

Graph Transformations with MOF 2.0

Carsten Amelunxen, Tobias Rötschke, Andy Schürr
Technische Universität Darmstadt
Institut für Datentechnik, FG Echtzeitsysteme
Merckstr. 25
Darmstadt, Germany

[carsten.amelunxen|tobias.roetschke|andy.schuerr]@es.tu-darmstadt.de

ABSTRACT

The MOFLON framework aims to contribute to various aspects of meta model-based software engineering on top of the Fujaba Tool Suite. Currently, Fujaba operates on an internal UML 1.4-like meta model to represent graph schemata. Graph transformations are formulated in terms of this internal model. With the upcoming Fujaba 5 release, plugins will be able to provide their own meta models which are mapped onto the internal one. MOF 2.0 as the standard meta modeling language provides improved new concepts compared to older versions of UML and MOF. By implementing a MOF 2.0 editor and a related JMI compiler as plugins for Fujaba, we are now able to define graph schemata with MOF 2.0. In this paper, we discuss the implications of our choice of MOF 2.0 as schema language on the existing graph transformations in Fujaba and we sketch a roadmap how to adapt the Fujaba SDM editor and code generator appropriately.

1. INTRODUCTION

Meta modeling languages are used to describe other modeling languages. The Object Management Group (OMG) adopted MOF 2.0 [9] as standard meta modeling language. To be able to work with meta-modelled languages, tool providers need to map meta models on source code. Sun defined the Java Metadata Interface (JMI) [3] as standard mapping of MOF-compliant meta models to Java code. As we see interesting opportunities in the combination of MOF 2.0 as meta modelling language and Fujaba for graph transformations, we decided to bring both together.

During the last months, we have been busy on the first release of the MOF 2.0 plugin for Fujaba which, in more general terms, is a part of the MOFLON meta modeling framework¹, that had been proposed in [2] for the first time. On the last FujabaDays, we discussed the new association concept [1] of MOF 2.0 and how we can improve Fujaba so that the internal meta model used for graph transformations is separated from the external meta model defining the editor features visible to the user [12].

Meanwhile we have made major progress with the MOF 2.0 editor and the accompanied JMI code generator. We are able to bootstrap the creation of the underlying MOF 2.0 meta model. A MOF instance can be either directly created using the MOFLON/Editor plugin for Fujaba or drawn in Rational Rose, exported as XMI and finally imported in

the plugin. Using our MOFLON/Compiler plugin, we can not only provide JMI interfaces, but also an implementation that is able to store a consistent model with respect to features like subsetting, redefinition, union, composition, and multiplicities.

2. MOFLON INSIDE FUJABA

Figure 1 provides a sketch of the relationships between Fujaba and MOFLON. The MOFLON/Editor plugin realizes a MOF 2.0 meta model editor based on JMI-compliant MOF meta-meta model, and its adaption to the internal Fujaba meta model. The MOFLON/Compiler plugin adds a JMI-compliant Java code generator for MOF 2.0 instances created using the editor component. The MOFLON/RoseXMI plugin allows to import UML class diagrams from Rational Rose using XMI. During the import, a corresponding MOF 2.0 instance is created by our plugin, which can be further processed using the editor and compiler component.

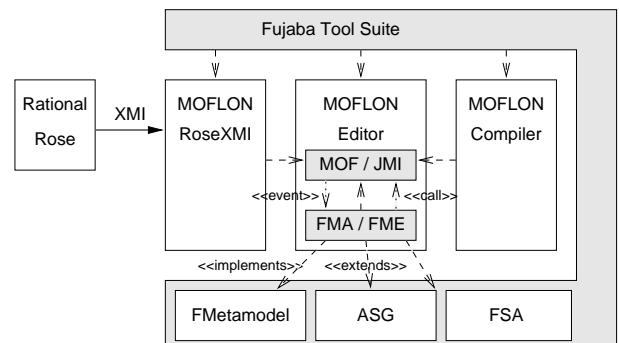


Figure 1: Sketch of MOFLON dependencies

The Fujaba framework provides the internal meta model (*FMetamodel*) together with a basic abstract implementation (*ASG*). The MOFLON/Editor plugin contains a generated JMI implementation of the MOF 2.0 meta model (*MOF*). This implementation fires MDR-like [7] events whenever a model instance is changed, and hence has no static dependencies on any Fujaba packages.

