Developing a Model Composition Framework with Fujaba – An Experience Report

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

ABSTRACT

Reuseware is an open-source model composition framework for composing models defined in arbitrary Ecore-based languages. In its four years of development, Reuseware has experienced many extensions and refactorings due to the integration of new research results and requirements. One year ago, a redevelopment of Reuseware's core was started. Thanks to its EMF code generation, Fujaba was introduced as a new development tool into Reuseware's development toolchain to replace major parts of Java coding through story driven modelling. With this we solved problems with behavior modelling and code generation we faced in the development so far. This paper summarizes our experiences in developing with Fujaba and suggests improvements for Fujaba and its EMF code generation based on that.

1. INTRODUCTION

The Reuseware project[1] was started at the Software Technology Group of TU Dresden in 2005 as successor of the COMPOsition SysTem (COMPOST) framework[2]. COMPOST implemented the concepts of Invasive Software Composition (ISC) [1] for Java and XML. ISC is a static software composition approach that can act as a basis to implement a variety of composition techniques for arbitrary languages. While COMPOST showed the applicability of the approach for Java and later for XML, it was completely hand-written. Adapting it for XML for instance, took considerable effort.

The aim of the Reuseware project is to build a framework for ISC where new languages can be plugged in without manual coding only by providing a grammar or a metamodel of the language. It was clear from the beginning that Reuseware should be developed as an Eclipse extension to profit from the features already provided in the open IDE. The second thing required was a meta language to describe languages that can be plugged in. After several experiments, Ecore—an implementation of the OMG's EMOF standard [8]—of the Eclipse Modelling Framework (EMF) [10], was chosen. We decided for Ecore because of its standard conformance and code generation facilities that integrate nicely into Eclipse. Furthermore, since we initially focused on textual languages, we needed some grammar processing tooling on top of Ecore. Thus, we developed EMFText [5] that was initially part of Reuseware.

*This research has been co-funded by the European Commission in the 6th Framework Programme project MODELPLEX contract no. 034081 (`www.modelplex.org`).

Eventually, Reuseware itself was developed using Ecore for its metamodels, EMFText for textual and GMF [3] for graphical specification languages. What proved to be most problematic was the metamodelling in Ecore. Since it does not support behavior modelling, we had to add operation bodies manually to the generated code. As a consequence of that, we ended up with muddled generated and hand-written code and an unnatural separation of methods into utility classes. After several iterations and experiments, it became clear in the end of 2008 that Reuseware needed a major redesign.

At that point, we saw the main problems of the implementations in 1) the too tight integration of generated and hand-written code 2) the implementation of model (i.e., graph) transformations in Java, which was unnatural, buggy and hard to maintain. For both issues we desired a generative solution. Fujaba with its story diagram paradigm—to model graph transformations—and its EMF code generation [2]—to generate operation bodies of Ecore models from story diagrams—was the ideal candidate for that. This paper summarizes our experience in redeveloping huge parts of Reuseware with Fujaba. Furthermore, it explains extensions we made to Fujaba's EMF code generation.

2. DEVELOPING REUSEWARE WITH FUJABA

Describing Reuseware in detail is out of scope of this paper (please refer to [4, 6, 7] and the Reuseware website[1]). Nevertheless, we present the architecture of Reuseware in Sect. 2.1 to clarify which modelling technologies are used and where Fujaba fits in. We explain how Fujaba was integrated into our development toolchain and how it was customized for our purposes in Sect. 2.2. Our experiences in modelling with Fujaba are then described in Sections 2.3–2.5.

