

MOFLON: A Standard-Compliant Metamodeling Framework with Graph Transformations*

C. Amelunxen, A. Königs, T. Röttschke, A. Schürr

Real-Time Systems Lab
Darmstadt University of Technology
{*amelunx|koenigs|rotschke|schuerr*}@es.tu-darmstadt.de

Abstract. The crucial point in Model Driven Architecture (MDA¹) is that software and system development are based on abstract models that are successively transformed into more specific models, ideally resulting in the desired system. To this end, developers must be enabled to model different aspects like structure, behavior, consistency constraints of the system. This results in a variety of related models, which in turn need tool support on the metalevel. However, there is a lack of tools offering uniform support for metamodel definition, analysis, transformation, and integration. In this paper we present the metamodeling framework MOFLON that addresses these issues by bringing together the latest OMG standards with graph transformations and their formal semantics. MOFLON provides a combination of visual and textual notations and offers powerful modularization concepts. Using MOFLON, developers can generate code for specific tools needed to perform the desired modeling tasks.

1 Introduction

Implementing the Model Driven Architecture (MDA) paradigm in software and system development 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. Thus, developers have to deal with lots of different models, and need domain- and project-specific tool support in defining, analyzing, transforming, and integrating them.

Metatools provide assistance with the realization of the required tool support. Current solutions either are limited to a subset of model-related tasks, lack a proper formal foundation, are not compliant to common standards, or make use of not standard compliant or unfortunate notations. Therefore, they do not

* Work supported in part by the European Community's Human Potential Programme under contract HPRN-CT-2002-00275, SegraVis

¹ "Model Driven Architecture" and "MDA" are registered trademarks of the Object Management Group (OMG)

provide full support for the realization of MDA-enabling tools, make it hard to validate such tools, cannot easily be integrated with related approaches, or are difficult to use and understand.

In this paper we present our metamodeling framework MOFLON² that addresses these deficiencies. MOFLON aims at compliance to the latest OMG standards, and integrates them with the well-known formalism of graph transformations.

Section 2 presents some industrial case studies that motivate our efforts. In Section 3 we provide a running example that will be used in Section 4 in order to illustrate the application of MOFLON. In Section 5, we present the architecture of our metamodeling framework. We compare our solution with related ones in Section 6. In Section 7, we summarize our results and discuss further steps.

2 Case Studies

To begin with, we use MOFLON in its own development process. Initially, we created a simplified MOF 2.0 [1] metamodel in Rational Rose, exported it as XMI and generated the resulting Java representation using an existing code generator [2]. This Java code forms the core of the schema editor and the code generator. Meanwhile, we are able to generate this core from a metamodel, which consists of the complete UML 2.0 Infrastructure Library and MOF 2.0 specification with a small number of deviations. During this bootstrapping process, we gained a very deep insight in the advantages and shortcomings of the latest OMG specifications and were able to improve the MOFLON machinery.

Several industrial partners from various domains (medical imaging, multi purpose industrial printers and enterprise storage solutions) have faced the task of restructuring their software architectures to deal with increased complexity and adopt new technologies. As business needs demand continuous output of new releases, all projects require an evolutionary analysis approach to monitor restructuring progress during the ongoing development of "normal" features. MOFLON is used to support these efforts by generating individual reverse engineering and architecture monitoring tools from project-specific metamodels, metrics, and consistency rules [3].

Our industrial partner from the automotive sector develops integrated systems such as adaptive cruise control or windscreen wipers with rain sensors. The development processes involve quite a number of different tools each specialized in certain tasks (e.g. requirements engineering, modeling of software and hardware functionality, test case maintenance). Thus, the data of a project as a whole is distributed over different tools. Typically, these tools are commercial-of-the-shelf (COTS) that are rarely designed to integrate with each other. Nevertheless, the data stored in the separate tools is related. Our running example (cf. Section 3) demonstrates how these relationships can be detected automatically and utilized for integration purposes. Our integration approach is part of the tool integration framework Toolnet [4].

