

A MOF 2.0 Editor as Plug-in for *FUJABA*

Carsten Amelunxen
Technische Universität Darmstadt
Institut für Datentechnik, FG Echtzeitsysteme
Merckstr. 25
Darmstadt, Germany
carsten.amelunxen@es.tu-darmstadt.de

ABSTRACT

In this paper we describe how we build a MOF 2.0 editor as a plug-in for Fujaba. The new versions of UML and MOF offer new concepts for structural modeling. We will use these new concepts to generate metamodels for several domains in compliance with common standards like MOF 2.0, JMI, OCL. We figure out how our efforts can be used as a starting point to improve Fujaba with regard to UML 2.0. Finally we present how we implemented the plug-in, what kind of technologies and components we used and what we will achieve with our efforts.

1. INTRODUCTION

The new upcoming version of the Unified Modeling Language (UML) [7] introduces new modeling concepts. Beside the new version of the UML there is also a new version of the Meta Object Facility (MOF) [1] which strongly depends on UML. The part of the UML specification that describes structural modeling (UML infrastructure [6]) is adopted by MOF. So MOF comprises the new UML constructs for structural modeling, too. We want to use MOF 2.0 and decided after a period of evaluating alternatives to realize our approaches in Fujaba.

One aim of our contributions for the development of Fujaba is to realize a MOF 2.0 plug-in which is able to generate metamodels for several domains. The generated metamodel implementations should comply with the Java Metadata Interface (JMI) [4] a standard provided by SUN. We want to use the generated metamodels in several fields like tool integration and re-engineering for example. Thus we have to be able to generate different kinds of metamodel implementations with a flexible code generation mechanism which uses MOF 2.0 compliant metamodels created with Fujaba as input.

The realization of our target plug-in or rather the actual stage of development as well as the benefits for Fujaba are

subjects of this paper. The new features of MOF 2.0 compared to Fujaba are described in section 2 followed by an overview of further steps of development in section 3. Section 4 gives an overview on JMI. The application of JMI is described in section 5. Finally the conclusions and future work are part of section 6.

2. FEATURES OF MOF 2.0

The new features of the MOF 2.0 plug-in compared to the actual features of Fujaba's UML class editor mainly concern associations and packages. MOF 2.0 separates between real associations and implicit relations. Each class attribute may have an opposite attribute thereby realizing a simple bidirectional association. Such implicit relations are visualized by the MOF 2.0 plug-in as usual associations are visualized in the current Fujaba version. The separation of implicit relations and real associations gets clear regarding code generation. Implicitly related attributes would be mapped on code by using referencing attributes. A real association is mapped on an own class as demanded by the JMI specification. The separation has to be reflected in visualization and therefore real associations are depicted by a diamond node according to the MOF 2.0 specification (see figure 1). Each diamond node represents an association and each line connecting the association with the associated classes represents an association end. With such a mechanism we keep the opportunity to extend the plug-in for n-ary association although in MOF 2.0 only binary associations are allowed.

Other new features of MOF 2.0 compared to Fujaba are the relations between association ends. MOF 2.0 offers the opportunity to declare an association end as redefinition, subset, or union of other association ends (see figure 1). Those relations enhance the possibilities of modeling by specifying the relation between associations which are indirectly related by the type compatibility of their associated classes. Considering the example in figure 1 it is also possible to associate an instance of class `HardwareDeveloper` with an instance of class `HardwareProject` by using association `Develop` instead of association `DevelopHardware` due to the type compatibility of the associated classes. Such an implicit relation can now explicitly be marked as redefinition or subsetting. Without those constructs it would not be possible to express that a hardware developer develops only hardware (without using OCL constraints). The redefinition of association end `developer` by the association end `hwDeveloper` ensures that a hardware developer just develops hardware projects by suppressing the instantiation of the association `develops` be-

tween an instance of class `Project` and an instance of class `HardwareDeveloper`.

Another important feature is the possibility to define an association end as subset of another association end. The subsetting of an association end causes the propagation of link instances from the subsetting association end to the subsetting association end. In the example of figure 1 the query on association `Develop` for all projects for a specific hardware developer returns all projects that have been registered as instance of association `DevelopHardware` although there have never been an instantiation of association `Develop` with those instances. The linkage between both association is caused by the subsetting.

