Electronic Communications of the EASST
Volume 54 (2012)

Proceedings of the 7th International Workshop on Graph Based Tools (GraBaTs 2012)

ModGraph - Generating Executable EMF Models

Sabine Winetzhammer

12 pages

Guest Editors: Christian Krause, Bernhard Westfechtel
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

ModGraph - Generating Executable EMF Models

Sabine Winetzhammer

sabine.winetzhammer@uni-bayreuth.de
Lehrstuhl für angewandte Informatik 1 - Software Engineering
Universität Bayreuth, Germany

Abstract: Model driven software engineering aims at creating high level executable models which may be interpreted or compiled. For efficient execution of operations on model instances code generators play an important role. A well-established tool for structural modeling and code generation is the Eclipse Modeling Framework (EMF). We extended EMF by behavior modeling within ModGraph, a tool to model behavior by graph transformation rules. Each rule corresponds to an operation modeled in the Ecore class diagram. This paper focuses on ModGraph's code generator. Therefore I describe the matching of graph transformation rules as well as the exact translation of a rule and its seamless injection into the existing EMF Java code. A running example of a simple calendar application complements the explanation.

Keywords: EMF, Code Generation, ModGraph, Model Driven Software Development, Graph Transformation

1 Introduction

Over the last decades programming languages evolved from assembler code into high level object oriented programming languages. Executable models are the next step of abstraction. These highly abstract models may be interpreted or compiled. In model driven software engineering these models are often only partially executable like it is the case within the Eclipse Modeling Framework [SBPM09] (EMF). It provides code generation from structural Ecore class diagrams. The EMF code generator creates code for classes, attributes, references and elementary operations. Elementary operations are, for example, creating links (instances of references) or setting attribute values. For user modeled operations EMF creates only empty methods, which have to be implemented by a programmer. Here ModGraph ([BWW12] and [BWW11]) steps in: ModGraph supports behavioral modeling of complex operations based upon graph transformation. An EMF model instance is considered as a graph. The graph's nodes and edges correspond to instances of classes and references modeled in the Ecore class diagram. Each rule corresponds to one user modeled operation in the Ecore class diagram. A code generator creates executable code for this rule. Hence the modeler may use graph transformations whenever they are helpful to describe complex operations on EMF model instances in a declarative and graphical way. For everything else, in particular for the control flow, the modeler may resort to Java.

Here I present details on ModGraph's code generation, whereas the focus in [BWW11] and [BWW12] was set on the presentation and the meta model of the graphical tool. Section 2 provides an overview on ModGraph's modeling capabilities, Section 3 introduces the running