The internal Fujaba metamodel and the MOF metamodel are coupled via the Fujaba MOF Adapter (*FMA*) layer. Within the MOFLON/Editor plugin, every diagram item in terms of the internal Fujaba meta model is an adapter referring to a MOF element as adaptee (cf. Object Adapter

¹<http://www.moflon.org>

design pattern in [4], p. 141). For every adapter there is an adequate unparse module to visualize the item in the editor using the Fujaba Swing Adapter (*FSA*) layer. It must be mentioned that we proposed class adapters inheriting from *ASGElement* in [12] to implement the JMI interface, but we decided that our default implementation should be self-contained.

Method invocations to an adapter are directly delegated to the corresponding MOF element, which contains the actual state of the item. In the opposite direction, the Fujaba MOF EventListener (*FME*) package receives change events from the MOF implementation and fires the corresponding property change events to update the GUI.

In some cases (e. g. association ends, generalizations, mutual references), a number of adapters correspond to a single MOF element. In such a case, the adapters must remember their relationships with other adapters referring to the same MOF element. Besides, the adapters keep information related to those parts of the Fujaba meta model, that do not correspond to standard UML or MOF elements (e.g. projects, diagrams, files, and programming language-related data).

2.1 MOFLON vs. Fujaba

In the following the new features of MOFLON compared to Fujaba are listed, i.e. the delta from the Fujaba implementation of UML 1.4 class diagrams to MOF 2.0.

- **Packages** are namespaces in MOFLON and can be combined by two kinds of package merges². Each package holds a diagram. The package concept replaces Fujaba's view diagrams.
- **Attributes** with collections as values can be subsetted, redefined, unique, ordered or an union of all their subsets. As a result of combining ordering and uniqueness, multi-valued association ends are realized by four different kinds of collections³.
- **Associations** are distinguished into "real" associations and (mutual) references. References correspond to associations in Fujaba as they are mapped on attributes whereas associations are mapped on classes. Navigable association ends are treated like attributes of the appropriate classes. Thus the features concerning attributes are also relevant for navigable association ends. For the purpose of graph transformation both concepts of references and associations can be treated equally since there are no link objects for associations.
- **Navigability** specifies how association instances can be queried in MOFLON. Navigable association ends can be handled like attributes with the appropriate access methods. This easy access is only available for navigable association ends. Non-navigable association ends can be queried by using the methods of the class representing the association. By making use of the

²There are two kinds of package merge (merge and combine). See [9] and [10] for details.

³Set, OrderedSet, Bag, Sequence

class for an association each link can be queried independently of the navigability. Associations are always navigable in terms of Fujaba.

- **Composition** is semantically relevant since all children of a composite are deleted together with their parent.
- **Multiplicities** are dynamically evaluated similar to composites. For a given object, multiplicity determines the required number of linked objects of the associated class. If the number of linked objects falls below the lower bound the given object is deleted, as in our opinion it cannot exist if the lower bound is violated. This evaluation is just processed with each deletion of an object since not every update can result in a consistent meta model instance. The demand of a consistent meta model instance after each deletion has been proven to be very useful. A more sophisticated approach is a matter of current work.

3. NEW FEATURES OF MOF 2.0

Figure 2 shows an example from the domain of reengineering in which we demonstrate some MOF 2.0 features that are new compared to UML 1.x and previous MOF versions. The *FileSystem* package defines the general concepts of UNIX-like file system with files and directories. The second package specializes the first one for the purpose of typical C programming.

Note, that we drew the diagrams for package contents inside the package to achieve a more compact visualization. In the MOFLON/Editor component, a dedicated diagram exists for every package. Package dependencies are drawn in the diagram of the parent package or in the root diagram for root packages.

