Letting EMF Tools Talk to Fujaba through Adapters

Jendrik Johannes
Technische Universität Dresden
Software Technology Group
01062 Dresden, Germany
jendrik.johannes@tu-dresden.de

ABSTRACT

Many software modelling tools are built on top of the Eclipse Modeling Framework (EMF) through which they can communicate and exchange models. In contrast to that, the Fujaba Toolsuite defines its own modelling framework. Both frameworks are built on the same concepts of software modelling. Therefore, they can be adapted. This paper presents an implementation of a generic adapter layer that adapts Fujaba's modelling framework to EMF. Through this adapter layer, Fujaba models can be processed by any EMF-based tool without adapting each of those tools individually.

1. INTRODUCTION

The Fujaba Toolsuite and the Eclipse Modeling Framework (EMF) are both extensible software modelling frameworks. Fujaba, as an academic tool, has a long history and grew since the beginning of UML modelling into a set of tools based on a common core framework. EMF, as an industrial driven open-source framework, has attracted a larger community and consequently many modelling tools were built based on it during the last years. Today, Fujaba and EMF are both stable and productively usable modelling technologies, where each has its advantages and disadvantages for specific software modelling tasks.

In Model-Driven Software Development (MDSD), which attempts to use modelling during the whole software development process, tool integration is very important. Focusing on Fujaba and EMF-based tools, both could be used together and profit from each other in MDSD processes. Therefore, solutions are needed to integrate them tightly.

In earlier works, several ideas were presented that deal with integration and exchange between Fujaba and EMF. They were either based on aligning Fujaba-code itself [8] or on model-transformations that convert Fujaba to and from EMF models [5, 7]. The earlier require invasive changes of the Fujaba source-code, which is problematical when the tool evolves; the latter have a static nature, which requires explicit translations that hinder a smooth runtime integration of both tools and easily lead to data inconsistencies. The approaches so far also did either not succeed or not attempt to provide a generic integration of both frameworks.

In this paper we apply the well-known Adapter Pattern [4] to implement a small set of adapters that mediate between Fujaba and EMF tools at runtime. Therefore, no changes neither of Fujaba nor of EMF are required, but models are kept synchronised at runtime. The only premise for such adapters is that both tools expose the runtime model instances to the outside—which both do.

The paper is structured as follows: Section 2 shows the points at which Fujaba and EMF need to be adapted and describes our adapter implementation. The usability of the implementation is demonstrated using different examples that show how Fujaba interacts with different EMF-based tools in Section 3. Section 4 concludes and discusses possible enhancements of the adapter layer.

2. MAPPING AND ADAPTATION

In this paper, we focus on inspecting and modifying Fujaba models with EMF tools, which can only handle EMF models. To let Fujaba models look like EMF models, the following four concepts, found in both Fujaba's and EMF's implementation of *model*, need to be adapted: A model in Fujaba or EMF (1) conforms to a metamodel, (2) consists of model elements and (3) references between the elements, (4) is instantiated by a factory, and (5) persisted in a resource (e.g., a file). This section shows how the different implementations of the five concepts can be adapted.

2.1 Metamodel Mapping

A metamodel defines the concepts of a modelling language and can therefore be used to instantiate a modelling framework for that language. Since we want to access Fujaba models from EMF, the first step is to make Fujaba's modelling language—UML Class Diagrams, Statecharts, and Activity Diagrams with Story Patterns—known to EMF. This can be done by providing a metamodel of these languages in Ecore[1] format. The second step is to map the metaclasses of this metamodel (i.e., Fujaba metamodel in Ecore format) to the corresponding representations of the metaclasses in Fujaba (i.e., Java classes implementing the metaclasses).

To solve this issue, we implemented a tool that extracts information about the Fujaba metamodel from the Fujaba class files using Java's refelection facilities. For each Java class that belongs to Fujaba's metamodel implementation, the tool constructs an `EClass` (EMF metaclass representation) and organises all of them in `EPackages` (EMF metamodel representations), which are then registered in EMF's metamodel registry. For this, the tool makes assumptions about Fujaba's metamodel implementation but also requires additional information that can not be derived from the implementation allone. Describing all the assumptions and additional parameters in detail is not possible in this paper due to space limitations. Some assumptions are, however, presented in the context of Sections 2.2 and 2.3.