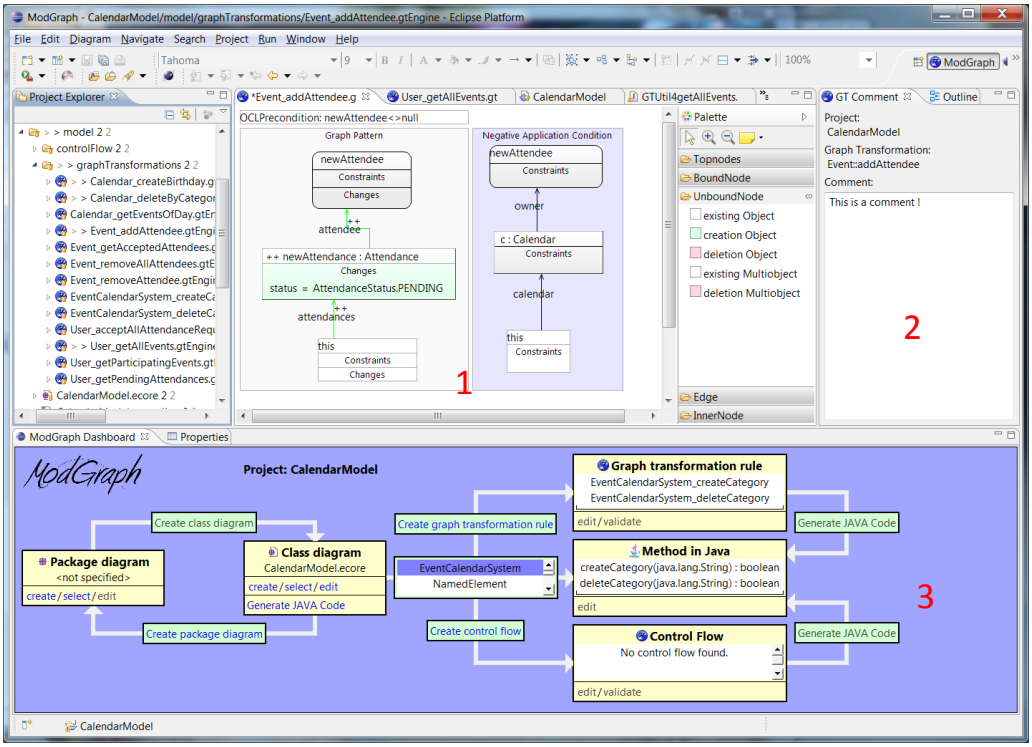

Figure 1: ModGraph perspective

example. Section 4 explains the code generator. It is illustrated with code snippets generated from the running example. Section 5 discusses related work, Section 6 concludes the paper.

2 Overview

Within ModGraph the body of a user-defined operation in an Ecore class diagram can be modeled using a graph transformation rule. A detailed description of the meta model is given in [BWW12]. The GMF-based graphical environment in use is shown in Figure 1. Next to the Eclipse package explorer on the left the main window (marked as 1) shows the graph transformation rule. A comment can be attached to each rule using the comment view (marked as 2). A dashboard (marked as 3) provides a structured view on the process toward the executable model. All rules shown here are snapshots of the main window.

The rule itself must contain a graph pattern that specifies the subgraph to be searched inside the Ecore model instance as well as the changes to be performed. To specify these changes different kinds of nodes and edges representing instances of classes and references in the Ecore model are available.

The current object corresponds to the Java `this` object (see Figure 1: this). ModGraph distinguishes between unbound and bound nodes. Unbound nodes represent objects. Bound

