Electronic Communications of the EASST
Volume 58 (2013)

Proceedings of the 12th International Workshop on Graph Transformation and Visual Modeling Techniques (GTVMT 2013)

ModGraph meets Xcore: Combining Rule-Based and Procedural Behavioral Modeling for EMF

Sabine Winetzhammer and Bernhard Westfechtel

13 pages

Guest Editors: Matthias Tichy, Leila Ribeiro
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

ModGraph meets Xcore: Combining Rule-Based and Procedural Behavioral Modeling for EMF

Sabine Winetzhammer[1] **and Bernhard Westfechtel**[1]

[1][firstName].[lastName]@uni-bayreuth.de
Applied Computer Science 1 - Software Engineering
University of Bayreuth, Germany

Abstract: Model-driven software engineering aims at increasing productivity by developing high-level executable models. The Eclipse Modeling Framework (EMF) significantly contributes toward this goal. Unfortunately, EMF supports only structural models based on the Ecore metamodel. Recently, Xcore has been developed to extend EMF with behavioral modeling. To this end, Xcore provides a single textual language for both structural and behavioral modeling. While Xcore follows a procedural approach to behavioral modeling, ModGraph is an EMF-based tool based on a rule-based paradigm (graph transformation rules, which allow to specify behavior in a declarative way). The combination of EMF, Xcore, and ModGraph results in an environment for model-driven software engineering which provides full-fledged support for both structural and behavioral modeling. Altogether, we obtain an environment in which software engineers are concerned only with models rather than with programs.

Keywords: Model-Driven Software Engineering, Behavioral Modeling, Graph Transformation Rules, Control Flow, ModGraph

1 Introduction

The *Eclipse Modeling Framework* (*EMF*) [SBPM09] has been designed with the intent to improve the software process by providing lightweight support for model-driven software engineering. For this reason, EMF provides a fairly minimalistic metamodel for structural modeling (*Ecore*, an implementation of Essential MOF (EMOF)). Using the components of the EMF core, software engineers create EMF models as instances of the Ecore metamodel. Furthermore, they generate code from an EMF model which provides basic operations for creating objects, assigning attribute values, as well as creating and deleting links. The generated code ensures the model's semantics, in particular with respect to references (which may be bidirectional and designated as containment references).

Unfortunately, the EMF core is confined to *structural modeling*. As a consequence, the code generator may create code only for the basic operations mentioned above. For user-defined operations, only methods with empty bodies may be generated. Software engineers need to extend the generated code, implying that they need to work on two different levels of abstraction. Thus, model-driven software engineering is supported only partially. ("Swiss cheese" approach to code generation.)

To close this gap, structural modeling needs to be complemented with *behavioral modeling*. To this end, we have developed *ModGraph* [Win12], which augments EMF with graph transformation rules operating on EMF model instances. With ModGraph, complex in-place model transformations may be specified at a high level of abstraction. Its rule language supports parameterized graph transformation rules with pre- and postconditions, multi-objects, negative application conditions, and OCL expressions for specifying constraints, attribute values, and paths. Each rule refines an operation defined in an EMF model. ModGraph is tightly integrated with the EMF code generator since it extends the generated EMF code with the code generated from graph transformation rules.

So far, however, the combination of EMF and ModGraph did not provide *total model-driven software engineering*, which would imply that software engineers may generate complete applications from structural and behavioral models. ModGraph covers only rule-based behavioral modeling. In virtually all practically relevant applications, *control structures* are required in addition to determine which rules are applied in which order. Furthermore, experiences gathered from inspecting large case studies indicate that the use of graph transformation rules pays off only for complex model transformation tasks [BWW12a]. In simpler cases, procedural approaches to behavioral modeling are sufficiently expressive and more concise than graph transformation rules.

So far, a software engineer using EMF and ModGraph has defined the structural model in EMF, has used ModGraph for sufficiently complex operations which may be specified as single graph transformation rules, and has encoded "the rest" in Java. Thus, operations have been implemented in Java if (a) control structures are required to compose graph transformation rules or (b) the operation is so simple that it does not pay off to write a graph transformation rule. Altogether, behavioral modeling has been supported only partially.

