Electronic Communications of the EASST
Volume 32 (2010)

Proceedings of the
Fourth International Workshop on
Graph-Based Tools
(GraBaTs 2010)

Visual Modeling of Controlled EMF Model Transformation using HENSHIN

Enrico Biermann, Claudia Ermel, Johann Schmidt and Angeline Warning

13 pages

Guest Editors: Juan de Lara, Daniel Varro
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Visual Modeling of Controlled EMF Model Transformation using HENSHIN

Enrico Biermann, Claudia Ermel, Johann Schmidt and Angeline Warning

Institut für Softwaretechnik und Theoretische Informatik
Technische Universität Berlin, Germany
henshin@tfs.cs.tu-berlin.de

Abstract: The tool HENSHIN is an Eclipse plug-in supporting visual modeling and execution of rule-based EMF model transformations. This paper describes the recent extensions of HENSHIN by control structures for controlled rule applications. The control structures comprise well-known imperative structures like sequences and conditions on rule applications. Moreover, application conditions for individual rules may now be arbitrarily nested and combined by logical connectors. We present the extension of the visual EMF model transformation environment HENSHIN to edit and perform controlled EMF model transformations along an example modeling a reactive Web service-based application (personal mobility manager).

Keywords: EMF, model transformation tool, graph transformation, Henshin

1 Introduction

Transformations are key modeling artefacts in model driven development. In graph transformation approaches and tools, rules express basic transformation steps. The application of rules may be controlled implicitly like in AGG [AGG09], i.e. by a fixed strategy such as "apply rules in arbitrary order as long as possible" and by providing negative application conditions for rules. Alternatively, control strategies may be defined explicitly like in Fujaba [FNTZ00], where an activity diagram (story diagram) defines loops or conditions on rule applications. Explicit control structures raise the expressiveness of transformation systems since they provide means to regulate the transformation process without having to introduce helper structures into the rules.

In this paper, we lift implicit and explicit control structures from graph transformation to EMF model transformation and introduce an extension of our recently developed tool HENSHIN[1] by visual editors for control structures. HENSHIN is an Eclipse plug-in supporting visual modeling and execution of EMF model transformations, i.e. transformations of models conforming to a meta-model given in the EMF `Ecore` format[2]. The transformation approach we use in our tool is based on graph transformation concepts which are lifted to EMF model transformation by also taking containment relations in meta-models into account [ABJ+10].

Applying EMF model transformation rules in HENSHIN changes a model *in-place*, i.e. the model is modified directly. Note that we speak of *EMF model transformation* in a general sense,

[1] http://www.eclipse.org/modeling/emft/henshin/, originating from EMF TIGER [EMT09, BEK+06, BEL+10]

[2] Note that we use the terms *meta-model* and *model* in this paper, which are called *EMF model* and *model instance* in the EMF documentation, respectively.

comprising not only source-to-target model-to-model transformations but also model refactorings or simulation of the system's behavior[3]. The HENSHIN transformation engine provides classes that can freely be integrated into existing Java projects relying on EMF.

Figure 1 shows the basic GUI of our HENSHIN tool before the extensions presented in this paper. The tree view $\boxed{1}$ allows the modeler to import EMF `EPackages` containing the basic meta-model(s) defining the domain of the transformation. The initial model is edited in a visual editor $\boxed{2}$. In the rule editor $\boxed{3}$, transformation rules can be created by editing a rule's left-hand side (LHS, the pre-condition) and right-hand side (RHS, the post-condition). The rule in Figure 1 defines an operation which adds a `Request` object and links it to existing `Departure` and a `Destination` objects. The property view $\boxed{4}$ shows additional information for selected objects. Note that all information edited using the editors in $\boxed{2}$, $\boxed{3}$ and $\boxed{4}$ can also be obtained via the tree view $\boxed{1}$.