[1]EMOF [9] conformant metamodelling language of EMF

During the extraction, a mapping between the Fujaba metaclass representation (Java class objects) and Ecore metaclass representations (`EClass` objects) is created. The mapping is used by the adapters described in the next sections.

2.2 Model Element Adapter

The most important objects to adapt are the ones representing model elements. In both Fujaba and EMF, each model element is represented by one Java object. In Fujaba, all those objects are instances of a class implementing `FElement`. In EMF, the objects are instances of a class implementing `EObject`. Both interfaces offer convenient functionality for each model element; for instance comparability or listener support. In Fujaba, the Java class of which a model element is an instance corresponds to that element's metaclass. In EMF, this does not have to be the case.[2] The metaclass of an element can always be determined by the `eClass()` method of `EObject`. A standard implementation of `EObject` that can be used to represent any model element is `DynamicEObjectImpl`.

This loose coupling between metaclasses and their Java implementation is possible in EMF because the `EObject` interface offers rich reflection capabilities to inspect and modify models. For example, `eGet(String feature)` delivers the value of a feature by naming it; `eSet(String feature, Object value)` sets a feature to the given value. In Fujaba, such reflection capabilities are not provided by the `FElement` interface. Models can only be inspected and modified by using Java methods that correspond to element features (e.g., `get<featureName>()`).

Our adapter implementation delegates the reflective methods of `EObject` to methods of Fujaba metamodel classes using Java reflection. It hereby makes assumptions about the method names in Fujaba metamodel classes which were already used in the metamodel mapping (Section 2.1) and correspond to method naming applied by the Fujaba code generator.

The dynamic model element adapter, `DynamicEObject4FujabaModels`, is implemented as an extension of the existing `DynamicEObjectImpl`. The `DynamicEObject4FujabaModels` constructor expects an instance of `FElement`. This is the adapted Fujaba object to which the constructed adapter is bound for its lifetime. Furthermore, the methods `dynamicGet()` and `dynamicSet()` are overridden that handle reading and writing properties of the object. They realize the connection between the public `EObject` interface and the storage of information.

`dynamicGet(int featureID)` is implemented as follows:

1. Use the featureID (an identifier for a `EStructuralFeature` object that represents a feature defined by an `EClass`) to obtain name, mutiplicity, and type of the feature
2. If the feature's multiplicity is 1: find and call the method `get<featureName>()` on the adapted fujaba element (call the method `is<featureName>()` instead, if the type is `boolean`)
   (a) If the called method returns a String, Integer, or Boolean value, return that value
   (b) If the called method returns an `FElement`, ask the `DynamicEObject4FujabaModelsFactory` (cf. Section 2.4) for the corresponding Adapter and return it
3. If the feature's multiplicity is > 1: if no `DynamicEList4FujabaModels` representing the multiplicity feature was constructed yet, construct one (cf. Section 2.3); return the `DynamicEList4FujabaModels` representing the multiplicity feature

`dynamicSet(int featureID, Object value)` is implemented as follows:

1. Use the featureID to obtain name, mutiplicity, and type of the feature
2. If the feature's multiplicity is 1:
   (a) If the type is an `EClass`, ask the metamodel mapping (cf. Section 2.1) for the corresponding Fujaba metamodel class and use that as type (otherwise the type is String, Integer, or Boolean and can be used as is)
   (b) Find and call `set<featureName>(<featueType>)` on the adapted fujaba element
3. If the feature's multiplicity is > 1: do nothing (handled by `DynamicEList4FujabaModels`, cf. Section 2.3)

2.3 Collection Reference Adapter

Features with a multiplicity > 1 have to be handled explicitly in modelling tools implemented in Java, since there is no notion of attributes with a multiplicity > 1 in Java directly. Instead, collection objects have to be used. Fujaba and EMF handle this differently: Fujaba offers three[3] methods for each multiplicity > 1 feature directly on the corresponding metaclass. These methods are `iteratorOf<featureName>()` (returns an iterator over the feature), `addTo<featureName>(value:<featureType>)` (adds an element to the feature), and `removeFrom<featureName>(value:<featureType>)` (removes an element from the feature). In EMF, no such methods are defined by metaclasses. Instead, a list object is returned when the value of a specific feature is requested. Clients can add/remove elements to/from the list which directly manipulates the feature. Therefore, EMF comes with its own extension of the Java Collections Framework to insert additional functionality into the list methods which are required for model manipulation. In particular, lists representing features have to implement the interfaces `EList`.