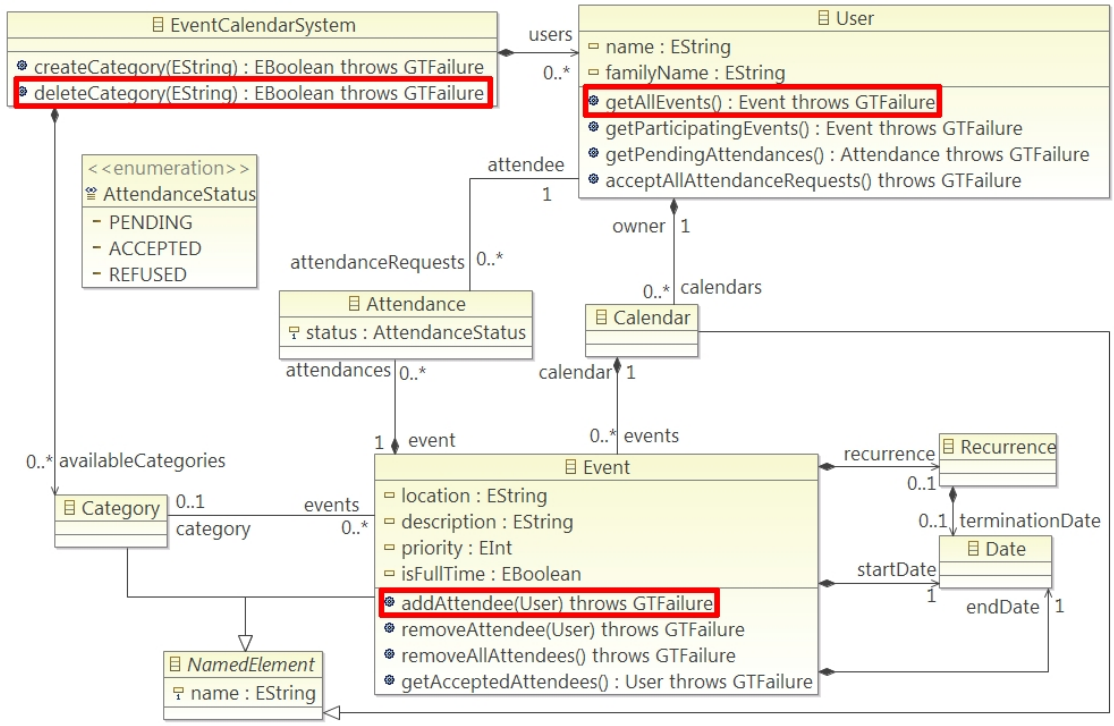

Figure 2: Ecore model of the running example, the event calendar system

nodes represent the non-primitive parameters of the operation. Primitive parameters are used to model constraints on objects or are employed as attribute values. If the current object is absent, at least one bound node is needed. Multi-valued parameters are shown as multi-parameter nodes, single-valued parameters as parameter nodes (see Figure 3), both with rounded corners. Any rule may contain an arbitrary number of unbound nodes represented by objects (see Figure 4) and multi-objects (see Figure 5). They need to be searched by pattern matching in the model instance. Unbound nodes must reference a class in the Ecore model. Bound nodes may reference one.

All nodes in the graph pattern – except for the current object – provide a status which is create (++), delete (−−) or preserve (no marker). The status indicates the changes to be performed on the model instance. Nodes may be marked as return parameter (<<out>>) or as optional (<<optional>>).

In general nodes may be connected by edges: links and paths. The connection of two multi-valued nodes constitutes an exception, due to an ambiguous assignment. If a node is optional, the related edges are assumed to be optional, too. Links represent an instance of a reference in the Ecore class diagram and provide a status. Multi-valued links may be ordered. Paths represent derived references. They are marked with a path expression written in OCL (see Figure 5) or Java.

Except for created ones, any node can be constrained by an OCL or attribute condition. On created or preserved nodes attribute values may be changed and operations may be called. These

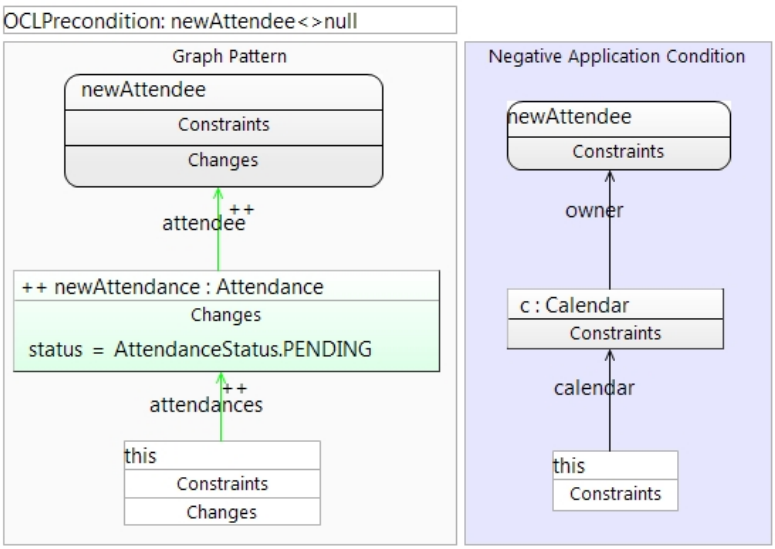

Figure 3: Rule for `Event:addAttendee(newAttendee:User)`

operations are executed after the successful application of the rule, simulating a sequence.