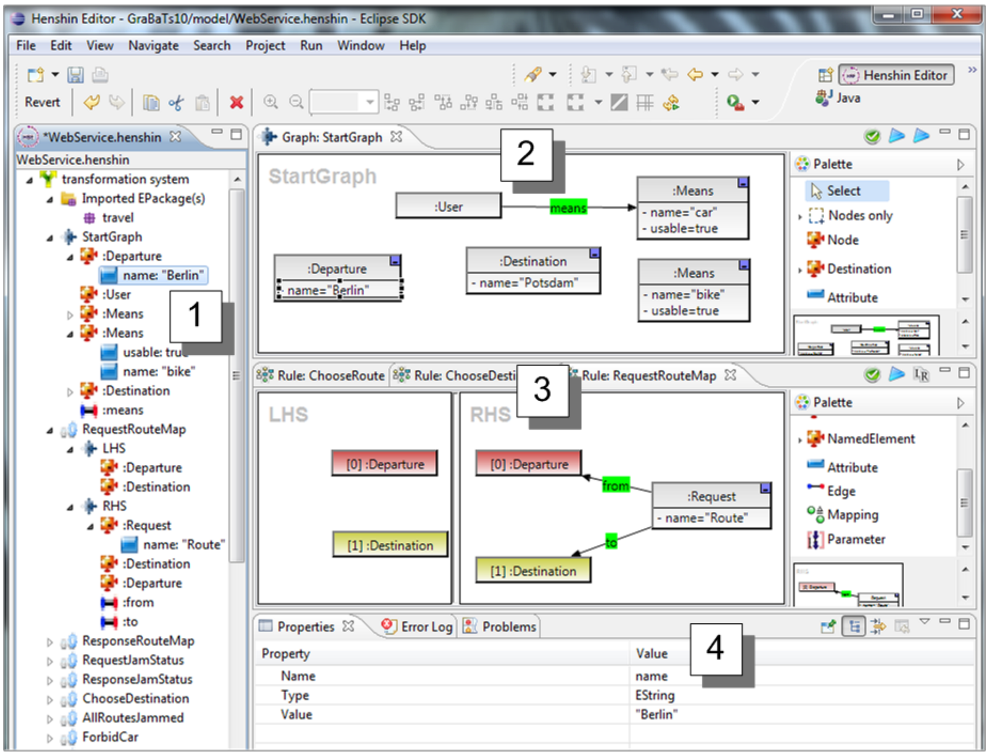

Figure 1: HENSHIN GUI with visual editors for graphs and rules.

The rule shown in $\boxed{3}$ can now be applied to the current model in $\boxed{2}$ leading to the transformed graph shown in Figure 2, where a `Request` object has now been created and linked to the `Departure` object named "Berlin" and the `Destination` object named "Potsdam". The layout of newly added object is computed automatically but may be adjusted by the user.

Currently there exist two implementations of the transformation engine. One is written in Java while the other translates the transformation rules to AGG [AGG09]. This is useful for

[3] like in our running example, the simulation of a personal mobility manager based on a web service.

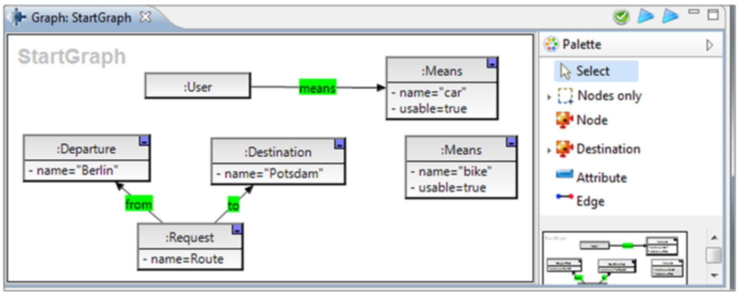

Figure 2: Transformed graph after applying rule *RequestRouteMap*

validation of consistent EMF model transformations which behave like algebraic graph transformations [BET08], e.g. to show functional behavior and correctness.

