MODGRAPH
A Transformation Engine for EMF Model Transformations

Thomas Buchmann, Bernhard Westfechtel and Sabine Winetzhammer
University of Bayreuth, Universitaetsstrasse 30, 95440, Bayreuth, Germany

Keywords: Model-driven development, Graph transformations, Code generation, EMF, Pattern matching, OCL.

Abstract: Model-driven software engineering aims at increasing productivity by replacing conventional programming with the development of high-level executable models. However, current technology focuses on structural models, while behavioral modeling is still neglected. The transformation engine ModGraph intends to fill this gap. ModGraph complements the Eclipse Modeling Framework with graphical transformation rules from which executable code is generated. An operation defined in an Ecore model is specified by a model transformation rule which is compiled into a Java method calling EMF operations. In this way, ModGraph complements the capabilities of EMF which would compile operations into empty Java methods. The net result is an environment which provides comprehensive support for executable models.

1 INTRODUCTION

The *Eclipse Modeling Framework (EMF)* (Steinberg et al., 2009) has been used successfully in a wide range of applications in both research and industry. In EMF, models are based on the object-oriented Ecore metamodel, in which classes as well as their structural and behavioral features are defined. However, concerning behavior, the modeling capabilities of EMF are limited: Only the signatures of operations may be defined. Thus, Ecore may be used to define structural models only. The EMF code generator creates code for the classes in an Ecore model. But this code comprises only elementary operations for setting attribute values, inserting and deleting links, etc. For user-defined operations, only a skeleton is generated consisting of the signature and an empty body. The user needs to supply an implementation for each user-defined operation. Usually, this is done by programming in Java.

Thus, *model-driven software engineering* is supported only partially by EMF. In this paper, we introduce *ModGraph*[1], a transformation engine for *EMF model transformations*, which adds behavioral modeling and code generation to EMF. In ModGraph, model transformations may be defined for any model defined with Ecore. ModGraph is based on the theory of *graph transformation* and views an EMF model instance as a directed graph. A model transformation rule is used to realize an operation defined in the Ecore model and describes a model transformation in a declarative way. The core of a transformation rule consists of a graph pattern, which specifies both a set of objects and links to be searched and the changes to be performed. In ModGraph, model transformations are specified at a much higher level of abstraction than Java code. A model transformation rule is validated against the underlying Ecore model. Furthermore, a valid rule is transformed into executable code which is injected into the code generated by the EMF code generator. Thus, ModGraph is tightly integrated with the EMF framework.

ModGraph adds support for behavioral modeling to EMF in an incremental way. This approach is in line with the philosophy underlying EMF, namely to offer an evolutionary path from Java programming to modeling. Using the EMF core technology, an application developer benefits from structural modeling and code generation for elementary operations. Using ModGraph, (s)he may use graph transformations whenever they are helpful to describe complex operations on EMF model instances in a declarative and graphical way. The ModGraph compiler will then compile these rules into ordinary Java methods. For the "rest", the application developer may still resort to plain Java programming. In particular, the control structure of an application resides completely in Java since Java — and other conventional textual program-

[1]ModGraph is an acronym of "*Mod*el Transformation with *Graph* Transformation".

Winetzhammer S., Buchmann T. and Westfechtel B..
MODGRAPH - A Transformation Engine for EMF Model Transformations.
DOI: 10.5220/0003606702120219
In *Proceedings of the 6th International Conference on Software and Database Technologies* (ICSOFT-2011), pages 212-219
ISBN: 978-989-8425-77-5