² <http://gforge.echtzeitsysteme.org/projects/moflon/>

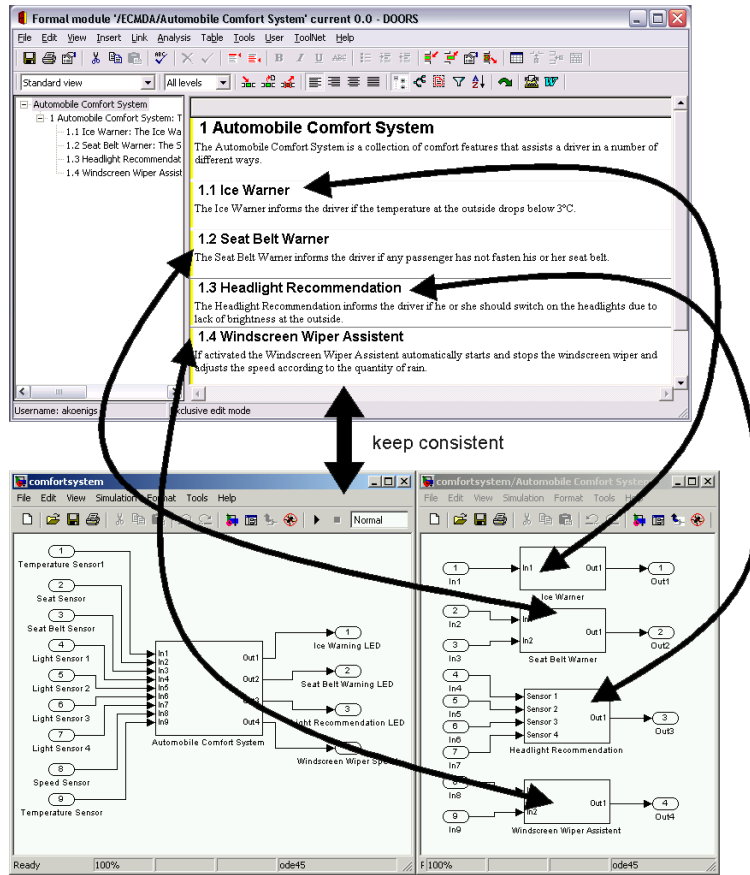


Fig. 1. Consistency between DOORS and Matlab/Simulink

From industrial case studies we have learned, that we need metamodel-based support for repositories, standard-compliant tool interfaces, constraint- and design rule-checking, as well as intra- and inter-model transformations. In our case however, it turned out that unlike classical meta-CASE tools, that we need no support for generating diagram editors.

3 Example: Automobile Comfort System

In this section, we introduce an illustrative toy example that describes a very small system development project motivated by a real world scenario from the automotive domain. We present tasks that a developer is confronted with, and demonstrate what kind of support he needs. The project deals with the development of an integrated automobile comfort system. The comfort system is a collection of several subsystems that assists the driver in a number of different

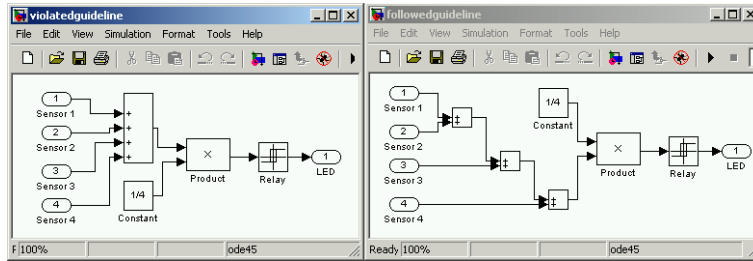


Fig. 2. Example of the application of design guidelines in Matlab/Simulink

ways. In the following we primarily turn our attention to the *Headlight Recommendation Subsystem (HRS)*. According to these requirements the developer chooses to design the system using the system modeling tool Matlab/Simulink (cf. Fig. 1 at the bottom). The left side depicts the *Automobile Comfort System* that is connected to a number of sensors. These sensors internally are connected to the corresponding subsystems as shown on the right side.

The developer is provided with a requirements document stored in the requirements management tool DOORS as shown in Fig. 1 at the top. The requirements state that the Automobile Comfort System consists of four subsystems including the HRS. He models the HRS as depicted in Fig. 2 at the left-hand side, providing the HRS with a number of light sensors. It calculates the average of all measured values by using a `Sum` and a `Product` block. If the `Relay` block calculates that the average is below a certain threshold, the HRS informs the driver by activating the `LED`.

As the developer is designing the system another developer is still working on the requirements refining, changing, and maintaining them. The system designer has to merge these changes into his system in order to keep it consistent with the requirements document. In particular, the requirements correspond to the subsystems of the Automobile Comfort System. Finally, the system designer has to ensure that he has implemented all requirements. Besides the requirements document the system designer is provided with modeling guidelines to which he must adhere to. Among others, the guidelines demand that the designer should not use `Sum` blocks with more than two `Input Ports`³. As a matter of fact this is not enforced by Matlab/Simulink itself. Nevertheless, the guidelines force the developer to change his model into another model that might look like the one depicted in Fig. 2 on the right-hand side. The `Sum` block has been replaced by an equivalent cascade of `Sum` blocks that only have two `Input Ports` each. Without any tool support the system designer has to fulfill the tasks of consistency preserving, traceability maintenance, and guideline constraint checking by hand. Although this is not impossible for rather small projects it is very time consuming and error prone in medium or large scale projects. The situation becomes even worse if the number of developers, involved tools, and documents increase.

³ A code generator produces erroneous code for blocks with more than two inputs.