2.1 Reuseware Architecture

In Reuseware, we distinguish two user roles: *composition system developers* and *composition system users*. A composition system implements a certain component and composition methodology. Module systems or aspect systems are examples of composition systems. Composition systems can usually only handle components (e.g., modules or aspects) written in a specific programming or modelling language. Integrating new languages or new types of components usually takes considerable effort. Reuseware however,

[1]`http://www.reuseware.org`
[2]`http://www.the-compost-system.org`

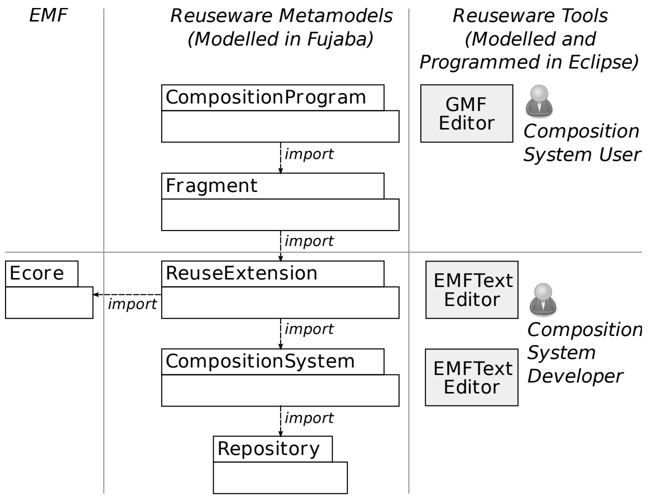

Figure 1: Reuseware architecture overview

is a framework which can be easily instantiated by *composition system developers* to support new composition systems for existing or newly developed languages. The such instantiated framework may then be used by *composition system users* to define components and composition programs (i.e., instantiations and compositions of components).

Reuseware is built around five metamodels shown in Fig. 1 (middle). Each metamodel is modelled as a class diagram in Fujaba. Each of these class diagrams is then translated to an Ecore model by Fujaba's EMF code generation. Instances of the metamodel are either derived by Reuseware or have a concrete syntax that can be used by a composition system developer or a composition system user directly.

The core metamodel is the *Repository* that models a package structure into which different types of elements can be placed. The repository metamodel is instantiated by the running Reuseware based on actual files in the workspace. It provides a component oriented viewpoint on these files for composition system users and developers alike.

A composition system developer may utilise two dedicated languages for composition system development—one language to define the concepts of a *CompositionSystem* and one language to specify where these concepts are found in a language defined as an Ecore metamodel. We call such a specification a *ReuseExtension* for a given language. Both, *CompositionSystem* and *ReuseExtension* language, have a textual syntax defined with EMFText. The *ReuseExtension* language embeds OCL [9] as expression language. These two metamodels are only used for specification and do not have operations that modify models. They therefore make use of story diagrams only to simplify access to elements (e.g., to find one item with a given name in a list).

A composition system user works with two kinds of artifacts: *Fragments* (the components in Reuseware) and *CompositionPrograms* (specifications for compositions of fragments). A fragment has a composition interface through which model elements are accessed (or modified) during a composition. Concepts for composition interfaces are modelled in the *Fragment* metamodel. Instances of that metamodel are created by interpreting *ReuseExtension* specifications on arbitrary EMF models. For this, story diagrams are used in combination with the OCL expressions embedded as strings in *ReuseExtension* models.

Composition programs can again be created by interpreting *ReuseExtension* models but also manually by a composition system user. It depends on whether the composition system developer specified a dedicated composition language for the composition system or decided to use the generic composition language of Reuseware. This composition language is defined in the *CompositionProgram* metamodel and has a graphical syntax defined with GMF. In any case, parts of a composition program can be derived and updated automatically. This is specified with story diagrams.

2.2 Setup and Customization

The development was performed with the SVN versions of Fujaba and the CodeGen2 plugin of Sept. 22nd 2008. To allow rapid development and testing, an ANT build script was written that deletes the previous generated code, distributes Ecore models and Java code generated by Fujaba correctly over several Eclipse plugin projects and triggers EMF's own code generation. Consequently, two clicks are needed to generate the code: Code generation in Fujaba and running the ANT script in Eclipse. The metamodelling was from now on performed in Fujaba. We continued to use Eclipse for EMFText modelling, GMF modelling and Java coding.