In this paper, we present the integration of ModGraph with *Xcore* [Ecl12], which has been developed recently to add behavioral modeling to EMF. Xcore strives for total model-driven software engineering by providing a single comprehensive language for both structural and behavioral modeling. To this end, Xcore introduces a textual syntax for EMF models as well as procedural behavioral models. Xcore is driven by the vision that software engineers need no longer deal with code in a programming language such as Java (as current programmers do not inspect assembly or byte code).

The integration of Xcore and ModGraph offers the following benefits:

- Complex transformations which are awkward to program in Xcore may be specified with ModGraph's high-level graph transformation rules.
- Graph transformation rules may be composed with control structures provided by Xcore.
- Simple operations may be encoded exclusively in Xcore.
- Complete application code may be generated by relying on the code generators of EMF, Xcore, and ModGraph.
- Current and future work aims at re-targeting the ModGraph code generator to Xcore, to gain *platform independence* for ModGraph: Generating code for a specific programming language may be delegated completely to Xcore.

The rest of this paper is structured as follows: Section 2 provides an overview about ModGraph, XCore, EMF and their interaction. Section 3 shows the interaction within a running example. Section 4 confines our approach from others while Section 5 concludes the presented facts as well as current and future work.

2 Overview

This section provides an overview about the ModGraph graph transformation rules, the Xcore modeling language, and their interaction with each other and EMF.

2.1 ModGraph Graph Transformation Rules

Within a ModGraph graph transformation rule you can model the behavior of an operation specified in an Ecore class diagram or in an Xcore model. Therefore an instance of the Ecore class diagram is considered as a graph, typed over the class diagram. The core of a ModGraph rule is the Graph Pattern. It includes the subgraph to be searched in the model instance and the changes to be performed. The Graph Pattern can be constrained using pre- and postconditions written in OCL or using a negative application condition modeled as a graph.

Modeling the Graph Pattern you may use several kinds of nodes and edges. Nodes are distinguished into a current object, named `this`, bound nodes, representing the non-primitive parameters of the operation, and unbound nodes, representing the objects to be searched in the model instance. Both may be single- (simple object and parameter) or multivalued nodes (multiobject and multiparameter). Nodes provide a status which may be preserved (grey, no marker), created (green, $++$), or deleted (red, $--$). They can be marked as return parameter ($<<$out$>>$) or as optional ($<<$optional$>>$) nodes. Nodes to preserve or to delete may be constrained, nodes to create or to preserve may be modified, for example by setting an attribute value or calling an operation (operation calls allow ModGraph rules to interact directly with each other).

All nodes may be connected by two kinds of edges, links (instances of references) and paths (derived references). Analogously to nodes links provide a status. Creation links, instantiating multivalued references may be ordered. Paths are marked with a path expression, written in OCL. For details on the metamodel of the graph transformation rules see [BWW12b].

Each rule is validated against the meta model of the graph transformation rule and the user defined Ecore model. After a successful validation the modeler may generate code from the rule. This code is injected right into the EMF or Xcore generated code, as described in [Win12].

2.2 Xcore

Xcore provides a concrete syntax for Ecore. Xbase [EEK+12] is used to model behavior, i.e. the bodies of operations. Both languages are specified in Xtext [Xte12], an open-source framework to develop domain specific languages. Xcore is a modeling and a programming tool. Its goal is to eliminate the dividing gap between modeling and programming.

Xbase itself is an expression language that was designed to be reused in any domain specific language. Its expressions provide procedural programming constructs: control structures and