For our adaptation, we require an `EList` that delegates all operations to methods of the adapted Fujaba object. In the implementation `DynamicEList4FujabaModels`,it was not possible to reuse an existing implementation (as for the model element adapter), since the functionality of storing data is so fundamental to Java's list implementations that nearly every methods accesses the data storage directly. Consequently, we implemented the required interfaces directly and did the following delegations to the adapted Fujaba element:

[2]It can be the case if EMF's code generation is applied

[3]There are more methods to access features, but the enumerated three are sufficient. Availability of other methods also varies between metaclasses.

1. `iterator()` delegates to `iteratorOf<featureName>()`
2. `add()`delegates to `addTo<featureName>()`
3. `remove()` delegates to `removeFrom<featureName>()`

The adapted Fujaba element—and the feature that is adapted by a `DynamicEList4FujabaModels`—are both passed to that list in its constructor, when it is created by a `dynamicGet()` method call of a `DynamicEObject4FujabaModels` (cf. Section 2.2). All other list operations dictated by the implemented interfaces are based on the three that delegate to the Fujaba element.

2.4 Model Element Factory Adapter

The two presented adapters are used to access and manipulate existing model elements. What is not supported yet is the creation of new elements, which is implemented in the `DynamicEObject4FujabaModelsFactory`. Both Fujaba and EMF use factories for model element creation and have a registry for factories that can be queried for a suitable factory for a given metaclass. Therefore, adaptation is straight forward: `DynamicEObject4FujabaModelsFactory` extends EMF's standard factory implementation `EFactoryImpl` by overriding the `basicCreate(EClass)` template method. Our implementation uses the metamodel mapping (cf. Section 2.1) to obtain the Fujaba metaclass corresponding to the given `EClass`. It then asks Fujaba for a factory suitable for that metaclass and uses that factory to create a new Fujaba model element.

A singleton `DynamicEObject4FujabaModelsFactory` is registered at the metamodel created by the mapping (cf. Section 2.1). Doing so forces EMF to use this factory for creating elements conforming to the corresponding metamodel.

The `DynamicEObject4FujabaModelsFactory` also acts as a model element adapter (cf. Section 2.2) registry. It is used by all four adapter types (cf. Sections 2.2, 2.3, 2.4, 2.5) to obtain an adapter for a Fujaba model element. If the adapter does not exist yet, it is created for the corresponding element and registered.

2.5 Resource Adapter

The last missing piece in the adaptation is the storing and loading of models. Fujaba models are loaded and stored by Fujaba and made available in Fujaba's workspace. In EMF, so called `Resources` are used to represent physical storage. We leave the loading and storing of models to Fujaba and adapt it as well, by providing a `FujabaResource` that, instead of loading from and writing to a physical storage, accesses the Fujaba workspace.

Resource types can be registered at EMF for a file extension. We register the `FujabaResource` for the extension `.fujaba`. When a file with the `.fujaba` extension is opened in the Eclipse workspace, EMF attempts to load it as a `FujabaResource`, which takes the file name of the opened file and looks for a Fujaba project (in Fujaba's workspace) with the same name. It then retrieves the adapter for that project (which is also a Fujaba model element) from the registry (cf. Section 2.4) and sets it as the resource's content. Therefore, any Fujaba project can now be accessed as EMF model from Eclipse by creating an empty file somewhere in the Eclipse workspace using the scheme: `<FujabaProjectName>.fujaba`.[4] Saving works in a similar fashion by manipulating the Fujaba workspace in `FujabaResource`'s saving method.

3. APPLICATIONS

This section shows how the adapter is used by different EMF-based tools without further effort. Its aim is to demonstrate the rich possibilities of a generic integration of Fujaba and EMF as demonstrated in the last section. Throughout this section we use s simple UML model modelled in Fujaba shown in Figure 1.

