Electronic Communications of the EASST
Volume 1 (2006)

Proceedings of the Third International Workshop on Graph Based Tools (GraBaTs 2006)

Creating Well-Structured Specifications in MOFLON

Carsten Amelunxen and Tobias Rötschke

12 pages

Guest Editors: Albert Zündorf, Daniel Varró
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Creating Well-Structured Specifications in MOFLON

Carsten Amelunxen and Tobias Rötschke

TU Darmstadt, FG Echtzeitsysteme
contact@moflon.org

Abstract: Considering the growing popularity of model-based development, specifications become more complex. As a consequence, graph-based modeling tools have to take measures to handle this complexity. In this paper, we present the metamodeling environment MOFLON which has been developed on top of the FUJABA Toolsuite during the last few years at our department. We focus one of MOFLON's strongest advantages, i.e. the realization of the abstraction and modularization features introduced by the recent UML 2.0 Infrastructure specification. The new concept of package merge allows to reuse and refine existing models without modifying the original. Subset and redefinition relationships become useful tools to refine associations due to the automatic propagation mechanism generated by the MOFLON code generator. We show how the user can organize large specifications using these concepts and how they effect graph transformation rules and code generation.

Keywords: Metamodeling, Graph Transformation, Package Refinement, Association Refinement

1 Introduction

The vision of Model Driven Architecture (MDA) paradigm means that the developers start with modeling the structure and behavior of the desired system on an abstract level using their favorite modeling tools. These abstract models will then successively be transformed into more specific models ideally resulting in the desired system. As a result, developers have to deal with lots of complex models that are distributed over various tools.

The metamodeling framework MOFLON[1] aims at bridging the gap between various tools by allowing to generate suitable analysis and integration tools helping to understand and interpret [Röt04] the available data and keeping distributed data consistent across tool boundaries [A+03]. To this end, it allows to create metamodels with class diagrams as well as related model transformations with graph transformations. Its code generator produces sophisticated Java code from these specifications. In [AKRS06], we have discussed how MOFLON can be used for model analysis, transformation, and integration with emphasis on the underlying triple graph grammar mechanism.

As human beings can only deal with a limited number of entities at the same time, individual diagrams should be rather small, however complex the specification. This paper supports a tool demonstration providing an example how MOFLON deals with complex specifications by realizing some of the new abstraction and modularization features introduced by the latest UML 2.0 specification [Obj06].

[1] http://gforge.echtzeitsysteme.org/projects/moflon/

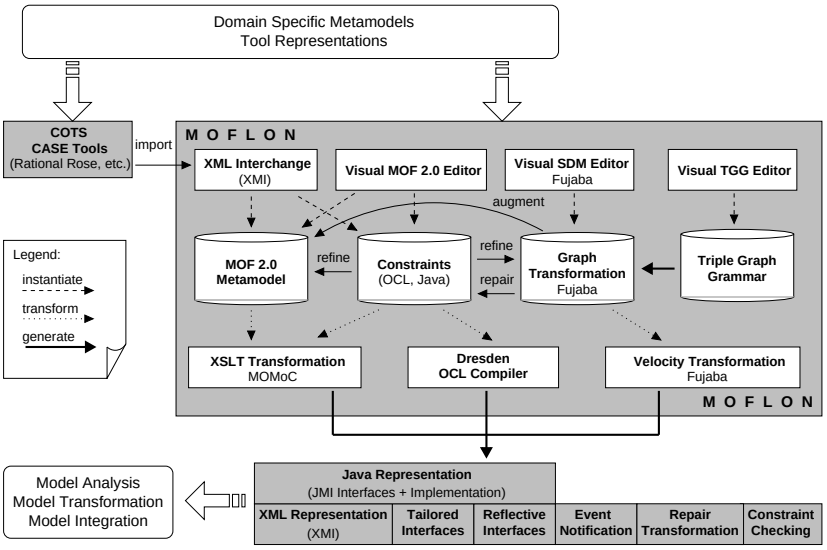

Figure 1: MOFLON architecture overview.

Section 2 provides an overview of the MOFLON architecture. The following three sections illustrate the application of MOFLON on the basis of an example. Section 3 deals with creating a schema, the influence of the new features to graph transformations are discussed in Section 4, and as a result, the code generation is considered in Section 5. We compare our solution with related ones in Section 6 and summarize the results in Section 7.