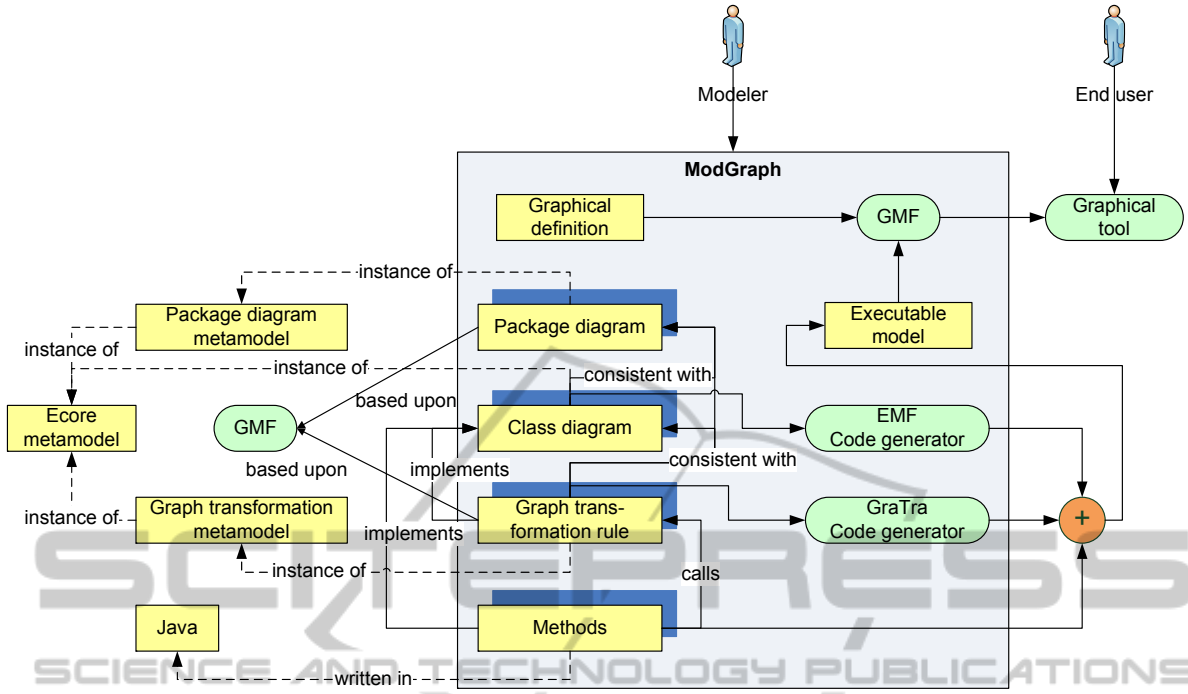

Figure 1: Overview about the different meta-models and models used in ModGraph.

ming languages — has proved to offer a concise and expressive notation for control flow.

2 OVERVIEW

In this section we provide an overview about ModGraph and its context. Figure 1 depicts a diagram showing the different modeling components and the interaction with the modeler and the end user. When specifying a software system, the modeler usually starts on a coarse-grained level, by defining an architecture. In our approach, package diagrams are used for this task. A package diagram describes the coarse structure of the software system by packages and their dependencies (public/private package and element imports are used for this purpose). We developed a package diagram editor which provides the modeler with modeling-in-the-large capabilities (Buchmann et al., 2009). This editor was developed in a model-driven way using the Eclipse Modeling Framework (EMF) (Steinberg et al., 2009) and the Graphical Modeling Framework (GMF) (Gronback, 2009). Thus, its metamodel is an instance of the Ecore metamodel.

In a subsequent step, the modeler can refine this package diagram with an Ecore class diagram. To create Ecore class diagrams, any Ecore compliant editor can be used, e.g., the tree editor which is part of Eclipse's EMF distribution or graphical editors like the Ecore Tools editor or even commercial tools like eUML2 (Soyatec, 2011). The package diagram editor provides several methods of use to maintain consistency with the corresponding class diagrams, including forward, reverse, incremental round-trip engineering and an additional validation.

In the following step, the modeler can refine an operation declared in the class diagram by a graph transformation rule. This graph transformation rule is an instance of the graph transformation metamodel which is also an instance of the Ecore metamodel. The graphical editor which allows the modeler to graphically specify the graph transformation rule was also developed in a model-driven way using EMF and GMF.

In addition, hand-written method implementations in plain Java code are used to call the graph transformation rules.

To generate code from the specification created by the modeler, two different code generation steps are used. First, the EMF code generator is used to generate EMF compliant code according to the Ecore class diagram. In a subsequent step, the method bodies which are specified by graph transformation rules are generated and merged with the code generated in the previous step. The different code fragments - the generated ones and the hand written ones - result in an executable model. An additional advantage of our

approach is that GMF can be used to build graphical tools on top of the executable model, since Ecore class diagrams are used to model the static structure.