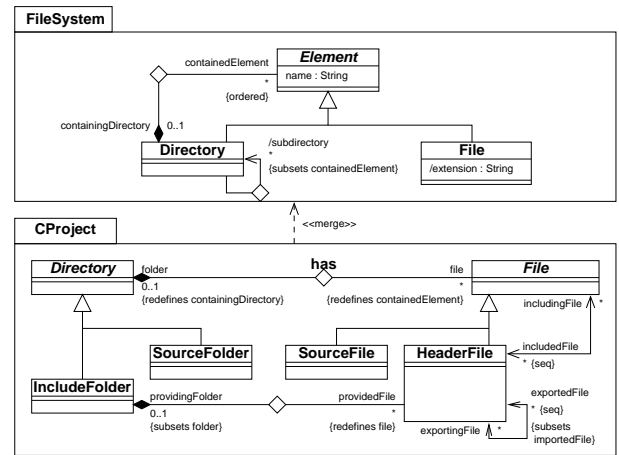


Figure 2: A sample MOF 2.0 diagram

To start with package relationships, consider the *merge* dependency between package *FileSystem* and *CProject*. This is one of the new package relationships, that are coming with MOF 2.0⁴. Every public *Element*⁵ of package *FileSystem* is

⁴Other package relationships are *import* and *combine* which have different semantics

⁵*PackageableElement* to be exactly

automatically available in the namespace of package *CProject*. If a new element with an existing name is introduced, a generalization relationship to the existing element in the merged package is established by the semantics of the merge dependency. So *CProject::Directory* is a subclass of *FileSystem::Directory* and *CProject::File* is a subclass of *FileSystem::File*. As they are different elements, *CProject::File* may be abstract, while *FileSystem::File* is not. In case of a combine relationship the elements of *FileSystem* would have been (deep) copied into *CProject*. Equally named classes would have been merged instead of inherited. In case of an import relationship the elements of the imported package would have just been added to the namespace of the importing package.

Considering the revised definition of navigability in MOF 2.0 in combination with the relations between association ends, associations evolved to a more sophisticated concept. In MOFLON, associations are only allowed between classes and always drawn with a diamond to distinguish them from (mutual) references, like the relationship between *File* and *Headerfile*. References are a means to define "light-weight" unidirectional relationships between classes or data types and other classes without instantiating associations. Pairs of mutual references are treated as one "light-weight" bidirectional relationship. Taking into account that adornments like aggregation and composite have no semantics in Fujaba, these "light-weight" relationships correspond closely to Fujaba's association concept.

In UML 1.x, association ends are always unique, but in MOF 2.0, they may be non-unique as well. In combination with ordering, multi-valued association ends have to be realized by four different kinds of collections.

Probably the strongest improvement in MOF 2.0 is the possibility to define union-, subset-, redefinition-relationships between properties and hence association ends. In figure 2, the property *subdirectory* is defined as *subset* of *containedElement*. So only *containedElements* of type *Directory* can be a *subDirectory*. But without further constraints, there might be contained *Directories* that are not considered *subDirectories*. A *redefinition* relation allows to express stronger restrictions. The property *providedFile* redefines *file* and hence a *IncludeFolder* may only contain *files*, i.e. *providedFiles* of Type *HeaderFile*.

Having discussed the new concepts in MOF 2.0, we should mention that some features of UML 1.x do not exist in MOF 2.0. The most important features are stereotypes and qualifiers, but in our opinion, these are not essential for the purpose of meta modeling. The lack of stereotypes is barely a problem, as Fujaba only interpretes stereotypes to distinguish classes, interfaces, data types, references and Java Beans. In MOF 2.0, there are dedicated meta classes for most of these stereotypes, i.e. classes, data types, primitive types, enumerations and element imports. Qualifiers determine keys to quickly retrieve association ends from the collections implementing them. As meta modelling is about defining modelling languages, the name of a model element can be assumed to serve as qualifier where needed. So explicit qualifiers are not needed for meta modelling. Obviously, every known MOF or UML meta model has been de-

finied without using qualifiers. Even if these meta models are far from perfect, most issues result from lacking constraints rather than missing qualifiers.