The rule's graph pattern can be constrained using pre- and postconditions as well as a negative application condition (see Figure 3). Pre- and postconditions are formulated in OCL. A negative application condition is represented as a graph showing a pattern that must not occur, using only single-valued nodes. A special single-valued node is a NACBoundNode (see Figure 4) which references an unbound node in the graph pattern. NACBoundNodes are bound because the unbound nodes in the graph pattern are bound before the negative application condition is checked. Nodes in a negative application condition can be analogously constrained and connected by links (without status) and paths.

Each rule is validated in a threefold way. A rule must conform to the underlying ModGraph metamodel. The editor performs live validation concerning the user model during editing. The remaining errors, for example, verifying the reachability of nodes, are caught by a batch validation.

3 Running Example

A running example, an event calendar, is used to show the rules' syntax as well as the generated code. Note that the example was built for demonstation purposes. Figure 2 shows the event calendar Ecore model. Each calendar system supports multiple users. Each user may own an arbitrary number of calendars that contain events. Each event may belong to a category that belongs to the event calendar system. There may be attendances to an event. Users are attendees. An event has a defined start and end date. It may reoccur. In the following I explain the implementations of the methods marked in Figure 2.

The implementation of `addAttendee` is shown in Figure 3. At the top of the rule the pre-

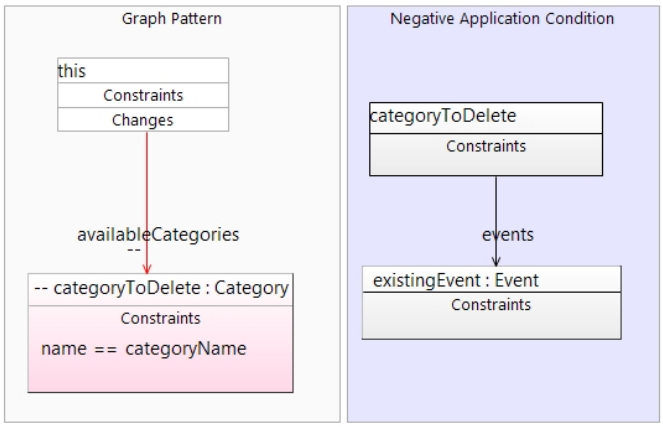

Figure 4: Rule for `EventCalendarSystem:deleteCategory(categoryName:String)`

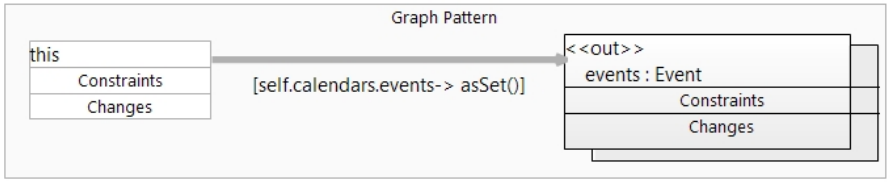

Figure 5: Rule for `User:getAllEvents()`

condition is shown. The OCL expression assures the existence of the parameter `newAttendee`. On the left the graph pattern is shown. It contains the non-primitive parameter `newAttendee` at the top and the current object at the bottom. In the middle an attendance `newAttendance` is to be created, and therefore shown as an unbound creation object. Its attendance status is set to pending. The new attendance is connected to the parameter and the current object by using two creation links. On the right a negative application condition shows what must not occur in the model instance in order to create a new attendance: the user, given by the parameter `newAttendee` must not be the owner of the calendar the event belongs to.

The second rule deleteCategory (Figure 4) consists of a graph pattern and an NAC. The pattern specifies that a category in the event calendar system is deleted if the name of the category equals the value of the method's primitive parameter `categoryName`. This may only happen if the negative application condition does not hold. Therefore the category to be deleted, shown as a NACboundNode, may not contain any event anymore.