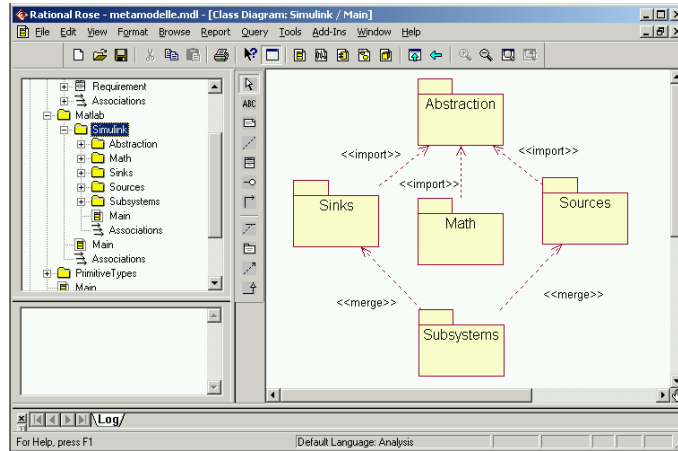


Fig. 3. Metamodel of Matlab/Simulink in Rational Rose

In the following we present how developers can specify their tasks on an abstract level. From these specifications we can then generate the needed tool support.

4 Generating Tool Support with MOFLON

In the following sections we describe how tool support for the outlined scenario can be generated with MOFLON. The scenario as described before is clearly a matter of metamodeling. For the application of transformations of Matlab/Simulink models as well as for the application of the integration between DOORS requirements and Matlab/Simulink models, the metamodels of both tools have to be defined. Based on these metamodels, the transformations, analysis, and integration operations can be specified.

4.1 Specifying Metamodels

MOFLON uses MOF 2.0, the latest OMG metamodeling-standard, as metamodel. MOF 2.0 is characterized by powerful concepts for modularization, abstraction, and refinement of large metamodels. In the following the new concepts of MOF 2.0 are demonstrated on the metamodel of Matlab/Simulink which is introduced in detail. Fig. 3 depicts a screenshot of Rational Rose that shows a very simplified part of the Matlab/Simulink metamodel. The metamodel could have been modeled in MOFLON directly as well. But since COTS modeling tools are very popular among our industrial partners, MOFLON offers an XMI import to allow the users to retain their established modeling customs. The imported metamodel is shown as screenshot of MOFLON in Fig. 4. It shows the packages Abstraction, Sources, and Subsystems from Fig. 3 in detail. The metamodel

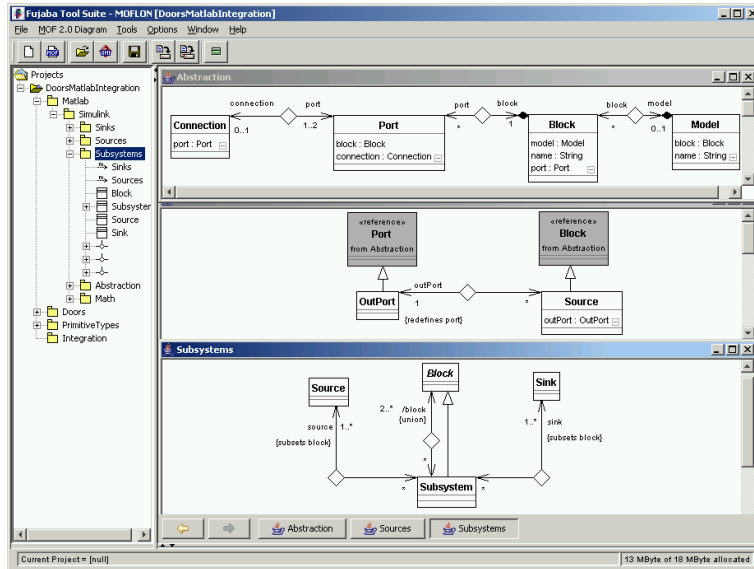


Fig. 4. Imported Metamodel of Matlab/Simulink in MOFLON

is decomposed in several packages. The package **Abstraction** describes the general dependencies between the most general elements **Model**, **Block**, **Port**, and **Connection**.

Those general concepts are reused and refined in the packages **Sources** and **Sinks** by importing **Abstraction**. In package **Sources** for instance, the relationship between the special block **Source** and the special port **outPort** is redefined in such a way that a **Source** always has exactly one **outPort**. The concepts, namely the classes, of the **Sinks** and **Sources** are extended in **Subsystems** by the application of package merges, the metamodel refinement concept of MOF 2.0. In **Subsystems** the class **Source** is extended by a navigable association end because each subsystem has at least one source. Due to the package merge to **Sources** the attributes (and association ends) of the classes **Source** in the packages **Sources** and **Subsystems** are merged in the class **Source** in package **Subsystems**.

Beside the redefinition of association ends in MOF 2.0, another very helpful feature is the subsetting of association ends as demonstrated by the association ends **block** and **source**. A subsystem contains an arbitrary number of blocks but at least one sink and one source. The mandatory sinks and sources are also blocks and should, therefore, be available through the association end **block**. Thus **source** is declared as subset of **block** which means that all instances of **Source** that are linked with an instance of **Subsystem** are available via **source** as well as via **block**. The declaration of **block** as union of its subsets causes the collection of **Block**-instances which is represented by **block** to be composed from all subsets of **block**.