Some modifications of the code generation templates of Fujaba were needed to add features not yet supported by Fujaba's EMF code generation. In the following we summarize these modifications.

1. *Splitting the metamodel* The layered metamodel architecture depicted in Fig. 1 is realized by providing one Ecore file and one Eclipse plugin per metamodel. The dependencies between the plugins correspond to the dependencies between the metamodels. To support this splitting while preserving references between the metamodels, the code generation was adjusted.

2. *Referencing existing Ecore models* As one can see in Fig. 1 (left), some of the Reuseware metamodels depend on the Ecore metamodel (i.e., the *Ecore.ecore* file found in EMF) that is modelled in Ecore itself. We made this metamodel available inside Fujaba by importing the *org.eclipse.emf.ecore.jar* that contains the code generated from *Ecore.ecore*. This allows us at least to reference classes from that model in class diagrams. The templates, however, did not support *referenced* classes at all. We modified the templates to create references to *Ecore.ecore*, which exists in every EMF installation, when appropriate.

3. *List return types* In Ecore, return types of operations can be multiple (upper bound > 1). This is not supported by Fujaba and its EMF code generation. Since we needed this feature, we integrated it by introducing the stereotype *multiple* in Fujaba. The extended code generation sets the upper bound of an operation with

that stereotype to *. EMF expects the code of such operations to return an *org.eclipse.emf.util.EList< T>* where T is bound to the return type of the operation. Consequently, the developer of the story pattern of a *multiple* operation has to instantiate, fill and return such an *EList* manually using Java statements.

4. *String arrays* In the case of strings (and other primitive types) the above situation is better, because Fujaba offers explicit primitive array types (e.g., *StringArray*) in its standard library. We extended the code generation to translate Fujaba's *StringArray* into *EList<String>*. Similar conversion could be done for the other primitive types but are not needed in our case.

5. *Navigating operations as links* In story diagrams, we sometimes needed to navigate a virtual path rather than a direct reference in a model. Fujaba supports this through *path expressions*. The drawback of these expressions is that they are translated into an interpreter call which is only checked at runtime and always requires a path expression interpreter. We needed path expressions not for complex expressions, but to use the result of an operation as path or to access references of the Ecore metamodel, which are not known by Fujaba (because *Ecore.ecore* was imported as jar file as described above). This limited use of path expressions enabled us to modify the templates to translate a path expression into an operation call instead of calling the expression interpreter. Additional code ensures that the result of the operation is wrapped into an iterator as expected by the rest of the template.

6. *Support for eKeys* To improve the serialization of references between elements in XMI files, Ecore offers the *eKeys* concept. An *eKey* is essentially a primary key that identifies a model element (e.g., a name attribute is a good candidate for an *eKey*). The serialization then uses the *eKey* to identify cross-referenced elements—instead of using the positions of the element which is the default. Using positions can lead to problems when two XMI files reference each other and one is changed independent of the other. This is the case for the composition program GMF editor, where the layout information (e.g., position of boxes) is saved in a different file than the model elements. To support *eKeys*, we introduced the stereotype *ID* in Fujaba. If an attribute is stereotyped with *ID*, the extended templates define an *eKey* based on that attribute.

7. *Generate code with bound list parameters* A not vital but nice extension is the binding of type parameters in lists and iterators. Since Reuseware is developed in Java5, the generated code produced compiler warnings concerning unbound type parameters. When we cleaned up the code, we addressed all compiler warnings. One of this was to generate the type parameter binding, which was not difficult because all required type information is available during code generation.

To summarize, modifications 1 and 5 seem to be very specific for the development of Reuseware. All other extensions however, could be beneficial for other projects as well. It should be investigated, if and how these modifications can be integrated into the current Fujaba trunk templates.