In addition to the mechanism of subsetting it is possible to declare a superset as exclusive union of its subsets. The definition of association end `project` as union of its subsets means that a developer can either develop hardware or software. The instances of the association ends `hwProject` and `swProject` are also part of the association end `project`. The union constraints prevents that there are instances of `project` beside the instances of the subsets. For further details see [6] or [2].

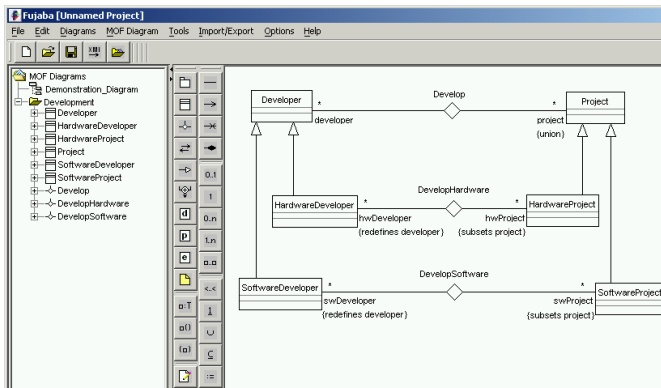


Figure 1: Screenshot of the MOF 2.0 plug-in

Furthermore MOF 2.0 offers the possibility to define dependencies between packages. Those dependencies (import, combine, merge) specify how packages are related to each other. Our work on the package editor has not proceeded far enough that an appropriate discussion is possible at the time of writing this paper. As a result of the packages and their significance in MOF 2.0 the current Fujaba package browser is not sufficient anymore. So we decided to build a new browser that supports package hierarchies and offers a higher degree of structuring.

Beside the new features which are undoubtedly an enrichment of available modeling facilities there are also useful features in the current version of Fujaba that are not available in MOF 2.0. For example, in MOF 2.0 class attributes are `public` or `private`. There is no possibility in MOF 2.0 to define an attribute as `protected` nor as `static` or `final`. That problem can be solved by the use of the adapter pattern as described in [9].

Table 2 summarizes some significant features and their avail-

ability in UML 1.x, Fujaba's UML, MOF 2.0 and UML 2.0. Obviously the most desirable modeling language concerning the widest range of modeling constructs is UML 2.0. There is a big gap between the possibilities UML 2.0 offers and the possibilities implemented by Fujaba as well as between Fujaba and MOF 2.0.

Constructs	UML 1.x	Fujaba	MOF 2.0	UML 2.0
private public	+	+	+	+
protected static	+	+		+
packages	+		+	+
package merge			+	+
kinds of package merges			+	
qualified associations	+	+		+
aggregations	+	+		+
composition	+	+	+	+
opposite - asso- ciation			+	+
n-ary associa- tions	+			+
related associa- tion ends			+	+
inheritance between associations	(+)		+	+
inner classes	(+)			+
dependencies	+			+
interfaces		+		+
reflection			+	

Table 1: Availability of modeling constructs

3. FURTHER STEPS OF DEVELOPMENT

Our work on MOF 2.0 can be regarded as an initial step for the evolution of Fujaba towards UML 2.0. Basically MOF 2.0 is a subset of UML 2.0 which covers the structural modeling of UML 2.0 (UML 2.0 Infrastructure [6]) expanded by some specific metamodeling features like extensibility and reflection. MOF 2.0 is a metamodeling language that can be used to specify modeling languages. The best known instances of MOF 2.0 are UML 2.0 as well as MOF itself. The features of MOF are limited compared to UML 2.0 but optimized with respect to the demands of metamodeling. So MOF represents a starting point in the definition of UML 2.0 and therefore can also be a good starting point for implementing UML 2.0. The mapping from MOF to Java (JMI) is designed regarding the demands of metamodels.

Thus a MOF 2.0 compliant Meta-CASE tool with a JMI compliant code generator is an appropriate basis to realize an extensible UML 2.0 CASE tool. Regarding those relations as described before we propose a scenario of evolution as follows.

1. First we realized an editor that is capable of generating code from an intersection of the modeling constructs of MOF 2.0 and the current Fujaba version. The graph transformations can be applied to the resulting modeling constructs. In the future we are interested to optimize the generated code with respect to the demands of embedded systems.