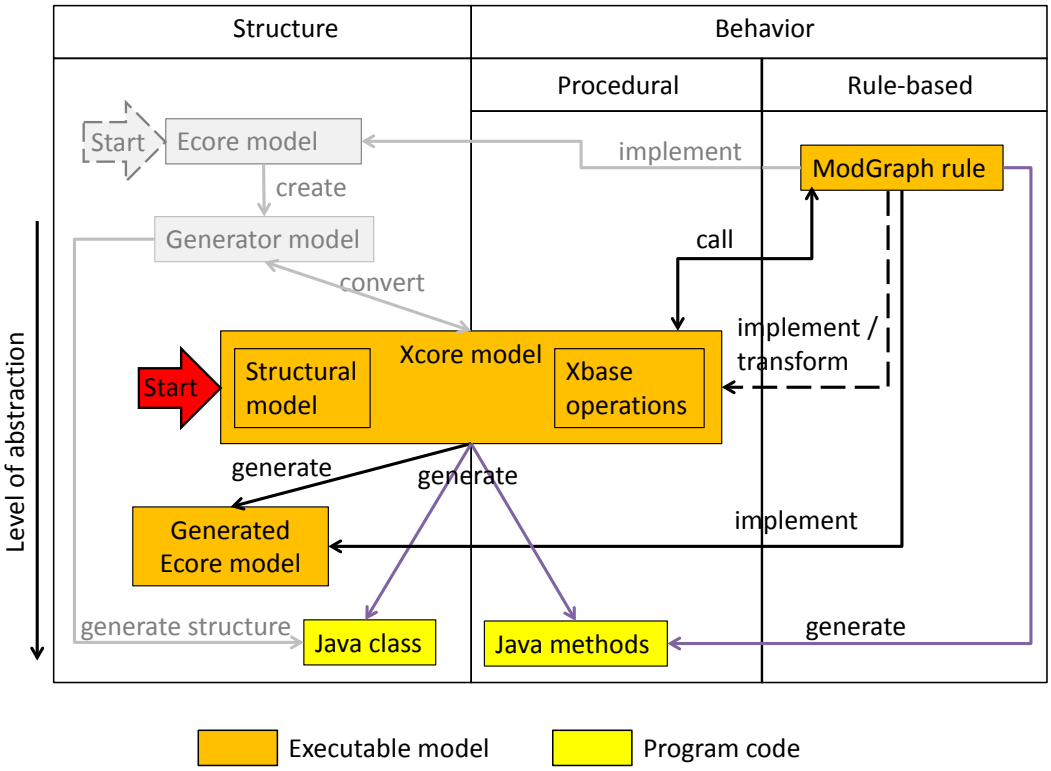


Figure 1: Overview about the tools and their interaction with ModGraph.

programming expressions. From a modeler's point of view we constitute that Xbase used with Xcore lifts Java on a modeling level, though the Xbase syntax is very Java-like.

Any existing Ecore model can be converted into an Xcore model via the generator model and then extended with behavior modeling.

2.3 Modeling with ModGraph, Xcore and Ecore

Here we show the different and permeable ways ModGraph, Xcore and EMF interact. Figure 1 depicts the interactions. We distinguish strictly structural from behavioral modeling. Furthermore behavioral modeling is partitioned into procedural and rule-based behavioral modeling. Specifying a model, the modeler may use the following approaches:

1. The new way: Starting with an Xcore model, marked with a solid arrow denoted with 'Start' (red) in Figure 1, the modeler may define the structural model and simple or procedural operations directly. Complex operations may be specified by ModGraph rules, taking the added value of graph transformation rules into account. Doing so, the modeler marks an operation in the Xcore file and selects 'Implement as ModGraph gt rule' from the popup menu shown in Figure 2, depicted by the dashed arrow 'implement' in Figure

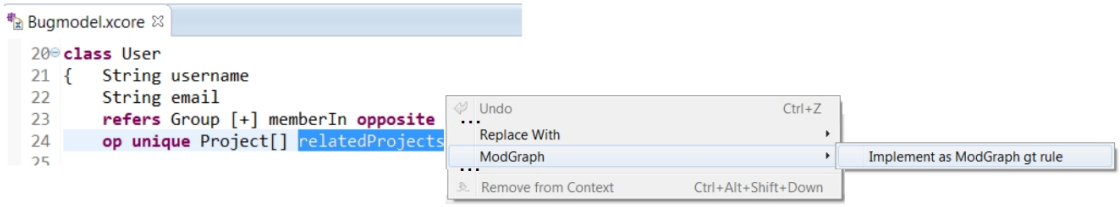


Figure 2: Implement Xcore operation as ModGraph rule.

1. This arrow is also marked with 'transform', because current work aims at a model-to-model transformation from ModGraph to Xcore. The code ModGraph generates for rules implementing Xcore operations is injected right into the Xcore generated code. Hence, ModGraph operations may call and may be called by Xcore operations.

2. The classical way: Starting with the dashed arrow marked with 'Start' (grey) in Figure 1 the modeler may still model a ModGraph rule based upon an Ecore class diagram. One may generate the static structure as usual via the generator model and then inject the ModGraph generated code right into the EMF generated one. This approach is described in detail in [BWW12b].