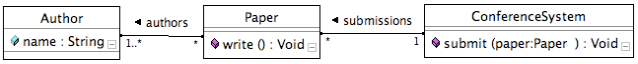


Figure 1: Model of a conference system in Fujaba

3.1 Displaying Fujaba Models in EMF-style

The first simple application is a tree model editor included in EMF. This editor can display any model independent of its metamodel. It uses the containment relationships between model elements to determine the tree structure. Figure 2 shows the example opened in that editor. Note that we can not only inspect, but also modify the model with the editor.

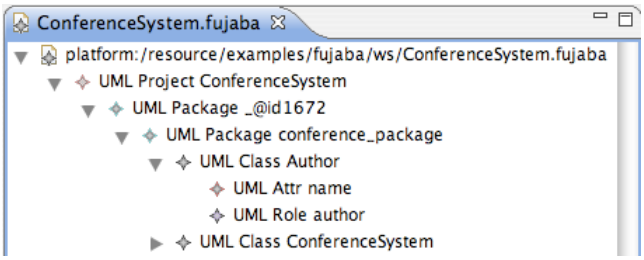


Figure 2: Fujaba model displayed in EMF's editor

3.2 EMF Compare: Diffing Fujaba Models

EMF Compare [3] is a tool that computes diffs between two versions of a model and visualises them using tree representations as used by the editor above. Imagine that we import a new version of the example into the Fujaba workspace under the name ConferenceSystemNew, where the class Paper has been renamed to Submission. EMFCompare suggests that Submission was indeed Paper before by inspecting both versions' structures as shown in Figure 3.

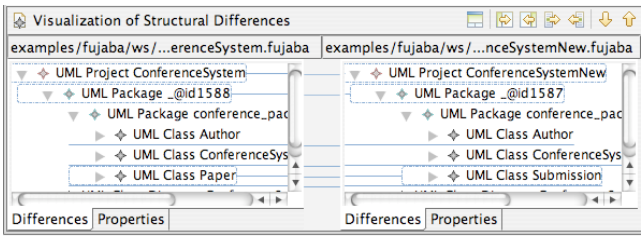


Figure 3: Visual diff between Fujaba models

[4]Fujaba has to run in the same JavaVM as Eclipse such that the Fujaba workspace can be accessed.

3.3 Transforming Fujaba Models with ATL

For EMF, there are plenty of model transformation and management tools available; many of them in the Eclipse Modelling Project [11]. As one representative, we show the ATL [1] transformation tool, in which model transformations can be defined in declarative rules. One could for instance define a transformation from Fujaba UML to Eclipse UML2 [2] and then open the result with editors for the latter. Figure 4 shows an excerpt of such a transformation and the result of transforming the conference system model.

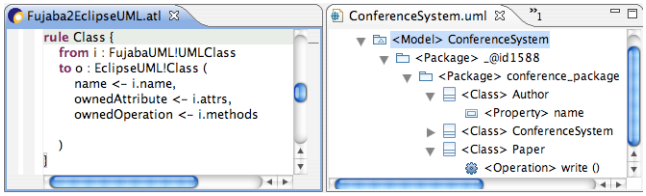


Figure 4: Transforming Fujaba to Eclipse UML

3.4 Reuseware: Composing Fujaba Models

In our own tool Reuseware [6, 10] we can define (cross-cutting) composition systems for modelling languages. We defined, for instance, a composition system for Fujaba class diagrams that can be used to extend classes with pre-defined functionality. Figure 5b shows a Reuseware composition program that extends the example model with observer behaviour by reusing the prior defined observer model (cf. Figure 5a). Reuseware can execute the composition program and produce a composed model (cf. Figure 5c). This application was indeed our motivation for developing the adaptation mechanism and will be further explored in future research.