The last rule getAllEvents (Figure 5) shows just a graph pattern. The current object inside the pattern is an instance of the `User` class. A path connects it to a multi-object of type `Event`. The path specifies that all events of all calendars the user owns are collected.

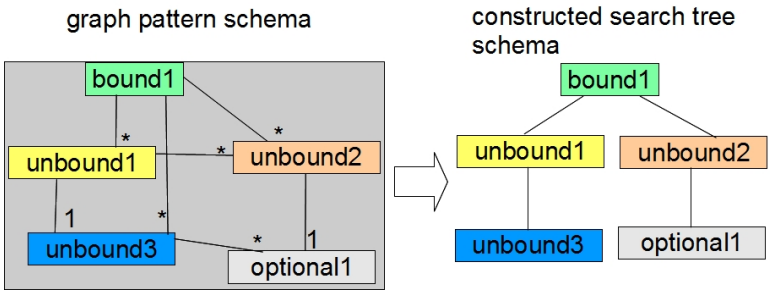

Figure 6: Logic of ModGraph's pattern matching algorithm

4 Generating an Executable Model

This chapter provides an overview about the ModGraph code generator. The generator follows a compiler solution and is written using Xpand templates. For illustration code generated from the example rules as well as an abstract example is used.

4.1 Preliminaries for the Generation of Pattern Matching Code

Before code generation starts the rule has to be parsed. A recursive, heuristic greedy algorithm is used to transform any graph given in the rule into a forest of spanning trees following these steps: (1) Use each bound node in the pattern as the root of a tree inside the forest. (2) Consider all outgoing edges of the forest's nodes. Choose the one instantiating the reference with minimum multiplicity. (Paths are considered like multiplicity many references. If two links instantiating references of the same multiplicity exist, one is chosen randomly.) (3) Check, if the node the link targets is mandatory and not contained in any tree yet. If true, add it as a child into the tree that contains the link's source. (4) Repeat steps (2) and (3) until there are no more mandatory nodes. (5) Repeat steps (2) and (3) – without the mandatory check – for all optional nodes. Please note that this prohibits searching a mandatory object from an optional one.

The forest specifies the reachability of nodes inside the graph pattern. It acts as a search plan. If any mandatory node cannot be inserted into a tree it cannot be bound and the matching fails.

To visualize these steps a schema of the search tree constructed successfully from a schematic graph pattern is shown in figure 6. The pattern has also been transformed into a schema assuming that all nodes are either to preserve or to delete. Some multiplicities are added for better understanding. The only bound node bound1 is used as a root (step(1)). unbound1 and unbound2 are added as children (steps (2) & (3)). The node unbound3 in figure 6 may be searched from bound1 and unbound1. In favor of the lower multiplicity, it is added as unbound1's child in the tree (step (4), repetition of (2) and (3)). The optional node is added as a leaf, since only optional nodes may be searched from it and the rule contains no more such nodes (step (5)).

Concretely: For the rule addAttendee the situation is shown in figure 7. For the graph pattern two roots are created, without children because the pattern only contains bound nodes and one node to be created. The NAC contains an unbound object. It may be searched either from the parameter of type user or the current object of type `Event`. The Ecore model specifies that an

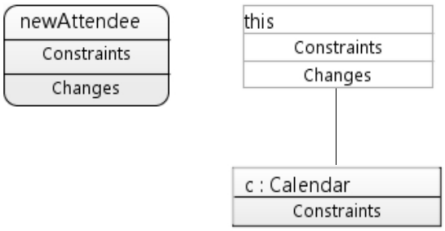

Figure 7: Spanning forest for the NAC in rule `Event:addAttendee` in Figure 3

event belongs to exactly one calendar, but a user may own several calendars. So the calendar is bound through the current object of type `Event`.

This forest is used to create the utility methods shown in the next section.

4.2 Code Generation

ModGraph injects the generated code into the EMF generated one, consequently the EMF code has to be generated independently in the first step.