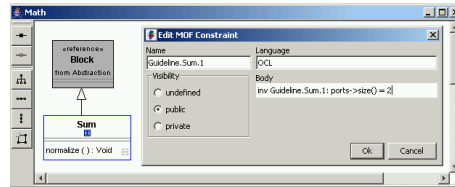


Fig. 5. Design guideline as OCL 2.0 constraint in MOFLON

From such a description of the abstract semantics of Matlab/Simulink it is possible to generate Java code with JMI-compliant (Java Metadata Interface [5]) interfaces. The code can be used to create, query, destroy, and serialize instances of the specified metamodel according to the static semantics specified in MOF 2.0. It can easily be integrated based upon an event mechanism which is compatible with the Netbeans Metadata Repository (MDR) [6]. Regarding the outlined example, the metamodels of DOORS and Matlab/Simulink act as basis on which design guidelines and transformation can be defined. They also provide the basis for the integration of DOORS and Matlab/Simulink models as instances of the generated metamodel representations.

4.2 Adding Constraints

With MOF 2.0 as metamodeling language, we are able to specify the abstract syntax of a modeling language, but to specify its static semantics, we need constraints as discussed in this section. For instance, it is not possible to express that a block's name has to be unique within a subsystem. In such cases, additional constraints are needed. The Object Constraint Language (OCL) [7] as the textual constraint language within the OMG standardization scenario is the appropriate choice to fill this gap. On the one hand OCL can be used to express the static semantics of a metamodel more precisely, on the other hand it is also possible to specify additional information about the usage and application of the metamodel, like for instance design guidelines. Concerning the outlined example, the design guideline stating that a **Sum**-block should not have more than two input ports can be expressed in OCL. Fig. 5 shows the appropriate constraint. OCL constraints can only be evaluated on instances of the associated metamodel. Thus, the evaluation of OCL constraints is a matter of the generated code. MOFLON integrates the Dresden OCL compiler framework [8] consisting of a parser and a code generator which generates code for the evaluation of invariants in a first version. In the long run, support for body constraints as well as pre- and postconditions is also planned. The generated code checks the compliance of the metamodel instances to the specified invariants.

Currently, we are working on the implementation of several (incremental) constraint evaluation strategies as well as on the integration of constraint checking with the execution of appropriate repair actions. A repair action in form of a transformation has to be triggered if a constraint is broken.

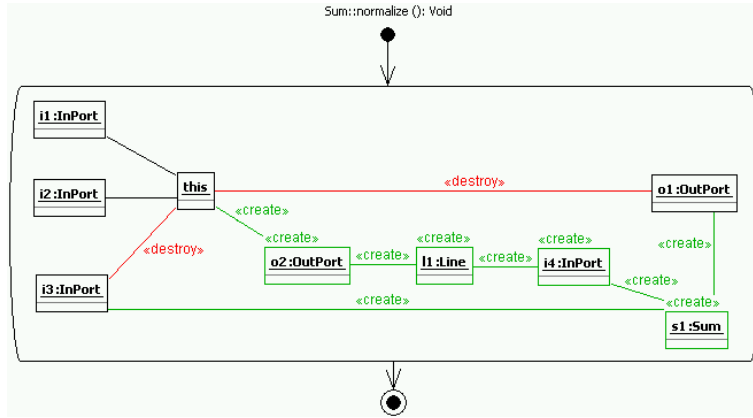


Fig. 6. Repair action as example of a graph transformation rule

4.3 Defining Model Transformation

Until now, we are only able to describe the syntax and static semantics of a modeling language. Still missing are means to specify queries and transformations on models. We also need to define actions that can be invoked if constraint violations are detected. Object-oriented graph transformation languages provide an intuitive means to implement such queries and transformations. When we designed MOFLON, we decided to adopt the graph transformation engine of the Fujaba Toolsuite [9], as it uses UML-like graph schemata and generates Java code from specifications. The transformation language is called Story Driven Modeling (SDM). In [10], we discussed the necessary modifications to adopt the SDM machinery for MOF 2.0 and JMI code generation.

The graph transformation `Sum::normalize()` in Fig. 6 can be used as repair action, until the constraint discussed in section 4.2 is no longer violated. The rule reads as follows: The given `Sum` block, that we try to normalize, is called `this`. If it is associated with at least three different `Inputs` and one `Output`, the rule matches. The links to the third matched `Input` `i3` and the `Output` `o1` are removed. As a replacement, `this` is connected to a new `Output` `o2`. `Input` `i3` is connected to a new `Sum` block `s1`. `Output` `o2` is connected to a new `Input` `i4` through a new `Line` `l1`. `Input` `i4` is then connected to `s1`, which finally is connected to the original `Output` `o1` to preserve the context. Currently, Fujaba is being equipped with a new template-based code generator [11] that allows developers to generate code for various target languages and interfaces. While the core developers are creating the templates for the proprietary Fujaba interface, we are busy with completing alternative templates to generate JMI-compliant code for transformations that works directly with the Java representation generated by the MOFLON code generator for the schema part. As a result, the code generated by MOFLON includes executable transformations that can be executed manually or triggered as repair actions of violated constraints. Consid-