In this paper we describe the recent extension of HENSHIN supporting the use of the control structures (called HENSHIN *transformation units*), e.g. constructs for non-deterministic rule choices, rule sequences or conditional rule applications. Those constructs may be nested to define more complex control structures. Passing of model elements as parameters from one unit to another is also possible. Apart from control units defined over sets of rules, we now also support the graphical definition of application conditions for individual rules. These are application conditions in the sense of [HP09] allowing for arbitrary nesting. Several application conditions can be combined by logical connectors.

The paper is structured as follows: in Section 2, the basic concepts of graph and EMF transformation are reviewed. Section 3 presents our running example, the simulation of a personal mobility manager based on a web service. Modeling this example, we made extensive use of transformation units and application conditions which are introduced in Section 4 and Section 5, respectively. Section 6 provides an overview of related approaches and tools in comparison to our tool, and Section 7 concludes the paper with an outlook to future work.

2 EMF Model Transformation based on Graph Transformation

In this section, we introduce the main notions of modeling by algebraic graph transformation [EEPT06] (Subsection 2.1) and relate these notions to EMF modeling terms (Subsection 2.2).

2.1 Typed Attributed Graphs and Graph Transformation

A domain-specific visual language (DSVL) is modeled by a *type graph* defining the underlying visual alphabet, i.e. the symbols (node types) and edge types which are available. Sentences or diagrams of the DSVL are given by graphs typed over (i.e. conforming to) the type graph. Node types may be attributed by attribute types.

The main idea of graph transformation is the rule-based modification of graphs where each application of a graph transformation rule leads to a new transformed graph. The core of a graph transformation rule $(LHS \xrightarrow{r} RHS)$ is a pair of graphs (LHS, RHS), called left-hand side and right-hand side, and an injective (partial) graph morphism $r : LHS \to RHS$. A graph morphism consists of structure-preserving mappings from nodes in LHS to nodes in RHS, such that for an edge from node n_1 to node n_2 in LHS which is preserved by the rule, we have a corresponding edge from node $r(n_1)$ to $r(n_2)$ in RHS. In our approach, all graph morphisms are injective, i.e. they do not merge elements. Applying the rule $(LHS \xrightarrow{r} RHS)$ means to find a match of LHS in the source graph and to replace this matched part in the source graph by the corresponding RHS, thus transforming the source graph into the target graph (this step is called a *direct graph transformation*). Intuitively, the application of rule r to graph G via a match m from LHS to G deletes the image $m(LHS)$ from G and replaces it by a copy of the right-hand side $m^*(RHS)$. Note that a rule may only be applied if the so-called *gluing condition* is satisfied, i.e. the deletion step must not leave *dangling edges*.

Definition 1 (Graph Transformation) Let $(LHS \xrightarrow{r} RHS)$ be a typed graph transformation rule and G a typed graph with a typed graph morphism $LHS \xrightarrow{m} G$, called match.
A *direct graph transformation* $G \overset{r,m}{\Longrightarrow} H$ from G to a typed graph H via rule r, match m, and co-match m^* is shown in the diagram to the right. A sequence $G_0 \Rightarrow G_1 \Rightarrow .. \Rightarrow G_n$ of direct graph transformations is called *graph transformation*, denoted as $G_0 \overset{*}{\Rightarrow} G_n$.

$$\begin{array}{ccc} LHS & \xrightarrow{r} & RHS \\ {\scriptstyle m}\downarrow & & \downarrow{\scriptstyle m^*} \\ G & \longrightarrow & H \end{array}$$

A rule may be extended by input parameters, i.e. variables used to compute new attribute values for nodes in the right-hand side. When the rule is applied, the input parameters have to be bound to concrete values (either by the match or by user input).