2. The second step will be the adaptation of the graph transformation for the generation of JMI compliant code. That code will be used for Meta-CASE applications.
3. The next major step is to expand the set of modeling constructs from an intersection of MOF 2.0 and Fujaba to a union of the concepts of both languages. There should also be both kinds of code generation (JMI compliant code for Meta-CASE applications and target code for embedded systems) with regard to the different kinds of application.

Those steps are planned for the near future and are intended to result in a Fujaba that is based on UML 2.0 Infrastructure. At least we propose to keep MOF and UML separately due to different code generations and different purposes. For both versions of the UML Infrastructure the graph transformation has to be adjusted to the new concepts.

4. METAMODELING WITH JMI

One of the central aspects of the implementation of our MOF plug-in is the compliance to the Java Metadata Interface. Therefore we give a short overview of JMI. JMI defines a structure for the creation, storage, access and discovery of metadata by specifying a Java language mapping for MOF 1.4. JMI provides a common Java programming model for handling of metadata. This is done by the description of a set of interfaces which represents the reflective parts of MOF as well as the structural characteristics of MOF instances. The interfaces that cover the structure of the metamodels are divided into four categories. An example for a concrete JMI mapping which covers instances of all four categories is depicted in figure 2. It shows the mapping of a package with a binary association between two classes on JMI. The four categories are:

Package Objects create and manage instances of all included metaclasses. Instances are class proxy objects, association objects and package objects for nested packages. The package *MetaModel* in figure 2 is mapped to the interface *MetaModel* and the appropriate implementation *MetaModelImpl*. The instance of the package implementation is the initial point for the instantiation of the metamodel. There are accessor methods for all class proxy objects as well as for all association objects.

Class Proxy Objects act as a factory and as a container for creating and storing instance objects. There is only one class proxy object for each metaclass. The class proxy object of the metaclass *Operation* in figure 2 is of the type *OperationClass*. The class proxies are managed by the superior package instance.

Instance Objects represent an instance of the appropriate metaclass. Each instance object is created and stored by a class proxy object.

Association Objects are just like class proxy objects container for the handling of association instances. There is only one association object of the same type at runtime. The association objects are created and stored

just like the class proxies by the superior package. Association instances are stored by the use of an unspecified link object contrary to the handling of class instances. In figure 2 the association *Has* is mapped to the interface *Has* with the appropriate implementation *HasImpl*.

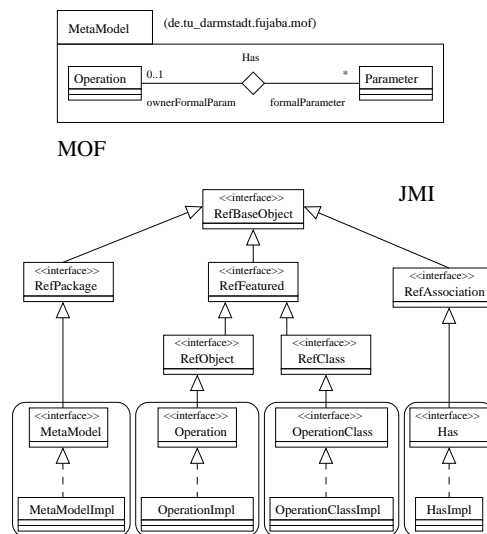


Figure 2: Example of JMI code mapping

For example the instantiation of the class *Operation* assumes an instance of the package implementation (*MetaModelImpl*) which provides access to the class proxy objects. The class proxy object contains the factory method *createOperation* that returns an instance object. Finally, the instance object can be used to store instance scoped attributes. The instantiation of an association requires just like the instantiation of classes the request of an association object from the package object. Association instances (so called links) can be handled by using the association object's methods like *add*, *remove* etc.

A metamodel conforms to the JMI specification as long as the interfaces are satisfied. The implementations of the interfaces are just informally described and therefore may vary depending on different usages of the metamodel.

The currently existing JMI specification is designed to map MOF 1.4 compliant metamodels to Java. So the actual JMI specification does not cover the latest version of MOF. There are a lot of features in MOF 2.0 that force an extensive revision of JMI. For example, the current storage of association instances is not able to cover all features of MOF 2.0, actually it is not even able to cover MOF 1.4 properly. Such a revision is an essential motivation of our work on JMI.