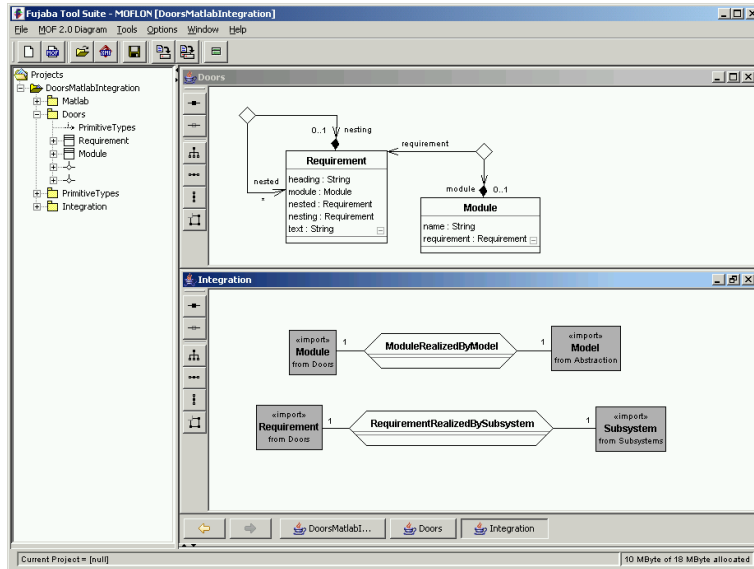


Fig. 7. Schema for the specification of TGG rules

ering the outlined example, the surveillance of design guidelines can already be realized until this point.

4.4 Specifying Model Integration

Finally, the model of the Automobile Comfort System should be kept consistent with the requirements document and vice versa⁴. To this end the developer must identify which model elements in Matlab/Simulink correspond to which requirements in DOORS. Since the correspondences should be used for traceability purposes as well as for consistency checking, consistency recovery, and change propagations it is preferable to have a declarative model integration approach. Using a declarative approach means that the developer has to specify only a single rule from which all needed operational rules for the desired integration tasks can be derived automatically. As we are using SDM which is based on graph transformations in Section 4.3 on the one hand and we want to provide a declarative approach on the other hand it is reasonable to rely on *triple graph grammars* [12] for that purpose. For a more detailed discussion on this decision the reader is referred to [13].

Triple Graph Grammars are based on a schema as common graph grammars do [14]. In our approach the schema is a metamodel that declares types for correspondences and links classes from the corresponding tools' metamodels. Fig. 7 shows the schema specified with MOFLON we use in our toy ex-

⁴ For a more realistic integration scenario the reader is referred to [12].