2 The MOFLON architecture

Although this paper only covers only one facet of MOFLON while neglecting many others, we provide a short overview of the whole concept in the following. When we started to design MOFLON, our goal was to reuse existing technology where possible and focus on conceptual improvements. To balance our desire to make use of the new features that come with UML 2.0 and the necessity to limit our effort, we decided to restrict the usage of UML 2.0 to the language MOF 2.0, which consists of UML 2.0 Infrastructure with only a small number of extensions. Back then, MOF 2.0-editors and code generators were not available, but there were several graph transformation tools. After comparing different approaches, we decided to realize MOFLON on top of the FUJABA Toolsuite [Zün01] which already featured graph transformations for UML-like graph schemata.

Fig. 1 provides an overview of MOFLON and FUJABA parts working together. Note that the large MOFLON block is divided into three layers: On top are various editor components to manipulate the specification. In the center, repositories symbolize related metamodels, constraints and transformation rules. The bottom layer consists of several code generators working together in MOFLON.

Domain-specific metamodels and tool representations can be created either using a commer-

cial *CASE tool* such as Rational Rose or directly using the new *MOF 2.0 Editor* plugin for FUJABA. The *metamodel* is kept in memory, as instance of a JMI-compliant Java representation of the MOF 2.0 metametamodel. Graph transformation rules are edited using the *SDM editor* that already exists in FUJABA. These rules augment the MOF 2.0 metamodel instance conceptually, by providing visually specified implementations of methods defined in the schema.

The *Triple Graph Grammar (TGG) editor* actually consists of a new schema editor and a rule editor adopted from FUJABA. Upon user request, ordinary SDM rules are generated from these TGG rules using a MOFLON-specific translation.

The metamodel can be refined using *OCL constraints*. They are used to define invariants and derived attributes as well as pre- and postconditions for methods implemented by graph transformations. Graph transformations are used to define *repair actions* for constraint violations.

From the MOF metamodel, JMI-compliant Java code [Dir02] is generated by *XSLT transformation* [ABS04]. Java code for graph transformations is generated using FUJABA's latest *Velocity-based code generator* [GSR05] and for OCL 2.0 constraints using the Dresden OCL compiler [LO04]. According to the JMI standard, the resulting Java representation features tailored and reflective interfaces, XMI import and export. Besides, the generated code features an event mechanism that makes our approach interface-compatible with MDR [Mat03].

At the time of writing, we consider the MOF 2.0 editor, adaption of the Fujaba graph transformation engine, JMI-compliant code generation for schema and graph transformations and import from Rational Rose and Magic Draw finished except for some bug fixing and performance optimization. Currently, our main effort is spent on the completion of the triple graph grammar and the OCL integration components, which already can be demonstrated on selected examples.

3 Creating the schema

In this section, we discuss the MOFLON schema editor based on an imaginary reverse engineering scenario. The general idea is to model the logic of a tool that can calculate the number of lines (LOC) in each file of a project, and lift that information to directory level. The specification should separate concerns and general parts of the specification should be reusable. In Fig. 2, we introduce relevant features step-by-step, rather than creating the schema in chronological order.

To start with, we model a file system, consisting of *Directories* containing *Files* (Fig. 2.1). Note, that we omit adornments like composition and navigability, as they are not of interest for this paper. For reverse engineering purpose, only directories related to a certain project, modeled as *ProjectDir* are of interest (Fig. 2.2). Let us assume, that these project directories contain only a certain type of file, e.g. *TextFiles*, consisting of an arbitrary number of *lines*. The more specific association between project directory and text file is represented by a new association. Apart from the additional possibilities of OCL, previous versions of UML did not provide any regular means to express the relationship between both associations. In UML 2.0, however we can *redefine* the association end *files* by the association end *textfiles*, thereby restricting the type of files contained in project files to *TextFile* (Fig. 2.3).