The code generation itself starts by modifying the EMF generated interface. ModGraph methods may throw exceptions of type `GTFailure`, for example, if the postcondition is not matched. Through the generated interface has to be supplemented by an exception.

Next, the code generated for the implementation class is injected into the empty EMF generated method body. This is done by generating it separately and injecting it using the abstract syntax tree.

The code generated for the method `addAttendee` is shown in Listing 1 and structured in the following way: (1) The comment, written in the comment field, marked with 2 in Figure 1, is transformed into a JavaDoc-Comment as shown at Listing 1, lines 1-3. (2) As described for the interfaces the method's header is extended by an exception (Listing 1, lines 4–5). (3) The precondition is checked using an `OCLHelper` class based upon Eclipse OCL (Listing 1, lines 6–8). (4) A utility class is instantiated and called to perform the matching and to check all conditions (Listing 1, lines 9–10). (5) All attribute values are calculated before anything is changed (listing 1, lines 11–12). (6) New objects are created, existing ones are changed or deleted. In listing 1, lines 13–15, the creation of the new attendance object is performed. The deletion of an object – not performed on the sample rule – is delegated to the utility class, see Listing 2. Its delete method is overloaded due to the possible deletion of a multi-object or multi-parameter. (7) Links are created or deleted. Listing 1, lines 16–17, shows the creation of the two links in figure 3.

Listing 1: Implementation of Method `Event:addAttendee(newAttendee:User)`

```
1 /** <!-- begin-user-doc --> This is a comment!
2  * <!-- end-user-doc -->
3  * @generated NOT modified by ModGraph */
4 public void addAttendee(User newAttendee)
5             throws  GTFailureImpl {
```

```
6       OCLHandler.evaluatePrecondition("newAttendee<>null", this,
7           new String[] { "newAttendee" },
8                   new Object[] { newAttendee });
9       GTUtil4addAttendee util = new GTUtil4addAttendee();
10      util.match(newAttendee, this);
11      final AttendanceStatus newAttendanceStatusValue =
12          AttendanceStatus.PENDING;
13      Attendance newAttendance = CalendarPackage
14          .CalendarPackageFactory.eINSTANCE.createAttendance();
15      newAttendance.setStatus(newAttendanceStatusValue);
16      this.getAttendances().add(newAttendance);
17      newAttendance.setAttendee(newAttendee);
18  }
```

Listing 2: Excerpt of the implementation and utility of Method `EventCalendarSystem: deleteCategory(categoryName:String)`

```
1       public boolean deleteCategory(String categoryName)
2           throws GTFailureImpl {
3           ...
4           util.delete(categoryToDelete);
5           ...
6       }
7
8   public class GTMatchingUtil {
9       public void delete(EObject eObject) {
10          EcoreUtil.delete(eObject, true);
11      }
12      public void delete(EList<?> objectList) {
13          checkResource();
14          for (Object o : objectList) {
15              if (o instanceof EObject)
16                  EcoreUtil.delete((EObject) o, true);
17          }
18      }
```

The utility class performs the matching on the instance graph. Referring to the schematic tree in Figure 6, the utility class matches this tree against an existing ecore model instance as shown in Figure 8. The result of the matching is also shown in Figure 8 using green connectors. The mandatory unbound objects are matched as you would expect. unbound1 is matched to the object possible unbound1 (2) with an unbound3 candidate reachable which matches here. If unbound1 is matched to the other possible candidate and then it is recognized that the matching of unbound3 is impossible, the match is reverted. unbound2 can be matched on two nodes in the Ecore model instance. The result shown here is reached assuming that possible unbound2 (1) does not fit the matching criteria. It is no problem that the optional node is not matched.