4. ADAPTING THE SDM EDITOR

Based on the internal meta model which is introduced in Fujaba 5, either UML or MOF can be used to define schemata for graph transformations. As we have seen in section 3, MOF provides some extensions that must be integrated in the graph transformation concept. In this section, we discuss how MOF 2.0 as schema language influences graph transformations in Fujaba.

Graph transformations in Fujaba, also referred to as Story Driven Modelling (SDM), require only very few concepts. On the one hand, activities, transitions and related guards allow to define the control flow of complex graph transformations. Choosing MOF 2.0 as schema language has no impact on these concepts at all. On the other hand, simple graph transformations are visually defined using a variation of collaboration diagrams. They consist of objects, links and multi-links, which are slightly affected by changing the schema language.

First of all, the advanced package concepts allow a better means to find available types for objects, associations and attributes than the existing solution with diagrams and views does. Namespaces and visibilities are taken into account and less name clashes occur, which result in nasty effects in the existing Fujaba implementation. For instance, a package merge as used in 2, effectively produces implicit generalization relationships between classes with identical names in both packages. When editing Story Diagrams, these additional generalization relationships must be considered during the creation of links, i.e. when the choice of possible edge types is determined.

The different kinds of inter-class relationships (associations and (mutual) references) in MOF 2.0 do not have impact on links in the first place. But when taking generalization, union, subsetting and redefinition into account, there are some affects on links.

The meta model implementation automatically maintains derived links which result from subsetting or redefinition. For instance, if an association end is marked as *subset* of another more general association end, changes to the subsetting end are automatically propagated to the subsetted end due to our meta model implementation. Considering the example in fig. 2, after creating a *subdirectory* link between two *Directories*, the *containedElement* link between *Directory* and *Element* does automatically exist as well and can be queried accordingly.

Next, if a general association end is marked as *union*, the association may not be directly modified by graph transformation rules. However, propagation from special association ends is still effective. Hence, the association becomes "abstract" so to speak. Unlike *redefinition* of association ends which will be discussed in section 5, the restrictions of *union* association ends can be analyzed statically.

In Fujaba, links in the set of Edges *E* have approximately

the following form [13]:

$$E \subset N \times EL \times I \times Q \times N \quad (1)$$

The first element denotes the source node, the second the edge label, the third the index (applies only for ordered associations), the fourth the qualifier (applies only for qualified associations), and the last the target node. Edge labels correspond to associations in the class diagram. Associations can be either plain, ordered or sorted, although UML like MOF 2.0 assigns this feature to association ends instead. However, MOF 2.0, does not feature sorted association ends.

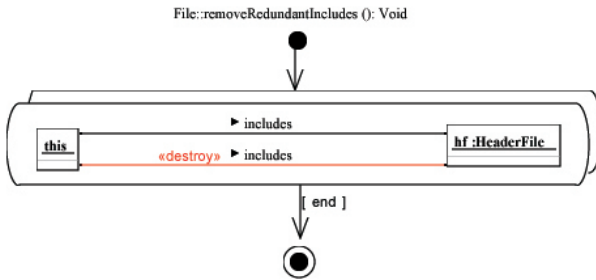


Figure 3: Remove redundant includes

Figure 3 shows an example of how non-unique association ends can be used to define a useful graph transformation for the schema in figure 2. If a *File* includes a *HeaderFile* more than once, the redundant includes should be deleted. Currently, Fujaba does not support these kinds of rules. The link is matched only once and then deleted. As a result, all include links will be removed.

In the case of ordered, non-unique association ends (sequences), multi-links might ostensibly allow to distinguish both links using multilinks. But a closer look to the generated source code in combination with the implementation of the related collection class reveals that in the general case, the multi-link cannot be matched.