As directories may be nested, we further extend the schema as shown in Fig. 2.4. *Element* represents a common super class of *Directory* and *File*. A new association indicates that a *Directory* may contain any kind of *Elements*. Nesting of directories is explicitly expressed by another

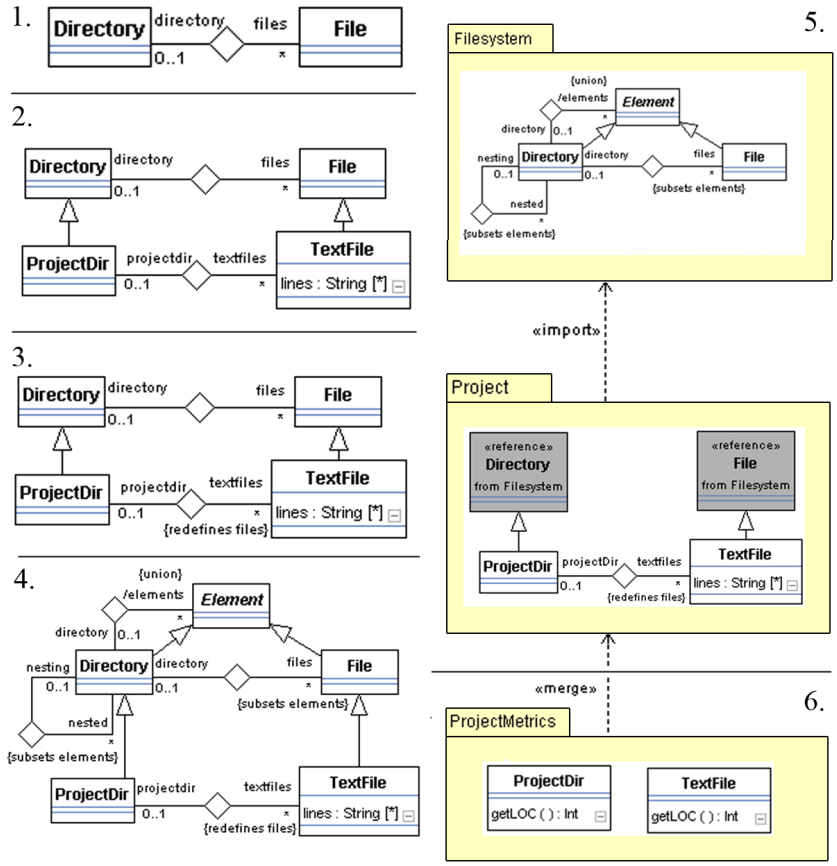

Figure 2: Stages of inventing the schema.

association. The original association between *Directory* and *TextFile* still exists. The association ends *files* and *nested* are marked as *subsets* of *elements*, thus expressing that each item in the set of *files* or *nested* directories is also contained in the set of *elements*. Because *elements* is marked as *union*, items are not stored redundantly, i.e. the set of *elements* is directly composed of *files* and *nested*. As a result, we can query the relationship between directories and its elements in several ways.

When reconsidering our example so far, one might draw the conclusion, that we have blended two different concerns, i.e. general file systems and projects-specific file system information for reverse engineering purposes. Therefore, we divide the model over the packages *Filesystem* and *Project* (Fig. 2.5). As the more specific part uses the general part, we create a package import relationship between the two packages. The package *Project* refers to the types *Directory* and *File* of package *Filesystem* to define generalization relationships. The types are visible through the package import and represented as darkened boxes with stereotype ”reference”.

Finally, we want to extend the package *Project* by adding methods for the calculation of the metric *Lines of Code* (LOC). However, we do not want to touch the package *Project*, so we