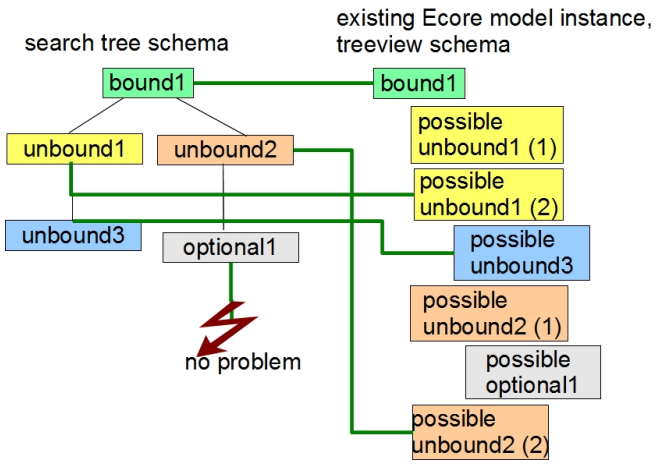

Figure 8: Result of the matching on an existing example Ecore model instance (schema)

Concretely: The utility class extends the ModGraph built in class `GTMatchingUtil`. The ModGraph code generator generates a coordinating match method, following the logic described above. This method carries the same parameters as the operation itself and one for the method's current object.

For the Method `addAttendee` all parameters are put into a global map in order to be accessible from anywhere (see Listing 3, lines 5–6). Next the bound nodes as well as the current object in the graph pattern are checked concerning their constraints and the links between them. Then a match of the unbound nodes in the graph pattern with the Ecore model instance is performed. Please note that both these steps need not to be performed for the sample rule. The graph in the NAC is matched analogously.

The matching methods for the running example are shown in Listing 3 for the NAC matching. The method `nacMatched` (Listing 3, lines 9–12) coordinates the matching of the NAC's elements. In the example code the object `c` of type `Calendar` is matched starting at the current object. If the match of the NAC succeeds, an exception is thrown as shown in lines 13–22. The matching itself is performed in another method shown in lines 23–33. Here the constraints given in the rule are checked.

Listing 3: Utility class generated for Method `Event:addAttendee(newAttendee:User)`

```
1  public class GTUtil4addAttendee extends GTMatchingUtil {
2
3      public void match(User newAttendee, Event event)
4              throws GTFailureImpl {
5          map.put("newAttendee", newAttendee);
6          map.put("event", event);
7          nacMatched(newAttendee, event);
8      }
9      private void nacMatched(User newAttendee, Event event)
10             throws GTFailureImpl {
```

```
11          matchUnboundNodesInNAC("c", event);
12      }
13      private void matchUnboundNodesInNAC(String searchedNodeName,
14              EObject boundPredecessor) throws GTFailureImpl {
15          if (searchedNodeName.equals("c")) {
16              Calendar c = getC4AddAttendee((Event) boundPredecessor);
17              if (c != null) {
18                  alreadyConsideredObjects.add(c);
19                  throw new GTFailureImpl("c  found.");
20              }
21          }
22      }
23      public Calendar getC4AddAttendee(Event event)
24              throws GTFailureImpl {
25          Calendar aCalendar = (Calendar) event.getCalendar();
26          if (!(alreadyConsideredObjects.contains(aCalendar))
27              && aCalendar instanceof Calendar
28              && ((User) map.get("newAttendee")).getCalendars()
29              .contains(aCalendar))
30              return (Calendar) aCalendar;
31          else
32              return null;
33      }
34  }
```

Matching via a path is performed by the OCL-Handler, as shown in Listing 4. Here the object is matched by calculating the OCL-expression and checking the constraints - not shown here - afterward.

Listing 4: Utility class generated for Method `Event:addAttendee(newAttendee:User)`

```
1       public EList<Event> getEvents4GetAllEvents(User user)
2               throws GTFailureImpl {
3           List<EObject> allEvents = OCLHandler.evaluatePath(user,
4                   "self.calendars.events->asSet()");
5               ...
6                   return (EList<Event>) events;
7               ...
8       }
```

5 Related Work

During the last decades researchers presented several approaches concerning model transformation languages [CH06]. Only a few of them are EMF related, based on graph transformations [EEKR99], generate executable code, perform endogenous model transformations and provide

a graphical environment as ModGraph does. Any tool not satisfying all these constrains is not considered due to space limitations.