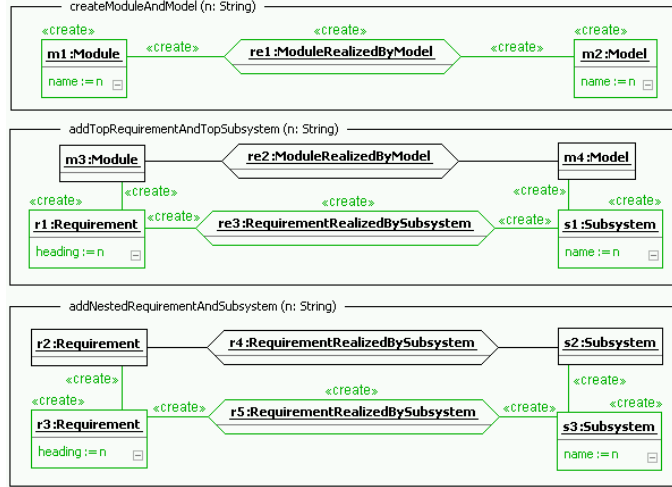


Fig. 8. Example of a declarative TGG rule

ample. The schema declares a correspondence type `ModuleRealizedByModel`. This type expresses that each `Module` from DOORS corresponds to one `Model` from Matlab/Simulink and vice versa. Accordingly, the schema declares a type `RequirementRealizedBySubsystem`. Besides the schema a Triple Graph Grammar provides a set of declarative rules. Generally speaking, the rules describe the simultaneous evolution of the tools' models and the correspondence model. Rule 8 at the top states that a DOORS `Module` `m1` is created simultaneously with a Matlab/Simulink `Model` `m2` which correspond to each other. The `name` attributes of `m1` and `m2` respectively are set to the value of the rule's parameter `n`. Rule 8 in the middle describes the simultaneous addition of a `Requirement` `r1` to a `Module` `m3` and a `Subsystem` `s1` to a `Model` `m4` that corresponds to `m3`. Again, the rule sets the `heading` of `r1` as well as the `name` of `s1` to the value of the parameter `n`. Accordingly, rule 8 at the bottom specifies the simultaneous addition of a (Sub-)Requirement `r3` to a `Requirement` `r2` and a `Subsystem` `s3` to a `Subsystem` `s2` that corresponds to `r2`. From these declarative rules we can automatically derive a number of operational graph rewriting rules. Fig. 9 gives an example of such an rule. This rule is used to transform the given `Requirement` `r` into a corresponding `Subsystem` `s5`. `s5` will be added to a `Subsystem` `s4` which corresponds to a `Requirement` `r4` that contains `r`. The value of the `name` attribute of `s5` will be set to the `name` of `r`. Furthermore, a correspondence link between `r` and `s5` will be created. For additional operational rules that can be derived from each declarative integration rule automatically the reader is referred to [12].

4.5 Applying the generated Code

As we have shown above, MOFLON generates Java code from all specifications. In particular, MOFLON generates JMI interfaces from the tools' metamodels.

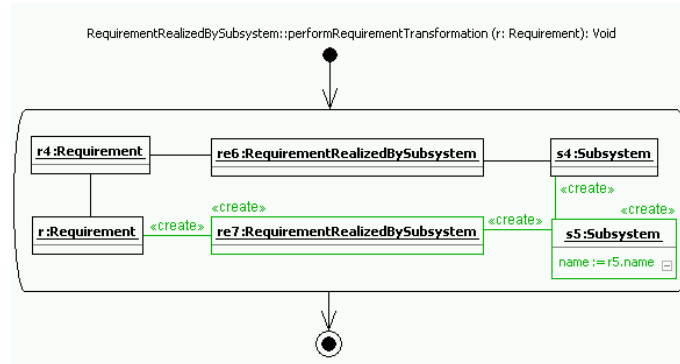


Fig. 9. Example of an operational integration rule

In order to access the tools' data in our running example the implementations of these interfaces have to realize adapters on the tools' APIs. These implementations can either be written manually or generated from templates. The Java code generated from the constraints, model transformations, and model integration operations complies to the JMI interfaces, and can, therefore, be mapped onto tool operations by the adapters. Thereby, using MOFLON we are able to realize the needed tool support for model-driven development related tasks which a developer is confronted with. Note that our case studies does not force us to deal with concrete syntax. Nevertheless, the integration of metamodels generated with MOFLON into graphical editor frameworks is possible as well.

5 MOFLON architecture

In this section, we provide an overview of the internal MOFLON architecture. When we started to design MOFLON in 2003, our goal was to reuse existing technology where possible and focus on conceptual improvements. Back then, there were no MOF 2.0-editors and code generators available, but there were several graph transformation tools. After comparing different approaches, we decided to realize MOFLON on top of the Fujaba Toolsuite which already featured graph transformations for UML-like graph schemata.

Fig. 10 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 data. In the center, repositories symbolize 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 commercial *CASE tool* such as Rational Rose or directly using the new *MOF 2.0 Editor* plugin for Fujaba. Metamodels from external tools are exported as XMI and imported by MOFLON using an *XMI interchange* plugin. As most commercial tools do not yet support MOF 2.0, new features must be entered using certain conventions

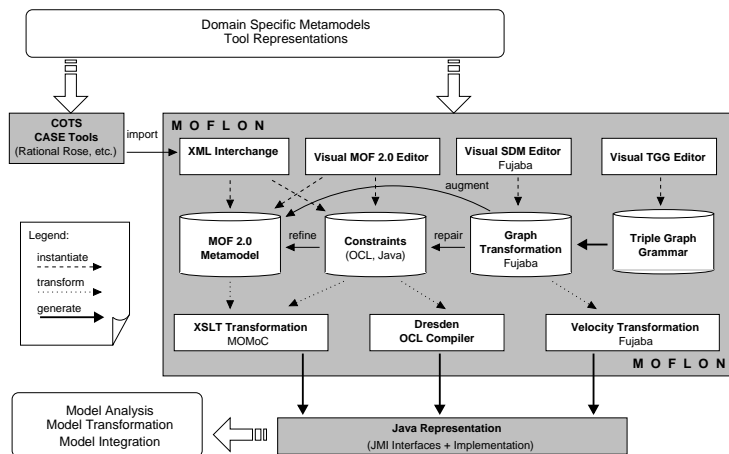


Fig. 10. MOFLON architecture overview

with respect to stereotypes or comments. We would like to mention that we have been able to import the complete UML 2.0 Infrastructure + Superstructure as provided by the OMG⁵, apart from some specification errors, which had to be fixed manually. The *metamodel* is kept in memory, as instance of a JMI-compliant Java representation of the MOF 2.0 metamodel.