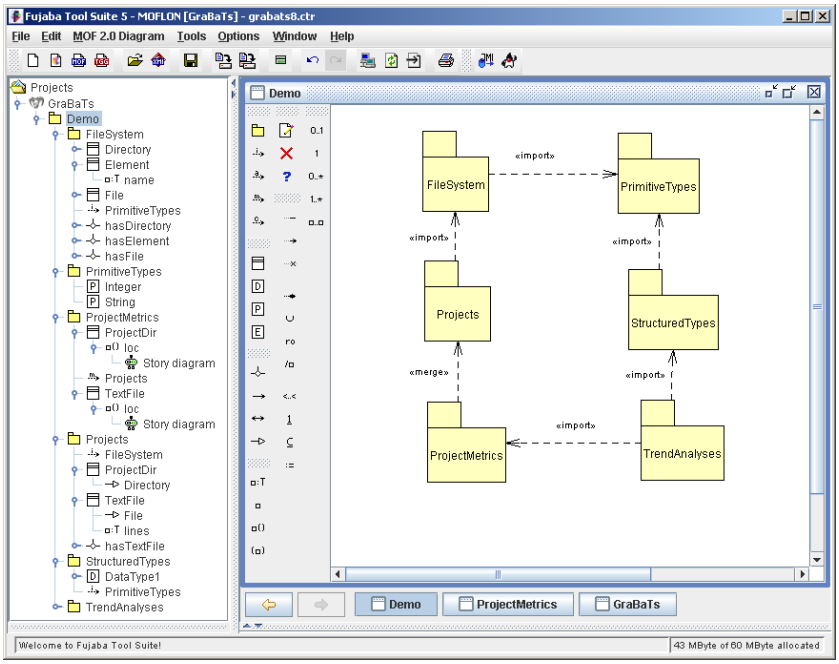

Figure 3: Package structure of complete specification

can reuse it for other purposes, where metrics might not be required. Using an import relation, we would have to define new classes similar to step 5, i.e. with different names and explicitly generalize *ProjectDir* and *TextFile*. Using a package merge such as in Fig. 2.6 provides are more elegant way to do so. Class with identical names than classes in the merged package automatically specialize them. Besides, the complete context is redefined as well. For instance, there implicitly is a new association between *ProjectDir* and *TextFile* whose association ends redefine the corresponding ends in the merged package. As a result, links between *ProjectDir* and *TextFile* can only be created, if the associated objects have types defined in the same page, i.e. either *Project* or *ProjectMetrics*.

Fig. 3 shows how the MOFLON tool looks like, after creating the schema as discussed so far. On the left-hand side, a browser displays the hierarchy of the specification. On the right-hand side, the selected diagram is shown, which represents the contents of a top-level package named *Demo*. We added an extra package *PrimitiveTypes* defining the types *String* and *Integer* used by our specification. To explain the effect of namespaces in MOFLON, we also added the packages *StructuredTypes* and *TrendAnalyses* not been used so far.

For instance, the package *StructuredTypes* contains a data type *Date*. This type does not appear in the list of available return types for the method *TextFile.loc* (Fig. 4). As one can see from the package structure in Fig. 3, the package *StructuredTypes* is neither imported nor merged by *ProjectMetrics*, hence *Date* is invisible. Opposed to that, *Integer* and *String* defined in *PrimitiveTypes* are visible, as package merge and import are transitive. *Void* indicates the absence of a return type.

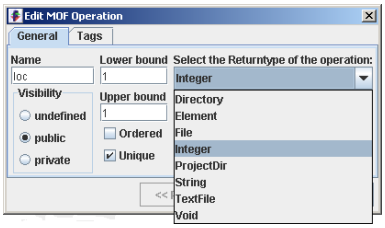

Figure 4: Available return types for `TextFile.loc()`.

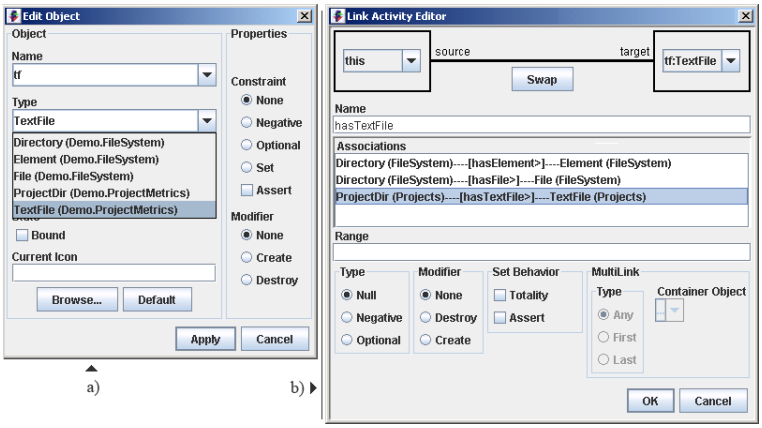

Figure 5: Available node and edge types.

4 Defining graph transformations

The visibility rules implied by package import and package merge affect the definition of graph transformations, too. Fig. 6 shows such a transformation, calculating the *ProjectDir.loc()* as sum of the result of *TextFile.loc()* of all contained text files. As mentioned in Section 2, we reuse the graph transformation language of Fujaba (SDM) for this purpose, where transformation rules are realized as implementations for methods defined in the schema. A rule consists of a UML-like activity diagram for defining the control flow, where activities either contain Java code or story patterns. Story patterns are special object diagrams describing patterns and transformations in the dynamic object structure of the model.