3 GRAPH TRANSFORMATION RULES

This section goes into detail concerning the graph transformation rules, which may be used to model behavior on a fine grained level.

3.1 General Description

Implementing an EMF operation can lead to large method bodies coded on a low level of abstraction. Our graph transformation rules provide a compact and declarative way to model complex operations. Each graph transformation rule represents the method body for an operation declared in the Ecore model. The parameters of this operation represented EMF objects and data values. Object parameters are used to bind nodes of the graph pattern. Data values are used e.g. in attribute conditions and attribute assignments.

EMF model instances are considered as graphs — called model graphs — whose nodes and edges represent objects and links, respectively. A graph transformation rule declaratively describes a transformation of the model, consisting of structural changes (insertion/creation of objects and links), changes of attribute values, and nested method calls. The core part of a graph transformation rule is a graph pattern, in which objects and links are represented by nodes and edges, respectively. The graph pattern specifies both the subgraph to be searched and the operations to be performed. The graph pattern may be constrained using pre- and postconditions as well as a negative application condition (NAC).

A graph pattern resembles a UML communication diagram, which is composed of a static part — the underlying object diagram — and a dynamic part (specifying change operations). The nodes of a graph pattern are classified into the current object (denoted by this), parameter objects, single objects and multi-objects (representing sets of objects). The current object and the parameter objects are bound via the method call. All other objects are unbound and need to be searched in the model graph. Unbound objects may be marked with <<out>> to designate them as return objects of the graph transformation rule.

Nodes are connected by two kinds of edges: paths and links. Paths stand for derived references and are marked with expressions. A path expression is written in OCL or Java. It is evaluated on the source node, resulting in all target nodes belonging to the model graph and fulfilling the expression. Links represent instances of references declared in the Ecore model on which the graph transformation rule is based.

All unbound nodes of the graph pattern need to be typed with classes from the underlying Ecore model. Bound nodes are not typed because the types are derived from the owning class (in the case of the current object) and the operation signature (in the case of parameter objects). Links are typed with references from the Ecore model.

Nodes in a graph pattern contain compartments for specifying constraints and changes. In the constraints compartment, OCL constraints and conditions on values of single attributes (in Java syntax) may be defined. The changes compartment may contain attribute assignments as well as other Java statements (e.g., method calls).

To model structural changes, the elements of a graph pattern may be decorated with a state. ++ and - - indicate the states “created” and “deleted”, respectively. An element without decoration owns the default state “preserved”. Bound nodes may not be created, and the current object may not be deleted. If the graph pattern does not contain any changes either in its structural part or in the nodes’ change compartments, it describes a graph test which does not have any effect on the model graph.

Preconditions are checked before the graph pattern is searched; they provide additional application conditions. Postconditions are checked immediately before the end of the execution of the graph transformation rule to ensure that the model graph is in a valid state. Both pre- and postconditions are written as OCL constraints.

A negative application condition (NAC) is checked after a match of the graph pattern has been located and before the transformation is executed. If the pattern specified in the NAC occurs in the model graph, the rule cannot be applied to the current match. If no other match may be found, the rule fails altogether. An NAC is composed of nodes of the graph pattern, all of which are bound at this stage of execution, and (optionally) further nodes, all of which are unbound. In contrast to the graph pattern, an NAC may describe only a graph test rather than a graph transformation.

3.2 Example

To show the functionality, a widely known example from an EMF tutorial[2], the library model, is adapted

[2]http://help.eclipse.org/ganymede/index.jsp?topic=/org.eclipse.emf.doc/references/overview/EMF.html