Graph transformation rules are edited using the *SDM editor* that already exists in Fujaba. These rules are also kept in memory and augment the MOF 2.0 metamodel instance conceptually, by providing visually specified implementations of methods defined in the schema. In our current implementation, we use the object adapter pattern ([15], p. 141) to map each MOF element to one or more Fujaba metamodel interfaces, which the SDM rules actually are built on. As most adapters are generated from XML descriptions, we could adopt the Fujaba graph rewriting engine with reasonable effort rather than writing our own.

The *TGG editor*, which actually consists of a schema and a rule editor has also been adopted from Fujaba. The *Triple Graph Grammar*, i.e. TGG schema and rules are also stored in memory. 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 and body constraints for methods implemented by graph transformations. Opposed to that, assertions are used to express application conditions in graph transformations. While constraints have been factored out in Fig. 10 to be discussed separately, they are actually stored as strings within the metamodel. Graph transformations are used to define *repair actions* for constraint violations. We

⁵ <http://www.omg.org/cgi-bin/apps/doc?ptc/04-10-05.zip>

also allow to directly enter Java code for constraint expressions as preliminary alternative for OCL constraints.

From the MOF metamodel, JMI-compliant Java code is generated by *XSLT transformation* with an descendant of the MOMoC code generator [2]. This generator also deals with constraints provided as Java code. We are currently working on the integration of the Dresden OCL compiler [8] to generate Java code for OCL 2.0 constraints. Java code for graph transformations is generated using Fujaba's latest *Velocity-based code generator* [11].

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 [6]. Depending on the requirement of the concrete task, we combine the generated code with suitable parsers, tool adapters and user interfaces to create a specific solution for model analysis, transformation, and integration.

6 Related work

Like MOFLON, there are a number of approaches for metamodeling and model transformation, for an extensive overview, cf. [16]. In this section we focus on tools that are specifically interesting for our approach.

Some tools like GReAT [17], MoTMoT [18], and MDR [6] value OMG *standard-compliant schemata* like MOF and UML, while Microsoft Domain Specific Language Tools [19], GME [17], *MetaEdit+* [20], PROGRES [21], prefer proprietary metamodelanguages. Fujaba [9] mixes standard and proprietary elements, EMF Transformation Engine / GMF [22, 23] only implements EMOF, a small subset of the current standard, and tools like AToM³[24] use ER-Diagrams that could be considered "standard" some time ago.

Only MOFLON puts a strong emphasis on complete standard compliance with MOF 2.0 to benefit from its new features. Among these are strong *modularization and refinement* possibilities. Tools like [19, 24] provide no schema modularization at all, others like [9, 17, 20, 21] provide either hierarchies or view mechanism to structure data. MOFLON uses package imports, package merges, element imports, redefinition of association ends etc. with full effect on identifier visibility not only in schemata but also constraints and graph transformations.

Many tools [19, 24, 17, 23, 25, 26], often meta-CASE tools, deal with the *concrete syntax* of modeling languages to create diagram editors. MOFLON is more about model analysis, transformation and integration and hence, does not support concrete syntax.

MOFLON provides *local model transformations* through graph transformations, which is also true for [21, 9, 17, 24, 18]. Other tools [22, 27] only provide textual model transformations or none at all [19, 6]. Like MOFLON, only some of these tools [21, 9, 17, 18] use visual *rule application strategies* to compose large transformations. While Fujaba and hence MOFLON use a proprietary syntax embedding Story Patterns inside UML Activity Diagrams, MoTMoT is truly

standard-compliant by providing an adequate UML profile, but at the cost of defining story patterns as class diagrams, which appears quite unnatural to us.

MOFLON and Tefkat[22] provide declarative model-to-model transformations that are QVT-like. Opposed to that, GReAT only provides the possibility to define operational model-to-model transformations.

7 Conclusion

MOFLON is an integrated, standard-compliant metamodeling environment that provides full support for the new MOF 2.0 modularization and refinement concepts. Due to JMI-compliance and the event mechanism, the model repository can either exist in main memory or in a NetBeans Metadata Repository [6]. Besides, MOFLON features constraint checking based on the Dresden OCL compiler [8]. A visual graph transformation language with control flow diagrams for rule-application strategies has been adopted from Fujaba [9] to perform local model transformations. For model-to-model-integration, we use a declarative QVT-like approach based on Triple Graph Grammars. Finally, MOFLON has a template-based code generator supporting XSLT and Velocity technology.

We are currently completing the first public MOFLON-release. In the near future, we will work on an incremental constraint checking algorithm adapted from PROGRES [28]. Finally, we plan to merge MOFLON with DiaGen II [25] to be able to generate support for both syntax-directed and free-hand diagram editing, constraint-based layout, and parsing of diagrams, thus making MOFLON also a meta-CASE tool.