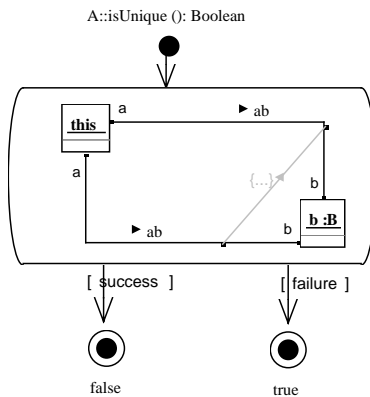


Figure 4: Detect non-unique links

Consider the example in fig. 4, and the resulting generated, but slightly condensed source code in fig. 5. The rule tries to detect two different links of the same *ordered* association between the same objects *a* and *b*. According to [13], p.

103, this should be possible: "..., in case of sorted or qualified associations there may exist multiple edges with the same label connecting nodes n1 and n2 as long as they carry different values for their index i or their qualifier q." We believe, that "sorted" in fact means "ordered", has sorted association ends are implemented using *FTreeSets* rather than *FLinkedLists* and hence no indices exist.

```
public boolean isUnique()
{
    boolean fujaba__Success = false ;
    B b = null ;
    Iterator iterB = null ;

    try {
        fujaba__Success = false ;

        // multilink bind b : B iMultiLinkSearchNormTop
        iterB = this.iteratorOfB() ;
        while (!fujaba__Success && iterB.hasNext()) {
            try {
                b = (B) iterB.next () ;

                // check To-Many-Link 'b' between this and b
                JavaSDM.ensure (this.hasInB(b)) ;

                // check multilink b to b
                JavaSDM.ensure (this.isBeforeOfB (b, b)) ;

                fujaba__Success = true ;
                // iMultiLinkSearchNormBottom
            }
            catch ( JavaSDMException e ) {
                // try catch
            } // while 00Variable[B,00VariableType[12],]
        }
        catch ( JavaSDMException e ) {
            fujaba__Success = false ;
        }

        if ( fujaba__Success ) {
            return (false) ;
        }
        else {
            return (true) ;
        }
    }
}
```

Figure 5: Source code for fig. 4

The crucial part is the call *A.isBeforeB(b,b)*. Tracing the call, it results in another call of *FLinkedList.isBefore(b,b)* (cf. fig 6), which obviously results in *false*, if *left* and *right* are identical as in this case, as *indexOf(b)* always finds the *first* occurrence of *b*. The rule would probably work, if a method like *nextIndexAfter(b,b)* would be invoked for the second argument.

```
//returns true if left object is before right object
public boolean isBefore (Object left, Object right)
{
    return (indexOf (left) < indexOf (right));
}
```

Figure 6: Source code for *FLinkedList.isBefore(l,r)*

As a result, association ends in Fujaba must always be unique with respect to the current implementation. Even the exception of ordered associations mentioned in [13] seems not to be implemented correctly. So dealing with non-uniqueness

requires modifications to Fujaba’s graph model, although the precise impact is still under investigation. In case of ordered, non-unique association ends, this only seems to be a matter of improving the implementation of access methods. In case of unordered, non-unique association ends however, the formal definition of graph model would be affected. This does not necessarily mean to give edges an identity, but we would at least have to count plain links of the same edge type between identical nodes (*LC*), like

$$E \subset N \times EL \times I \times Q \times \mathbf{LC} \times N \quad (2)$$

The semantics of the rule in fig. 3 would be, that there is at most one tuple for every pair of instances of *File* and *HeaderFile*, with a number *c* on the last but one position, representing the number of links. The rule matches, if such a tuple exists for *this* and *hf* and $c > 1$. After the transformation, the tuple is modified and $c' = c - 1$ ⁶.

5. BRINGING SDM AND JMI TOGETHER

The Java Metadata Interface (JMI) [3] is a standardized mapping of MOF compliant meta models onto Java. As the Java representation of MOF it specifies the generation of tailored interfaces for the creation and access of meta data as well as a set of reflective interfaces for a unified discovery of meta data. Furthermore, a JMI compliant meta model has to provide XMI [11] import and export functionality to facilitate an easy exchange of meta data. JMI compliant meta models can be implemented through various strategies and technologies. An implementation is compliant to the standard as long as the interfaces are met. The standardized interfaces facilitate an easy exchange and adaptation of meta models which make JMI a beneficial standard.