3. Up to the new: Migrating an existing Ecore model, the modeler may convert the model into an Xcore model via the generator model marked with 'convert' in the grey part of Figure 1. As this is a model-transformation all ModGraph rules can be reused without any problems. Procedural and simple operations can be modeled in Xcore now. Finally the code generated from the ModGraph rules is injected into the Xcore generated code.

4. Back to the roots: Using the generated Ecore model to implement operations with ModGraph rules also starts with an Xcore model. One may model the structure and some operations in Xcore. The Xcore code generator generates an Ecore model within code generation, shown in Figure 1 as 'generated Ecore model'. This model may be used to implement operations as ModGraph rules the classical way. The difference herein is, that the ModGraph code is still injected into the Xcore generated code.

Concluding we provide a very flexible and seamless integrated approach: you may start with Ecore or Xcore and convert one into another taking the specified ModGraph rules with you in order to get an executable model.

3 Running Example

We use a bug tracker model to demonstrate the power of ModGraph and Xcore working together. The bug tracker was developed the classical way, starting with an Ecore class diagram, shown in Figure 3, refined by ModGraph graph transformation rules and then converted into an Xcore model to model the procedural part.

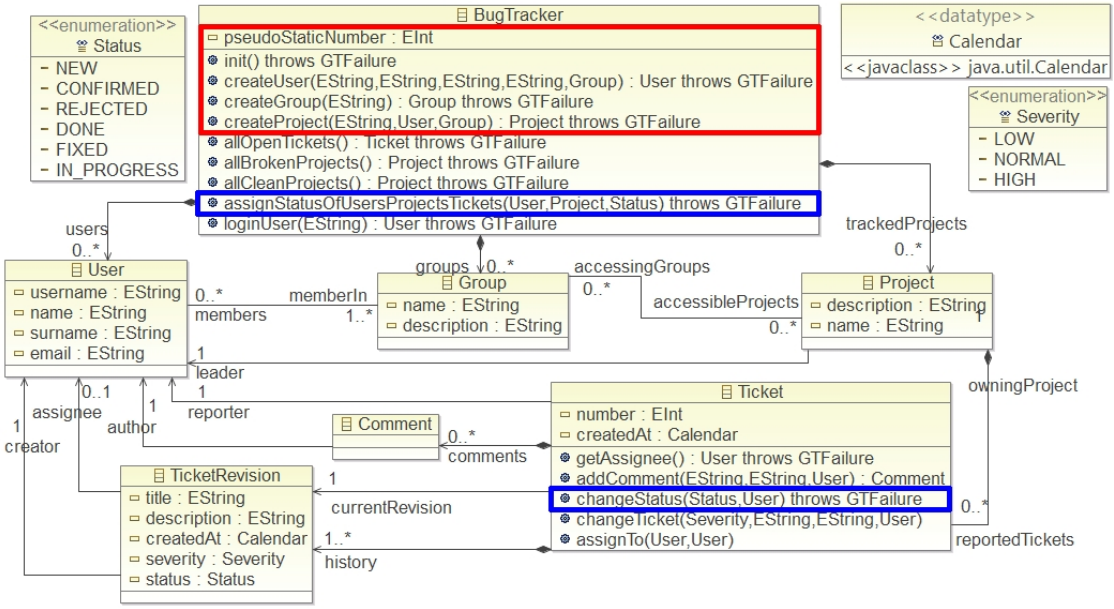


Figure 3: Running example: Bug tracker Ecore model.

The Ecore class diagram. The bug tracker tracks an arbitrary number of users and projects containing tickets. Each ticket, identified by a unique number, has a reporter of type `User` and revisions all of which are assigned to an user. Users are members of groups. Each group can access an arbitrary number of projects.

Each class modeled in the class diagram has a number of attributes and operations. In the following we are considering only the marked ones.