Fig. 5.a shows the available types for objects used in story patterns for methods in the package *ProjectMetrics*. As only classes can be used as object type, the five displayed classes remain. Note, that *Project.ProjectDir* and *Project.TextFile* are not available, as their simple names are hidden by the names in *ProjectMetrics*. Although MOFLON can deal with fully qualified names as required by the MOF 2.0 standard (e.g. for models imported through XMI), the user interfaces forces the user to use import and merge relationships instead to avoid ill-structured specifications.

Fig. 5.b shows the available edge types for a link between a *ProjectDir* and a *TextFile*. As discussed in Section 3, there is an implicit association defined in *ProjectMetrics* which is not

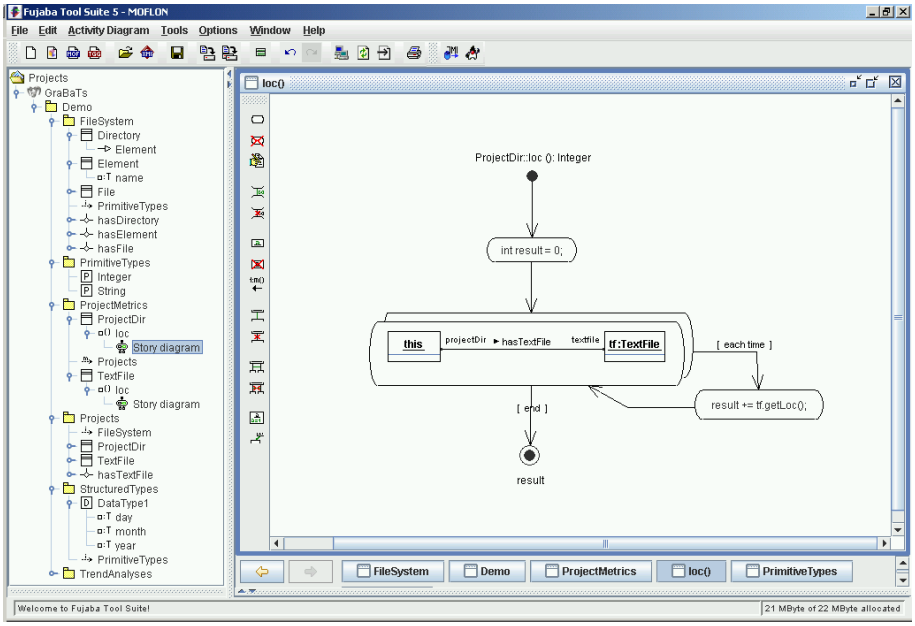

Figure 6: A Story Diagram for the MOF 2.0 schema.

displayed. Instead, there is the association *hasTextFile*, which is actually hidden by the implicit one. Our code generator deals with this correctly, but we decided to display the explicit association rather than the implicit one, because that is the one defined by the user. Obviously, we have to gather some practical experience with real applications to be able to judge whether this decision works out fine.

The association refinement features used in our specification have some influences on the specification of transformations as well: Similar to abstract classes, one cannot instantiate associations whose ends have been marked as *union*. It is also forbidden to instantiate associations between classes if the ends have been redefined. For instance, the transformation rule in Fig. 6 must not create links of type *hasFile*, as its ends have been redefined by *hasTextFile*. Nevertheless, it would be possible to create a link of type *hasElement*. Note that such a link would not be in the subset *file*.

5 Generating code

MOFLON generates Java code according to the JMI standard [Dir02] that originally has been designed for MOF 1.4. JMI only defines interfaces, which are also applicable for MOF 2.0 metamodels. Yet MOFLON does not only generate interfaces but also their implementation, providing access to the user-defined metamodel using tailored interfaces and reflective interfaces, persistency through XMI import and export, event notifications for user interfaces and additional persistency services, constraint checking and repair transformations, and finally user-defined model transformations specified through graph transformations (cf. Section 2). In this section,