Thus, for the purpose of MOF based graph transformation JMI is the appropriate choice. Currently, MOFLON is able to generate an JMI compliant implementation for MOF 2.0 compliant (meta) models. Future versions will be able to complete that static implementation with the code which is generated by Fujaba SDM. The static part of the generated meta models will be generated by MOFLON’s code generator and the dynamic parts with the graph rewriting engine of Fujaba. The current version of Fujaba uses proprietary Java interfaces for the static model representation (e. g. for multi valued association ends as listed in Table 1) which are used by Fujaba’s graph transformation. An utilization of Fujaba’s graph transformation engine for the needs of MOF affords first of all a mapping between the interfaces generated by Fujaba and the interfaces demanded by JMI.

5.1 Mapping of Fujaba’s interfaces on JMI

The major differences⁷ between JMI and the code generated by Fujaba consist in the different handling of packages, class and association instances. These are differences on a large

⁶In the practice, this association should be ordered, as it matters which include has to be removed. But we decided not to introduce another example to discuss multi-relations here.

⁷The following comparison relates to the tailored (typed) interfaces of JMI because the reflective interfaces can easily be mapped on the tailored interfaces.

scale, but with just a limited impact on Fujaba’s code generation as the static structure of the code will be generated by MOFLON. The most relevant parts concerning the dynamic implementation are the instantiation and deletion of class and association instances as well as the possibilities to navigate on the instance level.

Table 1 gives an exemplary overview how the mapping for the relevant parts might look like. For instance, JMI demands a more complicated class instantiation based on proxy classes representing the meta class. From the view of a code generator, the instantiation by calling a factory method is just syntactically overhead compared to the direct instantiation using the `new` operator. The mapping for the deletion of instances and for single valued association ends is even more simple since the methods in both interface variants are nearly or even exactly the same. Multi-valued association ends are treated in JMI by making use of Java’s `Collection` interface. Thus association ends are treated like ordinary attributes.

This feature reduces the interface to one single method but provides even more functionality through the `Collection` interface than the expanded interface generated by Fujaba does. There are other implementation concepts [6] that also reduce the interface to a single method. The main difference is that in [6] the links are stored in generic association end objects by using Java reflection for the creation of the backlink. In contrast, MOFLON generates a class for each association which centrally maintains the backlinks by using a listener concept. Due to the centralized storage the usage of Java reflection is not necessary. Nevertheless, the mapping still is no problem at all since each method of Fujaba’s interface has a counterpart in the `Collection` interface. In the case of an ordered multi-valued association end⁸ Fujaba generates convenience methods that do not have a counterpart in the `List` interface which JMI uses for ordered association ends. Those are methods that operate relative to the position of another linked object (e.g. `addBeforeOfX`). They will have to be replaced by an additional workaround of that parts of the implementation which use the non-mappable methods.

Considering navigability, some further aspects have to be taken into account. In terms of Fujaba, every association in MOFLON is navigable. The difference is, that navigable associations in MOFLON can be accessed as described in Table 1, whereas non-navigable associations can only be accessed by using the class representing the association. As a consequence, the call for accessing associations is expanded in such a way that the class representing the association has to be fetched from the package extent. After that, the handling of association instances is the same as for navigable associations. In general the interface mapping does not cause major problems at all.

5.2 Code generation for MOF 2.0

There are five major aspects that have to be considered as already mentioned in section 3. The package merge as one of the most significant features of MOF 2.0 demands no change in Fujaba’s code generation subsystem for graph transfor-

⁸The listing of that case is omitted.