Xcore as control flow language. Such a bug tracker has to be initialized to be used. Though all tickets are identified by a unique number, this number has to be initialized within the bug tracker. Operations like `createGroup`, `createUser` or `createProject` may be implemented very well by graph transformation rules. Figure 4 shows the rule implementation for `createProject`. This method carries three parameters: a `leader` for the Project of type `User`, an initial group of users named `firstGroup` of type `Group` and a `String name` for the project. A precondition ensures that the project will get a proper name, not the empty string or worse null. In the graph pattern the two parameters are shown with rounded corners and the current object, an instance of the `BugTracker` class, marked with this. A new project is created and its name is assigned to the method's primitive parameter `name`. The project is marked as return parameter of the method. It is connected to its leader and the accessing group as well as to its bug tracker. This model modification is only performed if the negative application condition does not hold: There must not be a project with the same name yet. The other methods are implemented similarly.

For the initialization of the bug tracker all of these methods are needed, hence they are called as a sequence during the initialization as shown in Listing 1.

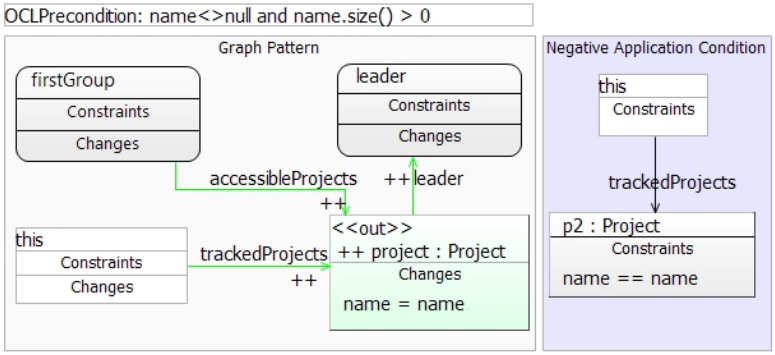


Figure 4: Graph transformation rule for `BugTracker::createProject`.

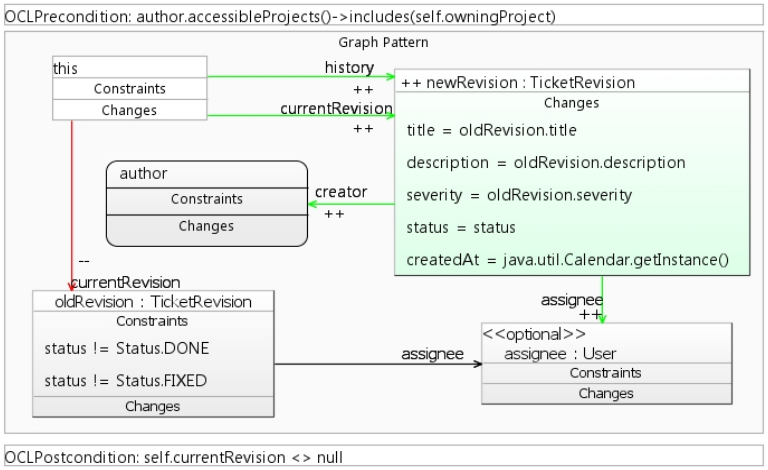


Figure 5: Graph transformation rule for `Ticket::changeStatus`.

Listing 1: Xcore model for `BugTracker::init()`.

```
class BugTracker
{   int pseudoStaticNumber
    ...
    op void init() throws GTFailure{
        pseudoStaticNumber = 0
        var group = createGroup("inititators")
        var user = createUser("firstUser",
                                "someone","","",group)
        createProject("Dummy Project", user, group)
    }
    ...
}
```

While working with such a bug tracker a number of projects may belong to the same project leader. The project leader may want to change the status of all tickets of (some of) his projects, e.g. if he does not want to continue the projects. Therefore the method `assignStatusOfUsersProjectsTickets` is implemented as shown in listing 2, which calls the ticket's method `changeStatus` implemented with a graph transformation rule (Figure 5) for each ticket.

Listing 2: Xcore model for `BugTracker::assignStatusOfUsersProjectsTickets(...)`.

```
...
    op void assignStatusOfUsersProjectsTickets(User user,
            Project[1..*] projects, Status status)
            throws GTFailure{

        for(project : projects){
            if(project.leader == user){
                for(ticket: project.reportedTickets){
                    ticket.changeStatus(status, user)
                }
            }
        }
    }
...
```

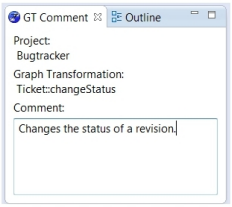
Xcore to model a simple operation. The getter-method for the initialization of the running number `pseudoStaticNumber` is overwritten as shown in Listing 3. The standard getter-method is also generated, but with different name.