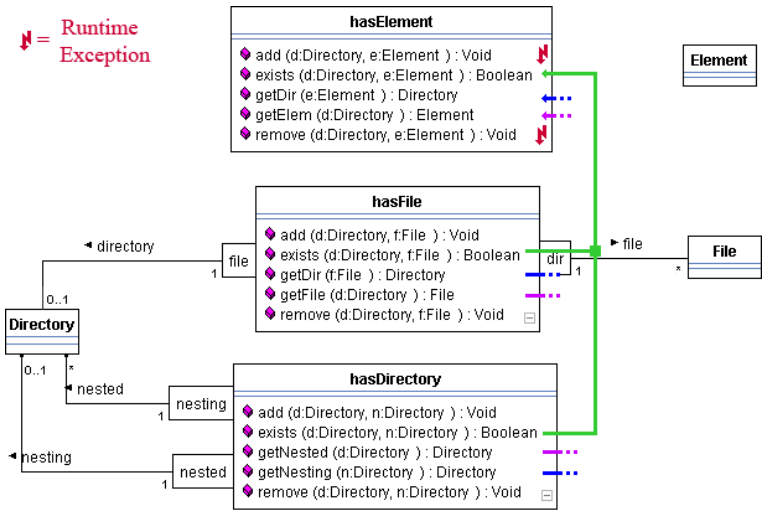

Figure 7: Sketch of generated association code.

we describe the effect of the features used in Sections 3 and 4 on the generated code.

5.1 Subsetting and redefinition

Fig. 7 visualizes the essential classes that are generated for the associations defined in Section 3, neglecting the specifics of the JMI interface. For each association in the schema, a corresponding class is generated. These classes are actually singletons. The classes provide a number of methods to realize the features presented in Section 2. Fig. 7 only shows the tailored interface.

Links of a certain type are kept in hash maps that are visualized as qualified associations in Fig. 7. Each link exists as pair of corresponding map entries. The association between *hasFile* and *File* represents a hash map that provides all *files* connected to a given *directory*. The association on the other side represents the corresponding hash map providing all *directories* a given *file* is contained in (hopefully only one). Although this approach consumes some extra memory and time when creating links, querying the links is fast.

The class *hasElement* represents the Association with ends marked as *union*. Therefore, it contains no hash maps to store links. Queries through *exists*, *getElement*, or *getDirectory* are delegated to the other classes depending on the actual type of the provided *element*. As a result, there is a data flow from the other association instances, as indicated by the thick arrows. Changes on associations with *union* ends are not allowed. Thus, method calls to *add* or *remove* lead to runtime exceptions. As mentioned in section 4, the rule editor does not allow story patterns that would create or delete such links.

Due to the *redefines* relationship between *textfile* and *file* in Fig. 2, there are also runtime exceptions, if the *add* method of *hasFile* is invoked with a parameter that actually is a *ProjectDir* or a *TextFile*. These links must be create using *hasTextFile.add()*, which is not depicted in Fig. 7. Note, that this violation can not be avoided by the rule editor, as the actual type of parameters

is only known at runtime.

5.2 Package import and merge

Similar to the code generation for the new association concepts, the generation of code for packages is partly given by JMI. One package is mapped on a package in Java with an additional interface. The interfaces generated for packages contain methods to query instances of inner packages as well as methods to query container classes for all included classes and associations. See [Dir02] for further details. The interfaces are not affected by any imports, regardless of the kind of import. Since all type references in the generated code use fully qualified names, there is no equivalent for imports in the generated code. In general it would be possible to map a package import onto a wildcard import in Java and a single element import onto a single import in Java. Such a mapping might increase the readability of the generated code which is a negligible feature since the generated code should just be used through the interfaces and not manipulated manually. In fact, such an import mapping would result in name clashes. Therefore, the code generated by MOFLON just uses fully qualified type references.

From the perspective of code generation, the package merge is rather a transformation instruction than a metamodel element. Thus, there is also no direct equivalent of a package merge in the generated code. There are detailed instructions how to transform a metamodel containing a package merge into a metamodel without package merges. Before the process of code generation, the package merge is transformed via a batch-process.

The current standard of MOF 2.0 describes the package merge as a process of merging equally named elements and deep-copying elements form the merged package into the merging package. Such a transformation based on deep copying has the effect that elements in the merging and the merged package are not type compatible. Considering the example in Fig. 2, with transformation semantics as described above, an additional package which uses package *Project* would not be able to use package *ProjectMetrics* instead because there would be no type compatibility between *Project::Textfile* and *ProjectMetrics::Textfile* after unfolding the package merge.