	Fujaba	JMI
	<code>new Class()</code>	<code>package.getClass().create()</code>
	<code>removeYou()</code>	<code>refDelete()</code>
0..1	<code>getX()</code> <code>setX(..)</code>	<code>getX()</code> <code>setX(..)</code>
0..n	<code>addToX(..)</code> <code>removeAllFromX()</code> <code>hasInX(..)</code> <code>iteratorOfX()</code> <code>removeFromX(..)</code> <code>sizeOfX()</code>	<code>getX()</code> . <i>java.util.Collection</i> <code>add(..)</code> <code>addAll(..)</code> <code>clear()</code> <code>contains(..)</code> <code>containsAll(..)</code> <code>equals(..)</code> <code>hashCode()</code> <code>isEmpty()</code> <code>iterator()</code> <code>remove(..)</code> <code>removeAll(..)</code> <code>retainAll(..)</code> <code>size(..)</code> <code>toArray()</code> <code>toArray(..)</code>

Table 1: Comparison of code generated by Fujaba and JMI

mation since both kinds of package merge can be unfolded into regular model instances. After the process of unfolding the model instance is free of any package merges. A preprocessor which is part of the MOFLON/Compiler will execute this task directly before code generation. Thus the graph transformation will not even notice the existence of a package merge. After the code generation all changes in the meta model instance will be rolled back to prevent changes in the model due to the process of code generation.

Beside the package merge the subsetting and redefinition of association ends are major differences between Fujaba and MOF 2.0. Subsetting and redefinition are specified on the instance level and therefore potentially relevant for graph transformation. The subsetting of association ends causes instances of the subsetting association to be part of the extent of the subsetted association. Thus, the dynamic semantics of an ordinary subsetting do not cause any kind of restriction. The combination with `union` respectively with derivation in general at least demands a static analysis, which we have already discussed in section 3.

By definition, the redefinition of association ends causes the redefining and the redefined association end to share exactly the same set of links. Due to this constraint the instantiation of the redefined association is restricted. That restriction demands a runtime analysis to prevent forbidden instantiations as we will demonstrate using the *CProject* example. In Fig. 2 the redefinition of the association end *file* by the association end *providedFile* prevents an instance of association *has* between an instance of class *IncludeFolder* and an instance of class *File* since the collections of the association end *file* and *providingFile* for a given instance of *IncludeFolder* need to be the same. Thus, a rule as depicted in Fig. 7 may not be executed in the case that the rule is matching

an instance of class *IncludeFolder* with an *element* which is not of type *HeaderFile*, whereas an execution in any other case does not cause any problems. Note, that the general rule defined for the package *FileSystem* is interfered by the more special package *CProject*. But such a case can only be detected during runtime and not by static analysis. In our opinion, wrong usage of redefined association ends is a specification error and should result in a runtime exception.

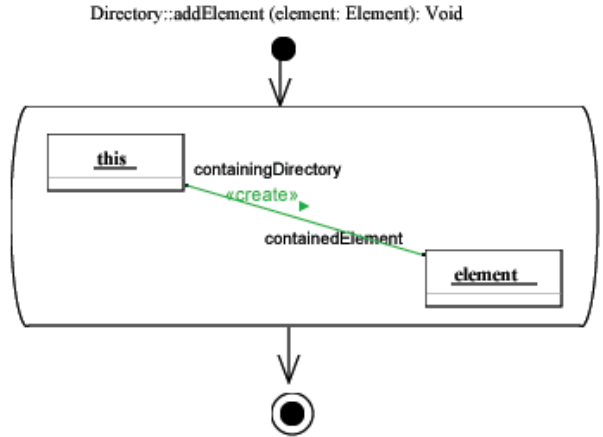


Figure 7: A general rule affected by redefinition

Other features whose compliance can only be checked at runtime are the upper and lower bounds of association ends and the composite constraint. Currently, Fujaba ignores the dynamic semantics of those features, whereas MOFLON at least tries to keep models consistent by deleting those objects whose lower bounds and composite constraints are not met. Such a solution guarantees consistent meta model instances in cases without an automated transformation. During automated transformation temporary inconsistent meta model instances might be desirable. Therefore, in future versions a sophisticated constraint evaluation mechanism combined with repair actions comparable to similar concepts in PROGRES [8] will control the compliance of any kind of constraint. This mechanism will keep the graph rewriting engine free of any contact with constraints.