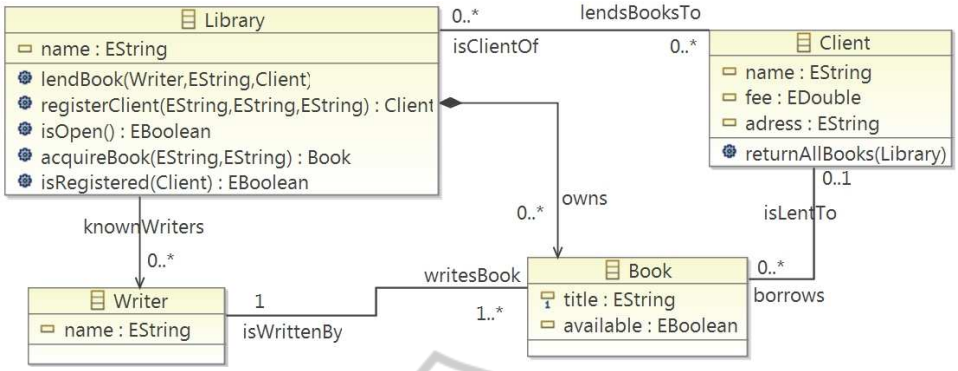

Figure 2: Ecore model of a simple library management system.

(Figure 2). The model defines a library with a name, that owns books and lends them to its clients. Each book has a title, is available or not, and is associated to exactly one writer. This writer has a name and is known to the library. Clients also have a name and an address. In addition, they have to pay a fee if they borrow a book longer than allowed.

The classes Library and Client define some operations. In the following we show the implementation of three operations with the help of ModGraph transformation rules. The rules have been designed for the purpose of demonstration. They are simple to understand, yet they cover most of the elements of graph transformation rules described in the previous subsection.

The rule acquireBook shown in Figure 3 acquires a book for the library provided that the author is not known yet. It receives two parameters title and authorsName of type String. The OCL precondition given on the top requires that both strings must not be null. If the precondition does not hold, the rule fails. The graph pattern below the precondition specifies that a new writer is added to the known writers of the library. Furthermore, a new book associated to the library and the writer is created. This new book is marked with <<out>> as the return object of the rule. In the changes compartment, the parameter title is assigned to the attribute of the same name. Furthermore, the books' availability is set to true, hence it can be lent to a client. The NAC to the right of the graph pattern is checked before the transformation is applied. The NAC succeeds if there already exists a writer with the name authorsName. If the NAC succeeds, the rule fails without having performed any changes.

The rule lendBook (Figure 4) lends a book owned by the library to one of its clients if the book is available and there are no unpaid fees charged to the client. A precondition forcing the library to be open and the client to be registered is set as the application condition to be checked before applying the rule. Alternatively, this precondition could have been specified in the constraints compartment of the current object. In addition to the current object, the graph pattern consists of two objects bound by the input parameters client and author (rounded rectangles), and an unbound object of class Book (since only the title was supplied as an input parameter, the book object needs to be searched in the model graph). The constraints compartment of the book object checks that the book has the title supplied as input parameter and that it is available. The constraints compartment of the client object ensures that there are no unpaid fees. If the pattern match succeeds, a link from the book object to the client object is created, and the book is marked as not available in its changes compartment.

Finally, returnAllBooks returns all books borrowed by some client in a single step (Figure 5). This rule is executed on an object of class Client. The rule demonstrates the use of a multi-object books representing a set of books. This set is determined by intersecting the target sets of the references borrows and owns emanating from the current object and the parameter object lib, respectively. This means that the set will contain all books which have been borrowed by the client and are owned by the library (please notice that the client may have borrowed books from other libraries, which are not considered in this operation). If the pattern match succeeds, all borrows links between the current object and books objects are deleted, and all books objects are marked as available.

4 IMPLEMENTATION

ModGraph provides a graphical editor, different mechanisms for validation, and a code generator. For their realization, several Eclipse based tools have been used.