2.3 General Development Experience

Despite the use of two different development environments, the development felt quite integrated. With the help of the above mentioned build script, changes in the Fujaba models are quickly updated in the Eclipse workspace. The overall generation process takes less than 10sec (on 2.33 GHz Intel Core 2 Duo) and requires only two clicks which is acceptable.

The adjustment of the templates themselves was manageable. It was done in a normal text editor. We have to admit that we did not bother to acquire a proper velocity template editor which could have eased the template modification. Very positive was however that the CodeGen2 templates could be updated in a running Fujaba, which made the debugging of changed templates easy.

Template customization was mainly a concern in the first development phase. Here frequent updates had to be done to support the desired behavior. Such adjustments however, became less frequent and no adjustment were required in the last six month—despite of ongoing development.

2.4 Working with Class Diagrams

Before we used Fujaba, the metamodels of Fig. 1 where developed directly in Eclipse. For this an open-source Ecore diagram editor provided by the TOPCASED project[3] was utilised. We compare our experiences using this editor with using the Fujaba class diagram editor.

The user experience of the Ecore diagram editor was in general not very good. 1) Often the editor feels unstable and sometimes a diagram looks different after we saved and opened it again. Most annoying was the weak support for automatic layouting such that drawing a straight line always was a challenging task. 2) Copy and paste was not supported very well. To perform such task, we used the graphical editor in combination with the tree Ecore editor that comes with Ecore itself (and supports copy and paste very well). However, keeping the such modified model synchronised with the diagram representation was also not a strength of the diagram editor. At some point, the diagram file could not be opened anymore and the whole layout was lost. 3) Another difficulty was the specification of bi-directional associations, which are modelled as two uni-directional references that are connected via an *opposite* relationship in Ecore. The editor did not provide a facility to specify or represent those reference together. Thus, specifying a bi-directional association was cumbersome and ugly in the diagram—it was represented by two lines.

Although the class diagram editing was not the reason to switch to Fujaba, we were very pleased that the class diagram editor of Fujaba overcame the weaknesses of the Ecore diagram editor. 1) Using the editor feels very smooth and stable. Lines between classes are drawn straight automatically when possible. 2) Copy and paste is supported to a high degree and we were always able to perform a desired restructuring without having to re-model anything. 3) Bidirectional associations can be defined in Fujaba naturally. The EMF code generation translates these associations into two references in the Ecore model just as we expected.

[3]`http://www.topcased.org`

2.5 Working with Story Diagrams

As mentioned, the main motivation to use Fujaba was to define metamodel operations as story diagrams. As in class diagrams, the combination of manual and automatic layout gives the user a smooth editing experience and even the (re)structuring of large diagrams was easy. Although we tried to model as much as possible, the openness toward calling Java code directly was vital to continue the work at places were it was not obvious how to model the functionality best. It was necessary to achieve the integration with existing code—mainly with the EMF and an OCL interpreter (cf. next section). The cut and paste functionality was also very useful in particular to refactor story diagrams.

We can make the following improvement suggestions for story diagrams based on our experience.

1. *Calling story diagrams* To call one story diagram from another one is currently only possible by calling the Java method that is generated from that story diagram. This feels unnatural, since one calls the generated code (i.e., the Java method) and not the story diagram (i.e., the UML operation) on the modelling level. Because we wanted to stay on the modelling level, we avoided to split story diagrams in the beginning and the diagrams grew unnecessarily large. Having an explicit mechanism to call story diagrams would improve the modelling experience here.

2. *List return types* Fujaba does not support a mechanism to declare a return type of an operation as *multiple*. One can set the return type to *FHashSet*, but this does not say anything about the type of the values that may be contained in the set and thus can not be properly processed by the EMF code generation. It is fine to work with the *multiple* stereotype extension we presented in Sect. 2.2, but having the capability directly integrated into Fujaba would be even nicer.