Listing 3: Xcore model for `BugTracker::getPseudoStaticNumber()`.

```
    op int getPseudoStaticNumber(){
        pseudoStaticNumber = pseudoStaticNumber +1
        return pseudoStaticNumber
    }
```

Advantage of ModGraph rule. Figure 6 shows two implementations of `changeStatus` in class `Ticket`. The method creates a new ticket revision and connects it to the ticket (current object), the `author` and the `assignee`. ModGraph's rule implementation is understandable within a glimpse: graphical, color-coded, clearly structured nodes and edges visualize the actions to be performed. In Xcore the comment, pre- and postconditions have to be searched inside the code, while the rule shows them always at the same place. Using OCL, Xcore requires an explicit annotation. ModGraph includes the usage of OCL. Creating objects simply means creating a node in a ModGraph rule, while Xcore requires a whole factory call. Assigning attribute values takes the same effort in both languages. But taking into account that they are calculated from the

```
//OCL support:
@Ecore(invocationDelegates="http://www.eclipse.org/emf/2002/Ecore/OCL/Pivot“,
settingDelegates="http://www.eclipse.org/emf/2002/Ecore/OCL/Pivot",
validationDelegates="http://www.eclipse.org/emf/2002/Ecore/OCL/Pivot")
 ...
annotation "http://www.eclipse.org/emf/2002/Ecore/OCL/Pivot" as OCL
...
class Ticket{
...
Status done = 'DONE'
Status fixed = 'FIXED'
@GenModel(documentation="Changes the status of a revision.")
@OCL(pre_pre1 = "author.accessibleProjects()->includes(self.owningProject)")
@OCL(post_post1 = "self.currentRevision <> null")
op void changeStatus(Status status, User author) throws GTFailure{
    //match
    var oldRevision = this.currentRevision
    if(oldRevision.status == done && oldRevision.status == fixed){
        return
    }
    var assignee = oldRevision.assignee
    //create
    var newRevision =
        BugmodelFactory::eINSTANCE.createTicketRevision
    //set Attributes directly
    newRevision.title= oldRevision.title
    newRevision.description = oldRevision.description
    newRevision.severity = oldRevision.severity
    newRevision.status = status
    newRevision.createdAt = Calendar::getInstance
    //set Links
    if(assignee != null)
        newRevision.assignee = assignee
    newRevision.creator = author
    this.currentRevision = newRevision
    this.history.add(newRevision)
}
...
```



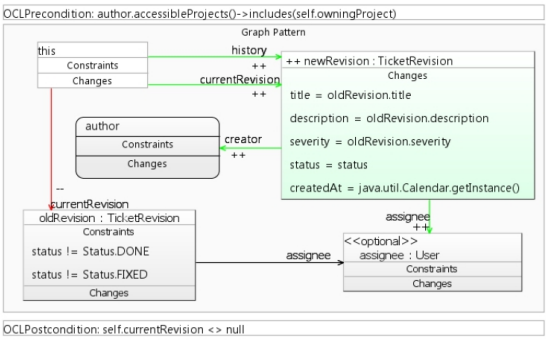


Figure 6: Comparing Xcore and ModGraph implementation.

pre-state of the model, complicates the Xcore code (not shown in the example). Linking objects means in Xcore performing a list operation or setting an attribute, while in ModGraph you may draw a simple line, without taking into account that a node may be optional. Concerning the matching of objects, Xcore forces the modeler to get objects or lists and in case of constraints, filter them manually. In a ModGraph rule the modeler simply draws the object and attaches the constraints.

4 Related Work

By combining ModGraph and Xcore, we provide an environment for model-driven software engineering which offers full-fledged support for both structural and behavioral modeling. This approach goes beyond our previous work which relied on Java for defining control structures for graph transformation rules [BWW11, BWW12b]. Please notice that we decided to reuse

language/tool	*GT rules*		*control flow*		*EMF-compliant*
	textual	*graphical*	*textual*	*graphical*	
eMOFLON [ALPS11]	-	x	-	x	x
Fujaba [Zün01]	-	x	-	x	-
GReAT [AKN+06]	-	x	-	x	-
GrGen.NET [JBK10]	x	-	x	-	-
Henshin [ABJ+10]	-	x	-	x	x
MDELab [GHS09]	-	x	-	x	x
ModGraph/Xcore	-	x	x	-	x
PROGRES [SWZ99]	-	x	x	-	-
VIATRA2 [VB07]	x	-	x	-	-

Table 1: Graph transformation languages and tools

Xcore's language for procedural modeling rather than to design a new language for this purpose. In this way, we stick to the constant goal underlying the ModGraph project: to improve EMF's modeling languages and tools precisely by the added value of graph transformations.