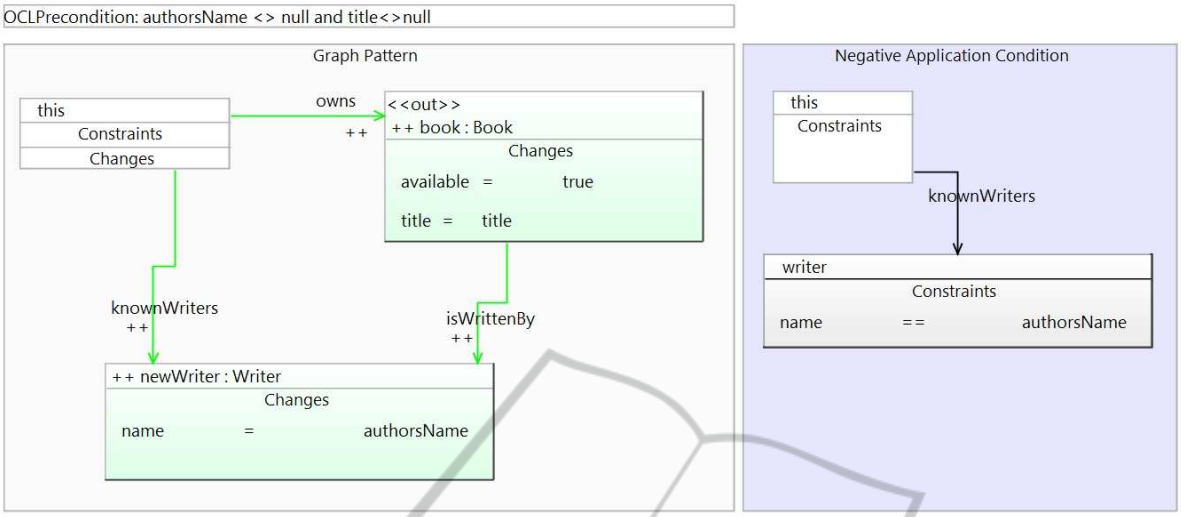

Figure 3: ModGraph rule for Library::acquireBook(title : String, authorsName : String) : Book.

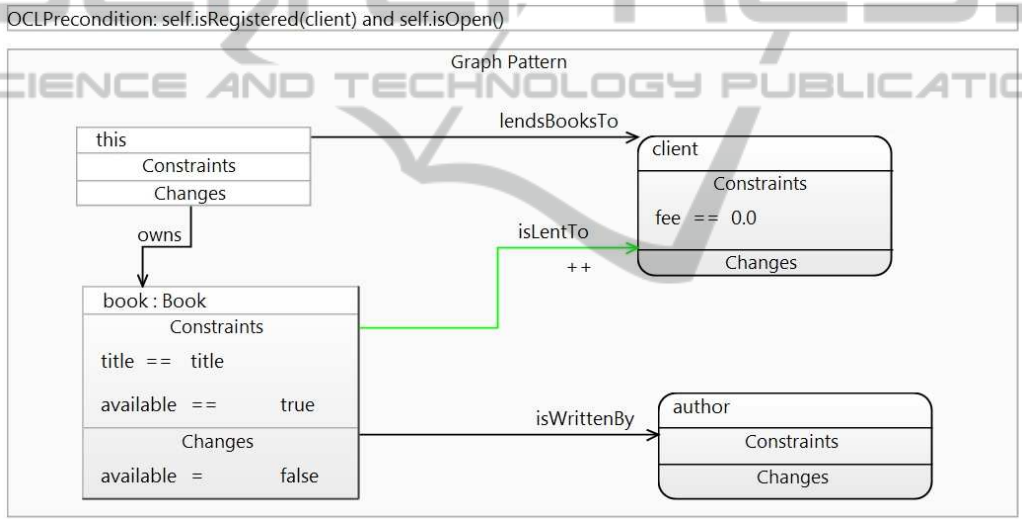

Figure 4: ModGraph rule for Library::lendBook(author : Writer, title: String, client : Client).