2.2 Typed Attributed Graphs versus EMF Modeling

The Eclipse Modeling Framework EMF [EMF08] is a modeling and code generation facility for building tools and other applications based on a structured data model. Based on a meta-model, EMF provides tools and runtime support to produce a set of Java classes for the meta-model, a set of adapter classes that enable viewing and command-based editing of models conforming to the meta-model, and a basic (tree-based) editor. EMF provides the foundation for interoperability with other EMF-based tools, e.g. OCL checkers.

The conceptual similarities of modeling based on typed, attributed graphs and object-based modeling as performed by EMF are shown in Table 1.

Table 1: Mapping EMF notions to graph terminology

EMF notion	**Graph term**
EMF model	Type graph with attribution, inheritance, multiplicities. Edges can be marked as containments.
Instance model	Typed, attributed graph with containment edges
Class	Node in type graph
Object	Node in typed graph
Association	Edge in type graph (with possible multiplicities or containment mark)
Reference	Edge in typed graph that satisfies multiplicity and containment constraints.

Classes in an EMF model (i.e. the meta-model) correspond to nodes in a type graph. Associations between classes can be seen as edges in a type graph. Generalizations and multiplicity constraints of association ends can also be defined in the type graph. Objects as instantiations of classes of an EMF model are comparable to nodes in a graph which is typed by a type graph. Objects can be linked to each other by setting reference values. Such references correspond to edges in a typed attributed graph.

3 Example: Personal Mobility Manager

As running example, we specify and simulate the operational behavior of a Personal Mobility Manager (PMM), a reactive service-based application designed to satisfy requirements related to individual user mobility [LMEP08]. The aim of the system is to help the user finding an adequate route from a departure place to a destination and to propose an adequate means of transportation (either car or bike) by taking the current traffic intensity into account. We model the control flow of messages that are exchanged between the user, the PMM and corresponding Web service. To keep things simple, we do not model the actual web service here but simulate its responses by suitable variable assignments.

The modeling domain is specified as meta-model, shown in Figure 3. We have model elements for a user, his departure and destination locations, the means of transport, and requests sent to web service. A `Route` element contains a route given as response by the mobility web service, and a `JamStatus` element contains the response returned by the web service concerning the traffic on a given route.

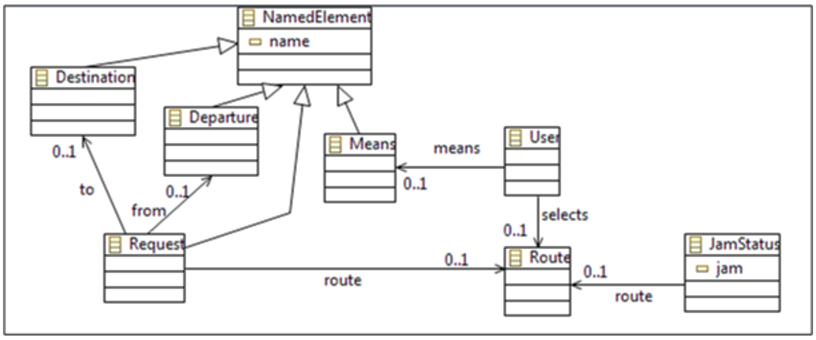

Figure 3: Meta-model for the Personal Mobility Manager

Basic PMM actions are modeled by EMF model transformation rules, shown in Figure 4.

Rule `ChooseDestination` creates a `Destination` object where the name of the destination is an input parameter; rules `RequestRouteMap` and `ResponseRouteMap` realize the creation of a route (modeled by a `Route` object) via a web service call. Having called this web service more than once, one of the returned routes is chosen by the user in rule `ChooseRoute`. For a given route, the web service is used by rules `RequestJamStatus` and `Response-JamStatus` to get information about the current traffic situation on this route. Depending on the information obtained by the web service (and coded in the `JamStatus` node), the means of