5. A JMI-COMPATIBLE MOF 2.0 PLUG-IN

One of the central components of the MOF 2.0 plug-in is a code generator for JMI compliant metamodels. We need a code generator that is highly flexible and easily configurable for several target applications. Such a generator is the MOF Metamodeling Tool (MOmo) Compiler [3]. The MOmo Compiler is able to generate JMI compliant MOF

metamodels from XMI [8] files. Its great advantage lies in its modularity concerning the architecture as well as the handling of several styles of target code by applying different sets of templates. Hence, the appropriate way to realize the Fujaba MOF plug-in was to bootstrap the metamodel by using MOmOC and a special rudimentary MOF 2.0 metamodel. The rudimentary MOF 2.0 metamodel for bootstrapping consists of just one package including the minimal set of necessary features. The distribution of the MOF 2.0 specification over several packages as done in the specification is planned for further iterations.

The first version of the plug-in's metamodel has been modelled in Rational Rose and passed to the MOmOC code generator via export of UML XMI. So the first metamodel that makes use of the new features of MOF 2.0 as done in the specification may not be introduced before the first bootstrap iteration due to the lack of MOF 2.0 features in common modeling tools. The way of realizing the plug-in is depicted in figure 3. Figure 3 shows a scenario with some of the main functions of the plug-in. The functionalities realized yet and needed for bootstrapping are marked in gray.

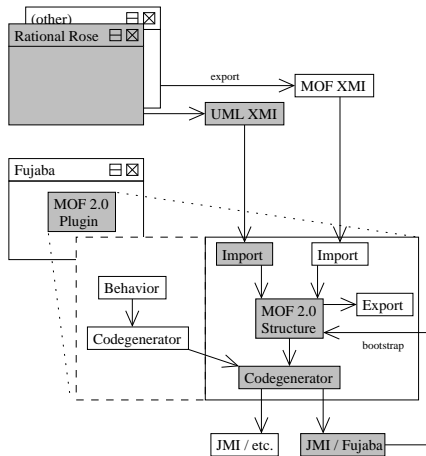


Figure 3: Scheme of the realized bootstrap scenario

The actual version of the plug-in at the time of writing this paper consists of a generated JMI compliant MOF 2.0 metamodel with a graphical editor for instantiating the most important elements except packages and their dependencies. The limitations are caused by the capabilities of the graphical editor which is based on the current editor for class diagrams in Fujaba. In general the metamodel is fully usable. The MOmOC compiler has been modified to generate a metamodel with an implementation compatible with Fujaba. The code generator has been integrated into the plug-in and operates on the plug-in's metamodel. So the plug-in is able to generate its own metamodel. The further improvement of the metamodel will be done with the plug-in itself as soon as the graphical editor's degree of usability permits it.

The first versions of the generated metamodel consisted of an implementation that did not consider any cooperation with the graph transformation subsystem of Fujaba. It consisted just of an independent metamodel with all necessary actions and unparse modules that were essential as designated by Fujaba's plug-in mechanism. The metamodel im-

plemented all interfaces demanded by JMI as described in section 4. But even such a rudimentary integration of a different metamodel caused a problem with the currently available plug-in mechanism. In the special case of a JMI compliant metamodel the instantiation is done by a factory method and not by a constructor as supposed by Fujaba's loading mechanism. So we had to modify Fujaba in a separate branch as described in [9] even in such an early state of integration.

Parts of our research activities also require the specification of behaviour in combination with the new features of MOF 2.0. Thus we had to enhance the metamodel's implementation with the intent to adopt Fujaba's graph transformation engine. The implementation has to apply a mechanism for cooperating with the graph transformation code generator. Such a mechanism is described by the adapter pattern [5]. The used adapter is depicted in figure 4. The former implementation `OperationImpl` of the JMI interface `Operation` as exemplarily used in figure 4 is replaced by a subclass `OperationAdapter` which implements all functions as demanded by JMI as well as all function demanded by Fujaba's graph transformation. There is an inheritance between Fujaba's graph transformation classes and the implementation of the metamodel. The instance of the related proxy class (`OperationClassImpl`) instantiates the adapter instead of the class `OperationImpl`. This is no violation of the JMI specification because the adapter still implements the JMI interfaces. For further details see [9].