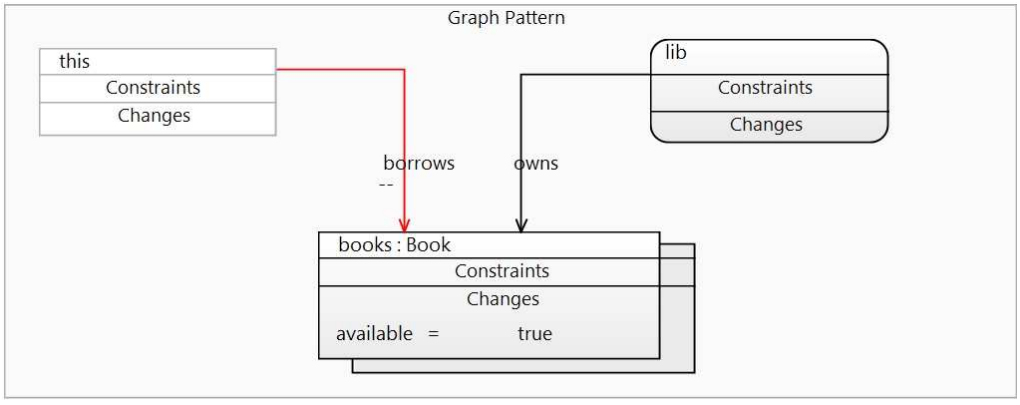

Figure 5: ModGraph rule for Client::returnAllBooks(lib : Library).

4.1 Graphical Editor and Validation

The ModGraph graphical editor has been developed in a model-driven way using the Eclipse Modeling Framework (EMF) and the Graphical Modeling Framework (GMF). The graph transformation rules shown in Figures 3–5 were produced as snapshots of the graphical view provided by the editor. In addition, the editor displays a property view and a tool palette.

Correctness of a ModGraph graph transformation rule is achieved using several mechanisms:

1. A graph transformation rule must conform to the metamodel underlying ModGraph. For example, nodes in NACs may not be marked as created or deleted. The Ecore model for ModGraph is built in such a way that errors of this kind cannot occur.
2. The graphical editor performs live validation during editing. Commands which would violate the rules of live validation are rejected. Live validation is realized by GMF constraints. For example, a link to be preserved must not be connected to a node to be deleted. Furthermore, we implemented restrictions in the source code. For example, the properties view allows only to select valid references for typing a link between two objects (a reference is valid if it is owned by the class of the source and the class of the target conforms to the target class of the reference).
3. All remaining errors are caught by batch validation. After a batch validation, the user will be informed about the rule's validity. In case of failures, error markers are shown in the diagram, and the Eclipse problems view shows error messages. For the implementation of the batch validation we used the EMF-Validation and the Check language (oAW[3]). Check is a validation language of the former openArchitectureWare, which is now part of Eclipse (MDT[4]). The ModGraph Check validation is called by an MWE-workflow (MWE[5]), which loads the user's model and performs all given Check files, containing various constraints. Batch validation catches errors which cannot be checked conveniently by live validation. For example, batch validation verifies the reachability of nodes, which is important for pattern matching (see below), and performs a syntactic check on OCL constraints.

[3]http://openarchitectureware.org/

[4]http://www.eclipse.org/modeling/

[5]http://wiki.eclipse.org/Modeling_Workflow_Engine_(MWE)

4.2 Code Generation

When the user invokes the code generator on a graph transformation rule, the rule is automatically validated. If validation succeeds, code will be generated. Since each rule implements an operation defined in the underlying Ecore model, code is generated directly into the EMF generated code. After parsing the EMF generated code, the code generator for graph transformation rules uses Xpand templates which were formerly provided by openArchitectureWare and were recently included into the Eclipse Modeling Project. These templates are, like the validation, called by an mwe-workflow.

The generated code will execute a graph transformation rule in the following steps:

1. The preconditions of the rule are checked. If one of the preconditions is violated, execution fails.
2. An instance of the graph pattern is searched in the model graph. If no match is found, execution fails.
3. If a match is found and an NAC is present, it is attempted to find a match for the NAC. If the matching succeeds, a forbidden pattern was located. Accordingly, rule execution returns to the previous step, trying to find the next match for the graph pattern.
4. Now, the changes specified in the graph pattern are applied to the located match.
5. Finally, the postconditions are checked. If one of the postconditions is violated, execution fails[6]. Otherwise, execution terminates successfully.

Pattern matching constitutes a crucial step in the execution of a graph transformation rule. Pattern matching starts at the bound nodes and needs to locate instances of the unbound nodes of the model graph. We require that all unbound nodes are reachable from the bound nodes via a directed path. In the case of bidirectional references, opposite links are automatically considered in the search, as well. Thus, pattern matching is performed exclusively by navigation and does not assume other mechanism for locating objects such as key attributes and indices.

