get(0).getUri();
38         URI resultUri = uri
```

```
39                     .trimSegments(1)
40                     .appendSegment(uri.trimFileExtension().lastSegment() + "result")
41                     .appendFileExtension(uri.fileExtension());
42             return model.newResource(resultUri);
43         }
44
45         /** Create the states. */
46         private void createStates(Model model, Instance stateMachine) {
47             Instance stateClass = getStateClass(model);
48             for (Instance subClass : getAllSubClasses(stateClass)) {
49                 if (!isAbstract(subClass)) {
50                     Instance state = model.newInstance("statemachine.State");
51                     state.set("name", subClass.get("name"));
52                     stateMachine.add("states", state);
53                     state.set("class", subClass);
54                 }
55             }
56         }
57
58         /** Get the class that is super class of all classes representing states. */
59         private Instance getStateClass(Model model) {
60             for (Instance c : model.getAllInstances("classifiers.Class")) {
61                 if ("State".equals(c.get("name"))) {
62                     return c;
63                 }
64             }
65             return null;
66         }
67
68         /** Get all direct and indirect sub classes of a class. */
69         private List<Instance> getAllSubClasses(Instance c) {
70             List<Instance> subTypes = new ArrayList<Instance>();
71             for (Instance subType : getSubClasses(c)) {
72                 if (!subTypes.contains(subType)) {
73                     subTypes.add(subType);
74                 }
75                 subTypes.addAll(getAllSubClasses(subType));
76             }
77             return subTypes;
78         }
79
80         /** Get all direct sub classes of a class. */
81         private List<Instance> getSubClasses(Instance c) {
82             List<Instance> subClasses = new ArrayList<Instance>();
83             List<Instance> classifierReferences = c
84                     .getInverse("types.ClassifierReference.target");
85             for (Instance classifierReference : classifierReferences) {
86                 Instance container = classifierReference.getContainer()
87                         .getContainer();
88                 if (container.instanceOf("classifiers.Class")) {
89                     subClasses.add(container);
90                 }
91             }
92             return subClasses;
93         }
94
95         /** Check whether a class is abstract. */
96         private boolean isAbstract(Instance c) {
97             for (Instance modifier : c.getLinks("annotationsAndModifiers")) {
98                 if (modifier.instanceOf("modifiers.Abstract")) {
99                     return true;
100                }
101            }
102            return false;
103        }
```

```
104    }


```

A.3   Extract Transitions

```
1    import java.util.List;
2
3    import org.eclipse.emf.edapt.migration.Instance;
4    import org.eclipse.emf.edapt.migration.Metamodel;
5    import org.eclipse.emf.edapt.migration.MigrationException;
6    import org.eclipse.emf.edapt.migration.Model;
7
8    public class ExtractTransitions extends ReengineeringCustomMigration {
9
10        @Override
11        public void migrateAfter(Model model, Metamodel metamodel)
12                throws MigrationException {
13            Instance stateMachine = model.getInstances("statemachine.StateMachine")
14                    .get(0);
15            createTransitions(model, stateMachine);
16        }
17
18        /** Create the transitions. */
19        private void createTransitions(Model model, Instance stateMachine) {
20            List<Instance> states = stateMachine.getLinks("states");
21            for (Instance targetState : states) {
22                Instance targetClass = targetState.getLink("class");
23                List<Instance> references = targetClass
24                        .getInverse("references.ElementReference.target");
25                for (Instance reference : references) {
26                    if (isTransition(reference)) {
27                        Instance sourceClass = getContainerOfType(reference,
28                                "classifiers.Class");
29                        List<Instance> sourceStates = sourceClass
30                                .getInverse("statemachine.State.class");
31                        if (!sourceStates.isEmpty()) {
32                            Instance sourceState = sourceStates.get(0);
33                            createTransition(model, stateMachine, sourceState,
34                                    targetState, reference);
35                        }
36                    }
37                }
38            }
39        }
40
41        /**
42         * Check whether a certain reference to a state class represents a
43         * transition.
44         */
45        private boolean isTransition(Instance reference) {
46            Instance next = reference.getLink("next");
47            if (next != null && next.instanceOf("references.MethodCall")) {
48                if ("Instance".equals(next.getLink("target").get("name"))) {
49                    next = next.get("next");
50                    if (next != null && next.instanceOf("references.MethodCall")) {
51                        if ("activate".equals(next.getLink("target").get("name"))) {
52                            return true;
53                        }
54                    }
55                }
56            }
57            return false;
58        }
59
60        /** Create a transition from a source to the target state. */
```