References

1. Object Management Group: Meta Object Facility (MOF) Core Specification. (2006) formal/06-01-01.
2. Amelunxen, Bichler, Schürr: Codegenerierung für Assoziationen in MOF 2.0. In: Proc. Modellierung 2004, Marburg, Germany. (2004) 149–168 In German.
3. Röttschke, T.: Re-engineering a Medical Imaging System Using Graph Transformations. In: Applications of Graph Transformations with Industrial Relevance. Volume 3062 of LNCS., Springer (2003) 185–201
4. Altheide, F., et al.: An Architecture for a Sustainable Tool Integration. In Dörr, Schürr, eds.: TIS 2003 Workshop on Tool Integration in System Development. (2003) 29–32
5. Dirckze, R.: JavaTM Metadata Interface (JMI) Specification, Version 1.0. Unisys. (2002)
6. Matula, M.: NetBeans Metadata Repository. SUN Microsystems. (2003)
7. Object Management Group: OCL 2.0 Specification. (2005) ptc/2005-06-06.
8. Löcher, S., Ocke, S.: A Metamodel-Based OCL-Compiler for UML and MOF. In Schmitt, P., ed.: Workshop Proc. OCL 2.0 - Industry standard or scientific playground? Volume 102 of Electronic Notes in Theoretical Computer Science., Elsevier (2004) 43–61
9. Zündorf, A.: Rigorous Object Oriented Software Development. University of Paderborn (2001) Habilitation Thesis.

10. Amelunxen, C., Röttschke, T., Schürr, A.: Graph Transformations with MOF 2.0. In Giese, H., Zündorf, A., eds.: Proc. 3rd International Fujaba Days 2005. Volume tr-ri-05-259., Universität Paderborn (2005) 25–31
11. Geiger, L., Schneider, C., Reckord, C.: Template- and Modelbased Code Generation for MDA-Tools. In: 3rd International Fujaba Days 2005, Paderborn, Germany (2005)
12. Königs, A., Schürr, A.: Tool Integration with Triple Graph Grammars - A Survey. In Heckel, R., ed.: Proc. SegraVis School on Foundations of Visual Modelling Techniques. Volume 148 of Electronic Notes in Theoretical Computer Science., Amsterdam, Elsevier Science Publ. (2006) 113–150
13. Amelunxen, C., Königs, A., Röttschke, T., Schürr, A.: MOSL: Composing a Visual Language for a Metamodeling Framework. Submitted to IEEE Symposium on Visual Languages and Human-Centric Computing 2006 (2006)
14. Nagl, M.: Graph-Grammatiken. Vieweg Press (1979) German.
15. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley (1995)
16. Czarnecki, Helsen: Classification Of Model Transformation Approaches. In: 2nd OOPSLA Workshop on Generative Techniques in the context of Model Driven Architecture. (2003) <http://www.softmetaware.com/oopsla2003/czarnecki.pdf>.
17. Agrawal, A., Levendovszky, T., Sprinkle, J., Shi, F., Karsai, G.: Generative Programming via Graph Transformations in the Model Driven Architecture. In: Proc. Workshop on Generative Techniques in the Context of Model Driven Architecture. (2002)
18. Schippers, H., Van Gorp, P., Janssens, D.: Levering UML Profiles to Generate Plugins from Visual Model Transformations. In: Proc. Software Evolution through Transformations. (2004) 7–17
19. Microsoft Corporation: Visual Studio 2005: Domain-Specific Language Tools. <http://msdn.microsoft.com/vstudio/DSLTools/> (2006)
20. MetaCase: MetaEdit+@metaCASE tool. <http://www.metacase.com> (2006)
21. Schürr, A., Winter, A., Zündorf, A. In: PROGRES: Language and Environment. Volume 2. World Scientific (1999) 487–550
22. Lawley, M., Steel, J.: Practical Declarative Model Transformation With Tefkat. In Bézivin, J., Rumpe, B., Schürr, A., Tratt, L., eds.: Proc. Workshop on Model Transformations in Practice. (2005) <http://sosym.dcs.kcl.ac.uk/events/mtip05/>.
23. The Eclipse Foundation: Eclipse Graphical Modeling Framework. <http://www.eclipse.org/gmf/> (2006)
24. De Lara Jaramillo, J., Vangheluwe, H., Moreno, M.A.: Meta-modelling and Graph Grammars for Multi-Paradigm Modelling in AToM³. Software & Systems Modeling **3**(3) (2004) 194–209
25. Minas, M.: Concepts and Realization of a Diagram Editor Generator-based on Hypergraph Transformation. Science of Computer Programming **44** (2002) 157–180
26. Böhlen, B., Jäger, D., Schleicher, A., Westfechtel, B.: UPGRADE: A Framework for Building Graph-Based Interactive Tools. In Mens, T., Schürr, A., Taentzer, G., eds.: Proc. International Workshop on Graph-Based Tools. Volume 72(2) of Electronic Notes in Theoretical Computer Science. (2002)
27. Jouault, F., Kurtev, I.: Transforming Models with ATL. In: Proc. Workshop on Model Transformations in Practice. (2005)
28. Münch, M.: Generic Modelling with Graph Rewriting Systems. PhD thesis, RWTH Aachen (2002)