The performance of pattern matching is severely affected by the order in which unbound nodes are searched. In the case of multi-valued references, all targets of outgoing links need to be considered in the matching process. Since these targets may have outgoing multi-valued references, as well, the set of candidates may grow combinatorially. To bound the can-

[6]In this case, an exception is raised which has to be handled by the caller, e.g., by running a compensating graph transformation rule.

didate set, we implemented a heuristic greedy algorithm which assigns costs to references according to the upper bounds defined in the Ecore model. In order to extend the current partial match, all edges from already matched to unmatched nodes are considered in parallel. From this set of edges, an edge with minimal costs is selected. In this way, a spanning forest of the graph pattern is constructed. In particular, the pattern matching algorithm considers single-valued references before multi-valued references. If only single-valued references are used to locate the unbound nodes, there is at most one candidate for each of them.

5 RELATED WORK

Research on model-driven software engineering has produced a wide spectrum of model transformation approaches (Czarnecki and Helsen, 2006). The term model-to-model transformation indicates that some target model is created from some source model. For example, ATL (Jouault and Kurtev, 2005) and QVT (OMG, 2011) address model-to-model transformations. In contrast, ModGraph supports updating model transformations, which modify an existing model.

In contrast to textual model transformation languages such as ATL and QVT, ModGraph provides a graphical notation (with embedded textual fragments). The graphical notation makes it easier to understand the effects of a complex model transformation. Furthermore, ModGraph is based on the theory of graph transformations. Many languages, tools, and environments have been developed for graph transformations (Ehrig et al., 1999). However, in most cases there is no connection to the EMF framework. There is only a small number of graph transformation approaches which are based on EMF. In the following, we will focus on these approaches because they are related most closely to ModGraph.

VIATRA2 (Varró and Balogh, 2007) provides a textual rather than graphical language for graph transformations. VIATRA2 has been built with EMF; it defines its own metamodel with the help of Ecore. In contrast, ModGraph has been built for EMF. In particular, Ecore is reused for structural modeling. As we have demonstrated, ModGraph is integrated seamlessly into the EMF framework and extends it with graph transformations.

Like ModGraph, TIGER (Biermann et al., 2006) exploits Ecore for structural modeling. However, TIGER supports only graph transformation rules which lack many of the advanced constructs implemented in ModGraph (multi-objects, pre- and post-conditions, paths, method calls, and OCL integration). Its successor Henshin (Arendt et al., 2010) extends TIGER's graph transformation rules with multi-objects and control structures, but the other features mentioned above are still missing.

In TIGER, rules are compiled into classes rather than methods. When an application developer wants to implement a user-defined operation introduced in an Ecore model with the help of a graph transformation rule, (s)he must write code to instantiate the rule's class, to provide the rule instance with parameters, and to execute the rule. Thus, calling a rule involves multiple steps which are awkward to implement and cause considerable overhead at run time. In contrast, in ModGraph a graph transformation rule is associated to a user-defined operation, and the ModGraph compiler generates an ordinary Java method. From an application developer's point of view, ModGraph provides both seamless and efficient integration of the Java code generated from graph transformation rules. This is not the case for TIGER. So far, Henshin provides only an interpreter, which is of limited use for conventional application development.

Fujaba (Zündorf, 2001) is an object-oriented modeling environment which offers class diagrams for structural modeling and story diagrams for behavioral modeling. A story diagram is a control flow diagram containing statement activities (plain Java code fragments) and story patterns (graph transformation rules). Fujaba was developed outside EMF, but was re-implemented partially in EMF (Giese et al., 2009). The re-implementation — called MDELab — uses Ecore for structural modeling and provides a graphical editor and an interpreter for story diagrams. Thus, the end user is limited to the interpreter when working with model instances. In our approach, the generated EMF compliant code can be used in all possible ways, including the generation of a graphical editor using GMF.

Fujaba's story patterns are less expressive than ModGraph's graph transformation rules because only the latter offer pre- and postconditions, OCL integration[7], and negative application conditions defined as separate forbidden graph patterns. Furthermore, the control flow of story diagrams is limited in its expressive power since it lacks explicit high-level control structures. For these reasons, ModGraph resorts to Java for expressing the control flow.