EMF Tiger [BEK+06] and its sequel Henshin [ABJ+10], as ModGraph, use Ecore for structural modeling. Tiger lacks some constructs, ModGraph provides, like pre- and postconditions. Concerning code generation, Henshin just provides an interpreter. This means that embedding the rules into applications depends mostly on the capabilities of the interpreter. Tiger compiles each graph transformation rule into one independent class instead of a method. If one wants to use this code in a user defined operation, some hand-coding into the EMF-generated code is required: instantiate the class and pass the necessary parameters to execute the rule. This leads to higher run times and programming efforts. ModGraph generates code right into the EMF generated one. It creates a standard Java method which can be used without further modification. ModGraph provides within that an efficient and seamless integration into the generated code.

Fujaba [Zün01] uses class diagrams for structured and story diagrams for behavior modeling. A story diagram represents the implementation of a method modeled in the class diagram. Storypatterns in a story diagram specify graph transformation rules. Fujaba itself was developed outside EMF. An EMF code generator is available, but the modeler has to specify a class diagram in Fujaba first and may not immediately define an Ecore model or reuse an existing Ecore model.

The idea of story diagrams was partially reimplemented in MDELab [GHS09]. MDELab uses Ecore class diagrams for static and story diagrams for behavior modeling. But story diagrams are interpreted, not compiled. This reduces the usability to the capabilities of the interpreter in working with model instances.

eMoflon [ALPS11] may use Ecore for structural modeling and may generate EMF compliant code as well as it extends professional case tools. As Fujaba and MDELab, it uses the concept of modeling behavior and control flow into one diagram, whereas ModGraph strictly separates them, in order to get clearly laid out rules. Concerning code generation eMoflon is uses the Moca Framework, which builds up a complex tree. Furthermore a separate parser is needed to generate code. ModGraph simply uses XPand templates and a simple, straight forward constructed tree for each rule.

Compared with other tools I summarize that ModGraph is the only lightweight, EMF-based tool providing seamless integration into the EMF generated code and a clear separation of behavior and control flow modeling. The generated code may be used for all kinds of applications which assume generated EMF code, e.g., GMF (for building a graphical editor).

6 Conclusion and Future Work

Here I introduced ModGraph's code generator. ModGraph is a tool to add behavior modeling to EMF at a high level of abstraction. Therefore graph transformation rules are in use. The generator is able to generate code based upon these rules. It follows a compiler solution and injects the code seamlessly into the EMF generated code. In the process each rule is treated as the implementation of a method. The result is a fully functional executable model.

Current and future work aim at extending the graph transformation rules by a control flow language, as you can see in the Dashboard (figure 1, marked with 3). This language will provide a possibility to model a control flow for the graph transformation rules.

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

[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.

[BEK+06] E. Biermann, K. Ehrig, C. Köhler, G. Kuhns, G. Taentzer, E. Weiss. Graphical Definition of In-Place Transformations in the Eclipse Modeling Framework. In Nierstrasz et al. (eds.), *Proceedings 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2006)*. Volume 4199, pp. 425–439. Genova, Italy, Oct. 2006.
http://dx.doi.org/10.1007/11880240_30

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

[BWW12] 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.

[CH06] K. Czarnecki, S. Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal* 45(3):621–646, 2006.
http://dx.doi.org/10.1147/sj.453.0621

[EEKR99] H. Ehrig, G. Engels, H.-J. Kreowski, G. Rozenberg (eds.). *Handbook on Graph Grammars and Computing by Graph Transformation: Applications, Languages, and Tools*. Volume 2. World Scientific, Singapore, 1999.

[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.

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

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