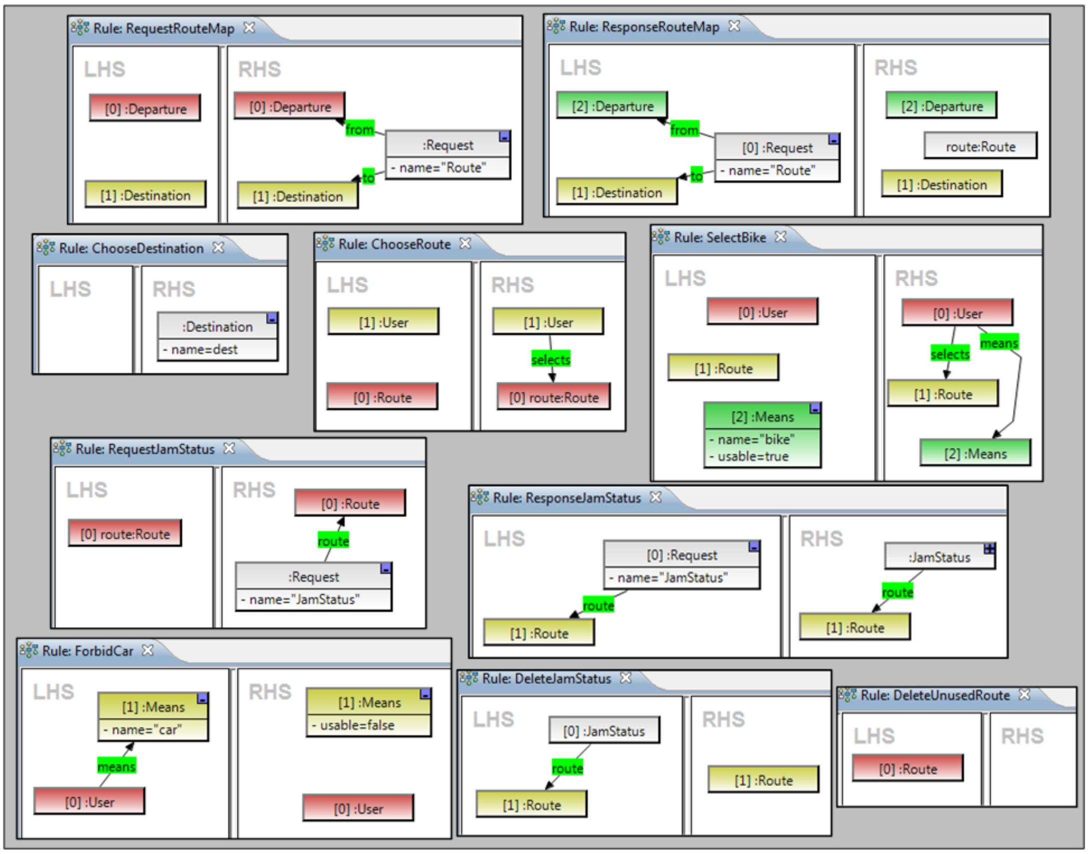

Figure 4: EMF model transformation rules for the Personal Mobility Manager

transport can be changed from the default means ”car” (as presented in the start graph in Figure 1) to the alternative means of transport ”bike”. This is realized by applying rules `ForbidCar` and `SelectBike`. At last, the information about traffic (`JamStatus` node) and possible alternative routes which have not been chosen, are deleted using rules `DeleteJamStatus` and `DeleteUnusedRoute`.

In the next section, we explain the use of HENSHIN transformation units to encapsulate and control the order of rule applications.

4 HENSHIN Transformation Units

HENSHIN transformation units may be arbitrarily nested inside each other. The most basic unit is a transformation rule. A HENSHIN transformation unit may be of type *IndependentUnit* (all subunits are applied in arbitrary order), *SequentialUnit* (all subunits are applied sequentially in a given order), *CountedUnit* (its subunit is applied a given number of times), *ConditionalUnit* (its subunits are applied depending on the evaluation of a given condition unit), and *PriorityUnit* (the applicable subunit with the highest priority is applied next). A unit is applicable (and returns `true`) if it can be successfully executed. *PriorityUnits* and *IndependentUnits* are always applicable, while *SequentialUnits* (*CountedUnits*) are applicable only if all subunits are applicable in the given order (the given number of times). A *ConditionalUnit* is applicable if either the *then*-subunit (in case the condition is `true`) or the *else*-subunit (in case the condition is `false`) are applicable.