Furthermore, the approach presented in this paper follows the lines of [BWW12a], where we analyzed several large models based on graph transformations which were created for different application domains. Altogether, the study implies that a language for behavioral modeling should offer a mix of rule-based and procedural elements such that the modeler may select the language constructs being most appropriate for the problem at hand. Moreover, we came to the conclusion that control flow should be written in a concise textual notation offering control structures known from structured programming. On the other hand, we still consider the graphical notation of graph transformation rules more intuitive and easier to understand than a textual notation.

Table 1 compares the ModGraph/Xcore approach to several other languages/tools for graph transformations with respect to the notations used for graph transformation rules and control flows, as well as EMF compliance. The latter is satisfied if the respective language employs the Ecore metamodel for structural modeling. Only eMOFLON, Henshin, MDELab, and ModGraph/Xcore are EMF-compliant. VIATRA2 was built with EMF, but introduces a custom metamodel for structural modeling. Fujaba and GReAT are based on UML, PROGRES relies on typed and attributed graphs.

Like ModGraph, most approaches favor graphical over textual representations of graph transformation rules. In contrast, there is almost a balance of textual and graphical control flow languages. Based on our experiences with Fujaba's story diagrams, we decided to adopt a textual notation for control flow, which we consider more structured and concise.

Thus, ModGraph/Xcore combines textual control flow with graphical graph transformation rules. Among all compared tools, only PROGRES provides a hybrid notation for rule-based and procedural modeling. While PROGRES provides a native control flow language, we decided to reuse the control flow language of Xcore.

Finally, most compared languages have been designed as *layered languages* which provide control flow on top of a rule-based language for graph transformations. For example, in PRO-

GRES the elementary construct for specifying changes on a graph is a graph transformation rule. Only Fujaba and ModGraph/Xcore deviate from this paradigm. In ModGraph/Xcore, simple changes may be coded as ordinary procedural Xbase operations being composed from elementary operations for creating objects and links, assigning attribute values, etc. Fujaba's story diagrams may contain statement activities, i.e., fragments of Java code which are copied directly into the generated code. However, statement activities introduce dependencies on the underlying programming language and break the separation between modeling and programming.

5 Conclusion

Here we demonstrated the benefits of EMF, Xcore, and ModGraph interacting tightly, while separating the structural, procedural behavior, and object oriented behavior model. The "Swiss cheese" problem may be solved without loosing full integration of ModGraph into the EMF world.

Current work aims at re-targeting the ModGraph code generator to lead ModGraph to platform independence. Therefore ModGraph rules will be translated into the Xcore model via an in-place model-to-model transformation, as depicted in Figure 1 with 'transform'. Concerning pre- and postconditions as well as the creation and deletion of objects no problems are expected so far. One of the sticking points is constituted by the matching. Furthermore a graph transformation rule contains lots of implicit information, like the calculation of attribute values before changing the model. This implicit information is mapped to the generated code so far. Two possibilities have to be discussed:

- Matching in the Xcore generated code: The Xcore code generator may be adapted to match again at code level. This leads to an implicit platform dependence: each change in the Xcore generator has to be taken into account.
- Matching in a model: Create a helper model containing modeled Utility-classes all of which contain the implicit information explicitly. This entirely uncouples ModGraph from code. The whole information is stored in Xcore models. Hence, the code generation is left to Xcore.

So far the second solution seems to be more feasible at the moment. ModGraph may be entirely lifted to the modeling level, performing a model-to-model transformation from ModGraph rules to Xcore operation bodies.

Strictly following the principle "Everything is a model", a software engineer is no more concerned with programs. He or she is enabled to *total model-driven software engineering* including the benefits of graph transformation rules described above without leaving the EMF-world.