Older versions of MOF 2.0 [Obj03b, Obj03a] supported two different kinds of package refinement, one with deep copy semantics and one with specialization through inheritance semantics. The latter effects that in cases of equally named elements in the merged and merging packages the element in the merging element inherits from the element in merged package. Such semantics ensure type compatibility and therefore MOFLON supports this kind of merge semantics.

6 Related Work

PROGRES [SWZ99] features modularization concepts to support large specifications based on the UML 1.4 package concept [Win00]. PROGRES provides two kinds of package dependencies: import and specialization, which are combined with visibilities public, protected and private. Imports allow to use elements from the imported package, whereas specialization of packages also allows to refine elements of the specialized package. The visibility of package dependencies allows to reduce the visibility of referred elements in the referring package. However, apart from the technical aspects, it has always remained unclear, how the various visibility

combinations could be applied in a useful way to solve practical problems. In our opinion, the package merge mechanism implemented by MOFLON involving less visibility calculation is a much clearer concept next to the well-known package import, although MOFLON still is to young to be able to validate this statement by case studies.

PROGRES does not provide the whole range of association refinement possibilities that MOFLON provides and especially lacks the propagation mechanism mentioned in Section 5. However, type restrictions for association ends, such as implied by association redefinition, can be achieved using metaattributes.

VIATRA [BV06] uses a hierarchical graph model, where each node defines a namespace. Foreign nodes may be referenced through explicit imports or fully qualified names. Large specifications are further supported by a zoom mechanism that visualizes nested elements up to a user-defined depth from the currently active element, hence reducing visual complexity. VIATRA also supports association refinement.

FUJABA [Zün01] offers a rather limited mechanism to reduce the complexity of large specifications: The main structuring mechanism consists of diagrams, which are categorized into their different kinds (class-, activity-, package diagrams, etc.). All diagrams contribute to a global namespace, i.e. simple names must be unique, although it is possible to circumvent this restriction. Classes may be assigned to packages, although this information is used for code generation purposes only, rather than controlling the behavior of the tool. Although references across diagram borders are possible, they are not always visualized, and require the user to obtain this information from dialogs. To further reduce the complexity when working with large diagrams, a view mechanism has been realized [Rec01], that has is able to show portions of a given host diagram. Views are configured by filter rules that allow to specify the visualized context of preselected elements. Although these views help to isolate different topics to better understand the specification, the host diagrams still exist and may become very complex. FUJABA supports no association refinement. All other graph transformation systems we are aware of, provide even less support for large specifications. GReAT [ALS+02] at least allows to structure specifications in hierarchies without providing namespaces, which effectively holds for MOTMOT [SVGJ04] as well, because its specifications are created with third-party UML-Tools. Other tools like AToM^3 [DVM04], GUPRO [EKRW02], and AGG [EEPT06] have no modularization support at all.

7 Conclusions

MOFLON is a tool to generate Java implementations for model repositories that support MDR all major features of MDR including reflexivity and event mechanism. Beyond that, it supports the new concepts recently introduced in MOF 2.0, generates Java code for transformations of one and between multiple models, and finally aims to produce Java code for model analyses generated from OCL constraints. In this paper, we have described a tool demonstration that showing how some of the new UML 2.0/MOF 2.0 abstraction and modularization features have been implemented in the metamodeling framework MOFLON. Association and package refinement help to create better structured, more compact specifications than before.

From our own experience from ongoing industrial case studies an by using the mechanisms

described in this paper to specify the underlying model of the MOFLON editor, we are convinced that the new features are beneficial when creating complex specifications. Ultimately, the entire OMG standardization approach around UML, MOF, OCL, QVT, and CWM depends on these features to manage the complexity of the specifications. Now MOFLON has reached a level of maturity where it can be released to the public, we want to stimulate other researches to try out our solution.

Together with the rest of the Fujaba community, we will work on the multi-project support of the Fujaba Toolsuite, which provides a convincing scenario for the package merge, as packages from foreign projects sometimes need to be extended, but usually may not be altered. Besides, we are looking forward to the upcoming release of the Dresden OCL Compiler for OCL 2.0, which will enable us to finalize our ongoing integration effort. Based on a new metametamodel integrating MOF 2.0, SDM, OCL, and TGG, we will intend to make MOFLON a very useful tool to rapidly develop model analysis, transformation and integration applications.