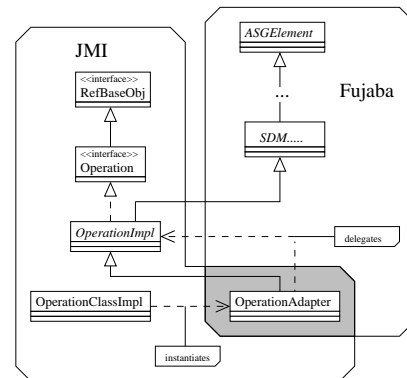


Figure 4: JMI Fujaba Adapter

6. CONCLUSIONS

The first version of our MOF 2.0 plug-in uses a rudimentary, but JMI compatible MOF 2.0 metamodel. Basically, the editor is able to generate its own metamodel. So further iterations of the metamodel will be done by using the plug-in itself. It is our demand to satisfy the MOF 2.0 specification including packages and their relations in detail although not in the first iteration. The primary concepts of MOF 2.0 have already been taken into account. All other features will follow in further iterations.

The plug-in is able to import UML XMI. There was no need for an import of MOF XMI yet. But the enhancement to MOF XMI is already planned for the future, when the plug-in will be able to write MOF XMI. It offers a code generation mechanism that is very easily expandable for the needs of

other target frameworks just by maintaining several sets of templates. In this respect the integration of the Fujaba code generation is still an open issue.

Furthermore it is our intention to integrate an OCL compiler to enhance modeling capabilities as well as to improve the bootstrap process by considering the constraints of the MOF 2.0 specification. Additionally we will concentrate on the improvement of the generated metamodels.

Finally, there are three major advantages the Fujaba community will benefit from. First of all our new MOF plug-in offers the new features of MOF 2.0 and might act as a basis for upgrading Fujaba class diagrams to UML 2.0. One fundamental disadvantage of the current Fujaba version is the missing package concept which complicates the use of Fujaba in large projects. The package concept implemented by the MOF plug-in solves this problem and offers an easy way to organize large projects. Last but not least the compliance to standards like MOF and JMI opens up new application domains and therefore might expand the Fujaba community.

7. REFERENCES

- [1] Adaptive Ltd, Ceira Technologies Inc., Compuware Corporation, Data Access Technologies Inc., DSTC, Gentleware, Hewlett-Packard, International Business Machines, IONA Technologies, MetaMatrix, Rational Software, Softeam, Sun Microsystems, Telelogic AB, Unisys, and WebGain. *Meta Object Facility (MOF) 2.0 Core Proposal*, April 2003. ad/2003-04-07.
- [2] C. Amelunxen, L. Bichler, and A. Schürr. Codegenerierung für Assoziationen in MOF 2.0. In *Proceedings of the Modellierung 2004*, volume P-45 of *Lecture Notes in Informatics*, pages 149–168. Gesellschaft für Informatik, March 2004.
- [3] L. Bichler. Tool Support for Generating Implementations of MOF-based Modeling Languages. In J. Gray, J.-P. Tolvanen, and M. Rossi, editors, *Proceedings of The Third OOPSLA Workshop on Domain-Specific Modeling*, Anaheim, USA, October 2003.
- [4] R. Dirckze. *Java™ Metadata Interface (JMI) Specification, Version 1.0*. Unisys, 1.0 edition, June 2002.
- [5] Erich Gamma AND Richard Helm AND Ralph Johnson AND John Vlissides. *Entwurfsmuster*. Addison-Wesley, 1996.
- [6] Object Management Group. *Unified Modeling Language: Infrastructure, Version 2.0*, September 2003. ptc/03-09-15.
- [7] Object Management Group. *Unified Modeling Language: Superstructure, Version 2.0*, April 2003. ad/2003-04-01.
- [8] Object Management Group. *XML Metadata Interchange (XMI) Specification, Version 2.0*, May 2003. formal/2003-05-03.
- [9] T. Röttschke. Adding pluggable meta models to FUJABA. In *Proc. Fujaba Days 2004*, 2004. To appear.