3. *Import of existing Ecore models* The Reuseware metamodels depend on the Ecore metamodel (cf. Fig. 1, left). The metamodel is modelled in Ecore itself (in an *ecore* file). As mentioned, we imported this metamodel by importing the jar file that contains the code generated from the metamodel. This gave us access to the metamodel types which was sufficient to model class diagrams. In story diagrams however, we could not model edges between instances of classes from the Ecore metamodel since associations were not extracted from the jar file. This information is however contained in the *ecore* file. Providing an import for *ecore* files into Fujaba (the inverse of the EMF code generation) would greatly enhance the integration of Fujaba and EMF and would allow people to define story diagrams for their existing Ecore models.

Recently, we refactored our story diagrams and split them into smaller diagrams such that each diagram fills one A4 page at maximum when printed. All Reuseware metamodels together now contain 61 story diagrams (and 73 classes).

4 `http://www.eclipse.org/modeling/mdt/?project=ocl`

2.6 Tool Interoperability

Through Fujaba's EMF code generation, the tool integration worked very smooth. The Ecore models produced by the code generation could be handled by any other EMF based tool. As illustrated in Fig. 1, we used EMFText and GMF to build editors for three of our five metamodels.

In one metamodel (*ReuseExtension*) we allow the specification of OCL expressions as strings. To interpret these expressions, we use the MDT OCL interpreter[4], which works with Ecore models. We implemented a small *Evaluator* utility class with static methods that initialize the OCL environment and use it to evaluate the embedded expressions. We then use Fujaba's ability to refer to arbitrary Java classes and methods inside of story diagrams to call methods on the *Evaluator*. In particular, it is used in to evaluate *boolean guard expressions* of transitions and to derive values for *attribute assignments* of an object in a story activity.

3. CONCLUSION

In this paper we reported on the development of Reuseware with Fujaba. We conclude that using Fujaba significantly improved the development experience and the quality of the developed tooling. Thus using Fujaba was the correct decision. With story driven modelling we were able to increase the amount of modelling and to improve separation of generated and hand-written code. We hope that the extensions of the EMF code generation discussed in Sect. 2.2 and the story diagram improvements suggested in Sect. 2.5 can help to improve Fujaba in the future. This paper showed that Fujaba's EMF code generation is usable in practice to develop EMF-based tools with Fujaba and to profit from story driven modelling in EMF.

4. REFERENCES

[1] U. Aßmann. *Invasive Software Composition.* Springer, Secaucus, NJ, USA, 2003.

[2] L. Geige, T. Buchmann, and A. Dotor. EMF Code Generation with Fujaba. In *Proc. of the 5th International Fujaba Days.* University of Kassel, 2007.

[3] R. C. Gronback. *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit.* Pearson Education, 2009.

[4] F. Heidenreich, J. Henriksson, J. Johannes, and S. Zschaler. On Language-Independent Model Modularisation. In *Transactions on Aspect-Oriented Software Development VI*, volume 5560 of *LNCS*. Springer, 2009.

[5] F. Heidenreich, J. Johannes, S. Karol, M. Seifert, and C. Wende. Derivation and Refinement of Textual Syntax for Models. In *Proc. of ECMDA-FA '09*, volume 5562 of *LNCS*. Springer, 2009.

[6] J. Johannes. Controlling Model-Driven Software Development through Composition Systems. In *Proc. of NW-MODE '09.* Tampereen teknillinen yliopisto, 2009.

[7] J. Johannes, S. Zschaler, M. A. Fernández, A. Castillo, D. S. Kolovos, and R. F. Paige. Abstracting Complex Languages through Transformation and Composition. In *Proc. of MoDELS'09*, volume 5795 of *LNCS*. Springer, 2009.

[8] Object Management Group. MOF 2.0 core specification. OMG Document, Jan. 2006. `www.omg.org/spec/MOF/2.0`.

[9] Object Management Group. Object Constraint Language, Version 2.0, May 2006. `www.omg.org/spec/OCL/2.0`.

[10] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *Eclipse Modeling Framework, 2nd Edition.* Pearson Education, 2008.