HENSHIN transformation units may be defined in the tree view or, alternatively, in a visual editor. The tree view shows all transformation units and their nesting hierarchy (see Figure 5). The visual editor for one unit shows the unit in a left view and one selected subunit in a right view. Unit and subunit may share parameters indicated by the coloring of the parameter fields (see Figure 5, where editors for unit `mainUnit` and unit `trafficWS` are opened in parallel). A transformation unit view shows the unit's name as header, a checkbox *Activated* which the user may select/deselect to indicate whether this unit is active (will be considered while executing), a set of parameters shown as boxes in the left column, and the names and kinds of its subunits in the right column. Arrows from (to) parameter boxes to (from) subunits indicate which parameters are input (output) of which subunit.

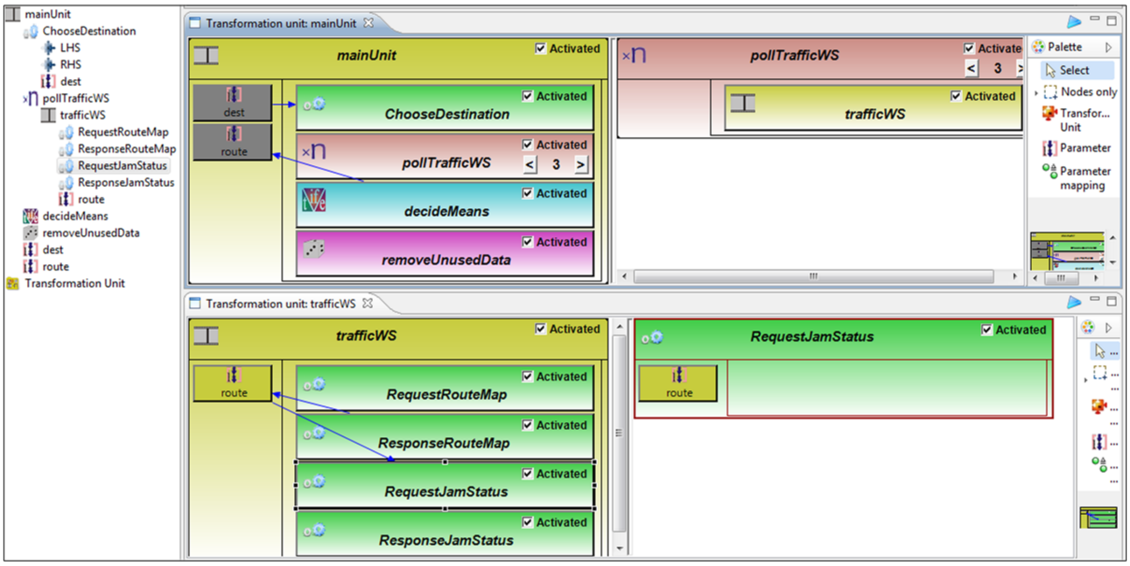

Figure 5: HENSHIN GUI with transformation unit editor

The transformation unit `mainUnit` shown in Figure 5 is the main control structure for the PMM example. It is a *SequentialUnit* (symbolized by a film strip as icon in the upper left corner) containing four subunits. This means that each subunit is applied once, in the given order from top to bottom. The first subunit, `ChooseDestination` is a transformation rule, marked by gear-wheels (see Figure 4 for the rule definition). This rule has an input parameter, the destination `dest`, a user-defined parameter. The second subunit of the main unit is a *CountedUnit* (symbolized by a "$\times n$" icon). The counter is set to 3, i.e. its subunit is applied three times. Unit `pollTrafficWS` is shown with its contents in the view to the upper right: it contains in turn a *SequentialUnit* (`trafficWS`) which controls four rules realizing the web service requests and processing the responses. The interaction of these rules within unit `trafficWS` can be seen in the lower left view: rule `ResponseRouteMap` produces an output parameter of type `Route` which serves again as input parameter for rule `RequestJamStatus`.