```
61    private void createTransition(Model model, Instance stateMachine,
62            Instance sourceState, Instance targetState, Instance reference) {
63        Instance transition = model.newInstance("statemachine.Transition");
64        transition.set("src", sourceState);
65        transition.set("dst", targetState);
66        transition.set("reference", reference);
67        stateMachine.add("transitions", transition);
68    }
69 }
```

A.4   Extract Triggers

```
1  import org.eclipse.emf.edapt.migration.Instance;
2  import org.eclipse.emf.edapt.migration.Metamodel;
3  import org.eclipse.emf.edapt.migration.MigrationException;
4  import org.eclipse.emf.edapt.migration.Model;
5
6  public class ExtractTriggers extends ReengineeringCustomMigration {
7
8      @Override
9      public void migrateAfter(Model model, Metamodel metamodel)
10             throws MigrationException {
11         for (Instance transition : model
12                 .getAllInstances("statemachine.Transition")) {
13             Instance reference = transition.get("reference");
14             transition.set("trigger", getTrigger(reference));
15         }
16     }
17
18     /** Get the trigger of a transition. */
19     private String getTrigger(Instance reference) {
20         Instance method = getContainerOfType(reference, "members.ClassMethod");
21         String methodName = method.get("name");
22         if ("run".equals(methodName)) {
23             Instance switchCase = getContainerOfType(reference,
24                     "statements.NormalSwitchCase");
25             if (switchCase != null) {
26                 return switchCase.getLink("condition").getLink("target")
27                         .get("name");
28             }
29             Instance catchBlock = getContainerOfType(reference,
30                     "statements.CatchBlock");
31             if (catchBlock != null) {
32                 return catchBlock.getLink("parameter").getLink("typeReference")
33                         .getLinks("classifierReferences").get(0)
34                         .getLink("target").get("name");
35             }
36         } else {
37             return methodName;
38         }
39         return "——";
40     }
41 }
```

A.5   Extract Actions

```
1  import org.eclipse.emf.edapt.migration.Instance;
2  import org.eclipse.emf.edapt.migration.Metamodel;
3  import org.eclipse.emf.edapt.migration.MigrationException;
4  import org.eclipse.emf.edapt.migration.Model;
5
6  public class ExtractActions extends ReengineeringCustomMigration {
```

```
7
8      @Override
9      public void migrateAfter(Model model, Metamodel metamodel)
10            throws MigrationException {
11         for (Instance transition : model
12                 .getAllInstances("statemachine.Transition")) {
13             Instance reference = transition.get("reference");
14             transition.set("action", getAction(reference));
15         }
16     }
17
18     /** Get the action of a transition. */
19     private String getAction(Instance reference) {
20         Instance container = getContainerOfType(reference,
21                 "statements.StatementListContainer");
22         for (Instance statement : container.getLinks("statements")) {
23             if (statement.instanceOf("statements.ExpressionStatement")) {
24                 Instance expression = statement.getLink("expression");
25                 if (expression.instanceOf("references.MethodCall")) {
26                     if ("send".equals(expression.getLink("target").get("name"))) {
27                         return expression.getLinks("arguments").get(0)
28                                 .getLink("next").getLink("target").get("name");
29                     }
30                 }
31             }
32         }
33         return "--";
34     }
35 }
```

A.6   Print Time

```
1  import org.eclipse.emf.edapt.migration.CustomMigration;
2  import org.eclipse.emf.edapt.migration.Metamodel;
3  import org.eclipse.emf.edapt.migration.MigrationException;
4  import org.eclipse.emf.edapt.migration.Model;
5
6  public class PrintTime extends CustomMigration {
7
8      public static long start;
9
10     @Override
11     public void migrateAfter(Model model, Metamodel metamodel)
12             throws MigrationException {
13         long time = System.currentTimeMillis() - start;
14         System.out.println(time + " ms");
15     }
16 }
```