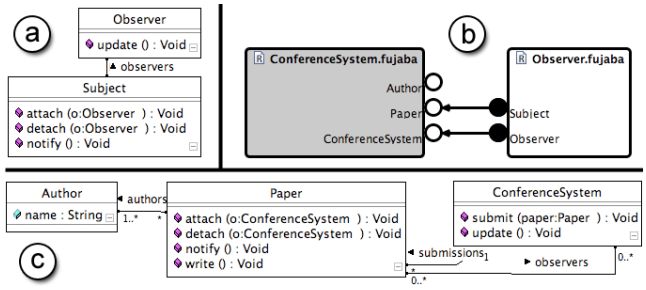


Figure 5: Composing two Fujaba models

4. CONCLUSION AND OUTLOOK

In this paper we described a metamodel mapping and a set of adapters that enable EMF tools to access models in the Fujaba workspace. This opens the door for interesting new projects that combine EMF tools and Fujaba.

A prerequisite for the adaptation is the availability of an Ecore version of the Fujaba metamodel. Since this was not available, a tool was written to extract this metamodel from Fujaba's class files. This tool requires additional information, which are at the moment statically encoded. In the future, this could be made configurable, and different configurations could be provided for different Fujaba versions to ensure that the tool works with them. Another approach would be to construct a complete Fujaba metamodel in UML by using Fujaba reverse engineering tools and, if required, manual modelling. [5] can then be applied to transform it into an Ecore metamodel of Fujaba.

We realised dynamic adapters using reflection. One could also think of a generative approach that generates specific adapters for metaclasses, avoiding the usage of reflection to increase performance. Such an approach might also be applicable the other way around—to access EMF models from Fujaba. For this, Ecore metamodels would have to be mapped to extensions of Fujaba's metamodel and adapters that work in the reverse direction must be generated.

For the adapters to work, both the EMF tool and Fujaba have to run in the same JavaVM, which caused some ClassLoader issues during our experimentation. How to resolve those has to be further investigated. Another open issue is to integrate the adapters with Fujaba4Eclipse, which should not be difficult, since Fujaba4Eclipse is based on the same framework as Fujaba. On the other hand, the integration should be much smoother and less problems should occur, since Fujaba4Eclipse and the EMF-based tools both run inside Eclipse (and therefore in the same JavaVM).

5. ACKNOWLEDGMENTS

This research has been co-funded by the European Commission within the 6^{th} Framework Programme project Modelplex contract number 034081 (cf. `www.modelplex.org`).

6. REFERENCES

[1] ATL Project. Atlas Transformation Language. `www.eclipse.org/m2m/atl`. Accessed Aug. 2008.

[2] Eclipse Foundation. Eclipse UML2 Project. `www.eclipse.org/uml2`. Accessed Aug. 2008.

[3] EMF Compare Project. Emf compare. `www.eclipse.org/emft/projects/compare`. Accessed Aug. 2008.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, MA, 1994.

[5] L. Geige, T. Buchmann, and A. Dotor. Emf code generation with fujaba. In L. Geiger, H. Giese, and A. Zündorf, editors, *Proc. of the 5^{th} International Fujaba Days, Kassel, Germany*. University of Kassel, Oct. 2007.

[6] F. Heidenreich, J. Henriksson, J. Johannes, and S. Zschaler. On language-independent model modularisation. *Transactions on Aspect-Oriented Development, Special Issue on Aspects and MDE (to appear)*, Nov. 2008.

[7] F. Heidenreich and U. Wemmie. Breaking the domination of the internal graph model. In L. Geiger, H. Giese, and A. Zündorf, editors, *Proc. of the 5^{th} International Fujaba Days, Kassel, Germany*. University of Kassel, Oct. 2007.

[8] J. Johannes, I. Savga, and T. Haupt. Integrating fujaba and the eclipse modeling framework. In H. Giese and B. Westfechtel, editors, *Proc. of the 4^{th} International Fujaba Days, Bayreuth, Germany*, volume tr-ri-06-275. University of Paderborn, Sept. 2006.

[9] Object Management Group. MetaObject Facility (MOF) specification version 2.0. OMG Document, Jan. 2006. `http://www.omg.org/cgi-bin/doc?formal/2006-01-01`.

[10] Reuseware Project. Reuseware Composition Framework. `http://www.reuseware.org`. Accessed Aug. 2008.

[11] The Eclipse Foundation. Eclipse modelling project. `http://www.eclipse.org/modelling`. Accessed Aug. 2008.