A last aspect that has to be considered is the different usage of MOFLON's kinds of associations. Since both associations and (mutual) references in MOFLON are mapped on associations in Fujaba, stereotypes have to be used to control Fujaba's code generation regarding the usage of different code variants for associations. The code generation needs to be flexible enough to take this aspect into account.

6. SUMMARY

In this paper, we have discussed the implications of using MOF 2.0 as schema language for Fujaba graph transformations. This adaption involves some modifications of different parts of Fujaba which are summed up in Table 2. As already indicated in [13], the introduction of unique association ends requires modifications of the graph model. Features like `union` require additional static analysis rules, whereas others (e.g. redefinition) demand an analysis during runtime. The remaining features are already covered by our implementation of the MOF meta model.

	SDM Editor	SDM Compiler	MOF Compiler
Packages	static analysis	✓	preprocessing
Attributes			
union	static analysis	✓	✓
subset	✓	✓	runtime propagation
redefines	✓	(✓) runtime exeception handler	runtime propagation
unique ordered	✓	?	✓
unique ordered	✓	?	✓
Navigability	✓	parameterized code generator templates	✓
Composition	flexible constraint checking and repair action concept		
Multiplicity	flexible constraint checking and repair action concept		

Table 2: Impacts of the adaption of MOF 2.0

Apart from the ongoing effort to integrate MOF 2.0 in the Fujaba 5 main branch, the next steps from our point of view are the integration of OCL constraints with repair actions and triple graph grammars [5]. Besides, we have to put some more effort in the correct implementation of namespaces defined by packages.

7. REFERENCES

- [1] C. Amelunxen. Building a MOF 2.0 Editor as Plugin for FUJABA. In A. Schürr and A. Zündorf, editors, *FUJABA Days 2004*, volume Technical Report tr-ri-04-253, pages 43–47. Universität Paderborn, 2004.
- [2] C. Amelunxen, A. Königs, T. Röttschke, and A. Schürr. Adapting FUJABA for Building a Meta Modelling Framework. In H. Giese and A. Zündorf, editors, *FUJABA Days 2003*, number tr-ri-04-247 in Reihe Informatik, pages 29–34. Universität Paderborn, 2003. Technical Report.
- [3] R. Dirckze. *JavaTM Metadata Interface (JMI) Specification, Version 1.0*. Unisys, June 2002.
- [4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- [5] A. Königs and A. Schürr. Tool integration with triple graph grammars - a survey. *Electronic Notes in Theoretical Computer Science*, 2005. Submitted for publication.
- [6] T. Maier and A. Zündorf. Yet Another Association Implementation. In A. Schürr and A. Zündorf, editors, *FUJABA Days 2004*, volume Technical Report tr-ri-04-253, pages 67–72. Universität Paderborn, 2004.
- [7] M. Matula. *NetBeans Metadata Repository*. SUN Microsystems, March 2003.
- [8] M. Münch, A. Schürr, and A. Winter. Integrity constraints in the multi-paradigm language progres. In *Proc. 6th International Workshop on Theory and Application of Graph Transformations TAGT, Paderborn*, volume 1764 of *Lecture Notes in Computer Science*, pages 338–351, Heidelberg, 2000. Springer Verlag.
- [9] Object Management Group. *Meta Object Facility (MOF) 2.0 Core Specification*, March 2003. ptc/03-10-04.
- [10] Object Management Group. *Unified Modeling Language: Infrastructure, Version 2.0*, September 2003. ptc/03-09-15.
- [11] Object Management Group. *XML Metadata Interchange (XMI) Specification, Version 2.0*, May 2003. formal/2003-05-03.
- [12] T. Röttschke. Adding Pluggable Meta Models to FUJABA. In A. Schürr and A. Zündorf, editors, *FUJABA Days 2004*, volume Technical Report tr-ri-04-253, pages 57–62. Universität Paderborn, 2004.
- [13] A. Zündorf. *Rigorous Object Oriented Software Development*. University of Paderborn, 2001. Habilitation Thesis.