[7] A prototypical integration, which did not make its way into the Fujaba release, was reported in (Stölzel et al., 2006).

6 CONCLUSIONS

We have presented ModGraph, an environment for modeling with graph transformations. ModGraph incrementally adds behavioral modeling to EMF. Thus, it provides an evolution path from programming to modeling. An application developer may take advantage of model transformations where they provide an added value, and program in Java otherwise. The examples given in Section 3 clearly demonstrate the benefits of modeling behavior graphically and declaratively with the help of graph transformation rules.

Implementation of ModGraph currently is under way. Most parts have already been completed. The package diagram editor and its integration with Ecore have been implemented completely. The metamodel for graph transformation rules, the graphical editor, and rule validations have been completed recently. The code generator is currently being implemented and is expected to be complete in the near future. Since the graphical editor is already available, we are performing case studies in parallel in order to obtain feedback from applications. The experiences gained from these case studies are promising, and we expect at most minor future changes of the transformation metamodel.

REFERENCES

Arendt, T., Biermann, E., Jurack, S., Krause, C., and Taentzer, G. (2010). Henshin: Advanced concepts and tools for in-place EMF model transformations. In Petriu, D. C., Rouquette, N., and Haugen, Ø., editors, *Proceedings 13th International Conference on Model Driven Engineering Languages and Systems (MODELS 2010), Part I*, volume 6394, pages 121–135, Oslo, Norway.

Biermann, E., Ehrig, K., Köhler, C., Kuhns, G., Taentzer, G., and Weiss, E. (2006). Graphical definition of in-place transformations in the eclipse modeling framework. In Nierstrasz, O., Whittle, J., Harel, D., and Reggio, G., editors, *Proceedings 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2006)*, volume 4199, pages 425–439, Genova, Italy.

Buchmann, T., Dotor, A., and Klinke, M. (2009). Supporting modeling in the large in fujaba. In van Gorp, P., editor, *Proceedings of the 7th International Fujaba Days*, pages 59–63, Eindhoven, The Netherlands.

Czarnecki, K. and Helsen, S. (2006). Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–646.

Ehrig, H., Engels, G., Kreowski, H.-J., and Rozenberg, G., editors (1999). *Handbook on Graph Grammars and Computing by Graph Transformation: Applications, Languages, and Tools*, volume 2. World Scientific, Singapore.

Giese, H., Hildebrandt, S., and Seibel, A. (2009). Improved flexibility and scalability by interpreting story diagrams. In Boronat, A. and Heckel, R., editors, *Proceedings of the 8th International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2009)*, volume 18 of *Electronic Communications of the EASST*, York, UK. 12 p.

Gronback, R. C. (2009). *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*. The Eclipse Series. Boston, MA, 1st edition.

Jouault, F. and Kurtev, I. (2005). Transforming models with ATL. In Bruel, J.-M., editor, *MoDELS Satellite Events*, volume 3844, pages 128–138.

OMG (2011). *Meta Object Facility (MOF) 2.0 Query/View/Transformation, v1.1*. OMG.

Soyatec (2011). euml2 studio edition 3.6.0.20110120.

Steinberg, D., Budinsky, F., Paternostro, M., and Merks, E. (2009). *EMF Eclipse Modeling Framework*. The Eclipse Series. Boston, MA, 2nd edition.

Stölzel, M., Zschaler, S., and Geiger, L. (2006). Integrating OCL and model transformations in Fujaba. In Chiorean, D., Demuth, B., Gogolla, M., and Warmer, J., editors, *Proceedings of the 6th OCL Workshop OCL for (Meta-)Models in Multiple Application Domains (OCLApps 2006)*, volume 5 of *Electronic Communications of the EASST*, Genova, Italy. 16 p.

Varró, D. and Balogh, A. (2007). The model transformation language of the VIATRA2 framework. *Science of Computer Programming*, 68(3):214–234.

Zündorf, A. (2001). Rigorous object oriented software development. Technical report, University of Paderborn, Germany.