Bibliography

[ABJ+10] T. Arendt, E. Biermann, S. Jurack, C. Krause, G. Taentzer. Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations. In Petriu et al. (eds.), *Proceedings 13th International Conference on Model Driven Engineering*

Languages and Systems (MODELS 2010), Part I. Volume 6394, pp. 121–135. Oslo, Norway, Oct. 2010.
http://dx.doi.org/10.1007/978-3-642-16145-2

[AKN+06] A. Agrawal, G. Karsai, S. Neema, F. Shi, A. Vizhanyo. The Design of a Language for Model Transformations. *Software and Systems Modeling* 5:261–288, 2006.

[ALPS11] A. Anjorin, M. Lauder, S. Patzina, A. Schürr. eMoflon: Leveraging EMF and Professional CASE Tools. In *INFORMATIK 2011*. Lecture Notes in Informatics 192, p. 281. Gesellschaft für Informatik, Bonn, October 2011. extended abstract.

[BWW11] T. Buchmann, B. Westfechtel, S. Winetzhammer. MODGRAPH - A Transformation Engine for EMF Model Transformations. In *Proceedings of the 6th International Conference on Software and Data Technologies*. Pp. 212 – 219. 2011.

[BWW12a] T. Buchmann, B. Westfechtel, S. Winetzhammer. The Added Value of Programmed Graph Transformations - A Case Study from Software Configuration Management. In Schürr et al. (eds.), *Applications of Graph Transformations with Industrial Relevance*. Lecture Notes in Computer Science 7233, pp. 198–209. Springer Berlin / Heidelberg, 2012.
http://dx.doi.org/10.1007/978-3-642-34176-2_17

[BWW12b] T. Buchmann, B. Westfechtel, S. Winetzhammer. ModGraph: Graphtransformationen für EMF. In Sinz and Schürr (eds.), *Modellierung 2012*. Lecture Notes in Informatics 201, pp. 107–122. GI, Bamberg, Deutschland, March 2012.

[Ecl12] Eclipse Foundation. Xcore. 2012.
http://wiki.eclipse.org/Xcore

[EEK+12] S. Efftinge, M. Eysholdt, J. Köhnlein, S. Zarnekow, W. Hasselbring, R. von Massow, M. Hanus. Xbase: Implementing Domain-Specific Languages for Java. In *GPCE '12 Proceedings of the 11th International Conference on Generative Programming and Component Engineering*. Pp. 112–121. ACM, New York, NY, USA, 2012.

[GHS09] H. Giese, S. Hildebrandt, A. Seibel. Improved Flexibility and Scalability by Interpreting Story Diagrams. In Boronat and Heckel (eds.), *Proceedings of the 8th International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2009)*. Electronic Communications of the EASST 18. York, UK, Mar. 2009. 12 p.

[JBK10] E. Jakumeit, S. Buchwald, M. Kroll. GrGen.NET — The Expressive, Convenient and Fast Graph Rewrite System. *International Journal on Software Tools for Technology Transfer* 12:263–271, 2010.

[SBPM09] D. Steinberg, F. Budinsky, M. Paternostro, E. Merks. *EMF Eclipse Modeling Framework*. The Eclipse Series. Boston, MA, 2nd edition, 2009.

[SWZ99] A. Schürr, A. Winter, A. Zündorf. The PROGRES Approach: Language and Environment. In *Handbook of graph grammars and computing by graph transformation: vol. 2: applications, languages, and tools*. Pp. 487–550. World Scientific Publishing, 1999.

[VB07] D. Varró, A. Balogh. The model transformation language of the VIATRA2 framework. *Science of Computer Programming* 68(3):214–234, 2007. http://dx.doi.org/10.1016/j.scico.2007.05.004

[Win12] S. Winetzhammer. ModGraph - Generating Executable EMF Models. In Margaria et al. (eds.), *Proceedings of the 7th International Workshop on Graph Based Tools*. Electronic Communications of the EASST 54, pp. 32–44. EASST, Bremen, Deutschland, September 2012.

[Xte12] Xtext 2.3 Documentation. 2012. http://www.eclipse.org/Xtext/documentation.html

[Zün01] A. Zündorf. Rigorous Object Oriented Software Development. Technical report, University of Paderborn, Germany, 2001.