Bibliography

[A+03] Frank Altheide et al. An Architecture for a Sustainable Tool Integration. In Dörr and Schürr, editors, *TIS 2003 Workshop on Tool Integration in System Development*, pages 29–32, 2003.

[ABS04] Carsten Amelunxen, Lutz Bichler, and Andy Schürr. Codegenerierung für Assoziationen in MOF 2.0. In *Proc. Modellierung 2004*, pages 149–168, 2004. In German.

[AKRS06] Carsten Amelunxen, Alexander Königs, Tobias Rötschke, and Andy Schürr. MOFLON: A Standard-Compliant Metamodeling Framework with Graph Transformations. In *Proc. ECMDA-FA*, volume 4066 of *LNCS*, pages 361–375. Springer, 2006.

[ALS+02] Aditya Agrawal, Tihamer Levendovszky, Jonathan Sprinkle, Feng Shi, and Gabor Karsai. Generative Programming via Graph Transformations in the Model Driven Architecture. In *Proc. Workshop on Generative Techniques in the Context of MDA*, 2002.

[BV06] András Balogh and Dániel Varró. Advanced Model Transformation Language Constructs in the VIATRA2 Framework. In *ACM Symposium on Applied Computing*, pages 1280–1287. ACM Press, 2006.

[Dir02] Ravi Dirckze. *Java™ Metadata Interface (JMI) Specification, Version 1.0*. Unisys Corporation, Sun Microsystems, Inc., June 2002.

[DVM04] Juan De Lara Jaramillo, Hans Vangheluwe, and Manuel Alfonseca Moreno. Metamodelling and Graph Grammars for Multi-Paradigm Modelling in AToM^3. *Software & Systems Modeling*, 3(3):194–209, August 2004.

[EEPT06] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. *Fundamentals of Algebraic Graph Transformation*. EATCS. Springer, 2006.

[EKRW02] Jürgen Ebert, Bernt Kullbach, Volker Riediger, and Andreas Winter. GUPRO - Generic Understanding of Programs – An Overview. *ENTCS*, 72(2):59–68, 2002.

[GSR05] Leif Geiger, Christian Schneider, and Carsten Reckord. Template- and Modelbased Code Generation for MDA-Tools. In *Fujaba Days 2005*, Paderborn, Germany, 2005.

[LO04] Sten Löcher and Stefan Ocke. A Metamodel-Based OCL-Compiler for UML and MOF. In Peter H. Schmitt, editor, *Workshop Proc. OCL 2.0 - Industry standard or scientific playground?*, volume 102 of *ENTCS*, pages 43–61. Elsevier, 2004.

[Mat03] Martin Matula. *NetBeans Metadata Repository*. SUN Microsystems, March 2003.

[Obj03a] Object Management Group. *Meta Object Facility (MOF) 2.0 Core Specification*, March 2003. ptc/03-10-04.

[Obj03b] Object Management Group. *UML 2.0 Infrastructure Specification*, 2003. ptc/03-09-15.

[Obj06] Object Management Group. *UML 2.0 Infrastructure Specification*, 2006. formal/05-07-05.

[Rec01] Carsten Reckord. Entwurf eines generischen Sichtenkonzeptes für die Entwicklungsumgebung Fujaba. University of Paderborn, 2001. Bachelor thesis, in German.

[Röt04] Tobias Rötschke. Re-engineering a Medical Imaging System Using Graph Transformations. In John. L. Pfaltz, Manfred Nagl, and Boris Böhlen, editors, *AGTIVE 2003*, volume 3062 of *LNCS*, pages 185–201. Springer, 2004.

[SVGJ04] Hans Schippers, Pieter Van Gorp, and Dirk Janssens. Levering UML Profiles to Generate Plugins from Visual Model Transformations. In *Proc. Software Evolution through Transformations*, pages 7–17, 2004.

[SWZ99] A. Schürr, A. Winter, and A. Zündorf. *PROGRES: Language and Environment*, volume 2, pages 487–550. World Scientific, 1999.

[Win00] Andreas J. Winter. *Visuelles Programmieren mit Graphtransformationen*. PhD thesis, RWTH Aachen, 2000. In German.

[Zün01] Albert Zündorf. *Rigorous Object Oriented Software Development*. University of Paderborn, 2001. Habilitation Thesis.