The third subunit of `mainUnit`, `decideMeans`, is a *ConditionalUnit* (symbolized by an *if-then-else* icon). Clicking on its field, a detailed view of this unit is opened (see Figure 6). Here, a condition called `AllRoutesJammed` (which will be discussed in Section 5) is checked which is given as an empty rule where we check its application condition. If the condition is evaluated to `true`, the two rules `ForbidCar` and `SelectBike` in the sequential unit are applied in this order. Otherwise, rule `ChooseRoute` is applied and the parameter `route` is returned to the parent unit `mainUnit`.

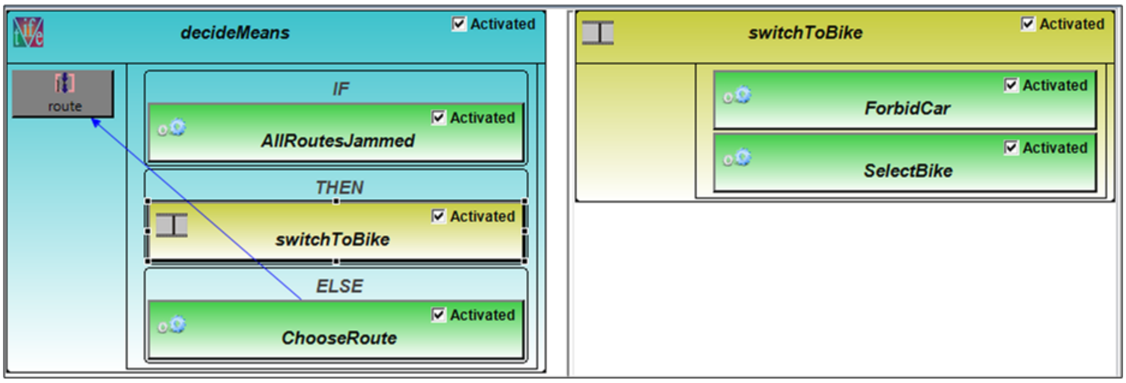

Figure 6: HENSHIN transformation unit `decideMeans`

The last child unit of `mainUnit` is the *IndependentUnit* `removeUnusedData` (with a die as icon symbol). This unit contains two rules, `DeleteJamStatus` and `DeleteUnused-Route` which perform garbage collection and are applied in arbitrary order, as long as possible.

5 Application Conditions

For graph transformation rules, well-known negative application conditions may be used that forbid to apply a rule if a certain structure is present in the graph. As a generalization, application conditions (introduced as *nested application conditions* in [HP09]) further enhance the expressiveness of graph transformations by providing a more powerful mechanism to control rule applications. While application conditions are as powerful as first order logic on graphs, we can still obtain most of the interesting results available for graph transformations *without* application conditions also for transformations *with* application conditions [EHL+10a, EHL10b] if certain additional properties hold.

Like transformation units, application conditions can be nested. Moreover, application conditions may be negated, and several application conditions may be combined by using the logical connectors AND and OR.

Definition 2 (Graph condition and application condition) A *graph condition ac* over graph G is of the form *true* or $\exists(a,c)$ where $a : P \to C$ is a graph morphism from a premise graph P to a conclusion graph C, and c is a condition over C. Moreover, Boolean formulas over conditions over P yield conditions over P, i.e. $\neg c$ and $\wedge_{j\in J} c_j$ are (Boolean) conditions over P where J is an index set and c, $(c_j)_{j\in J}$ are conditions over P. Additionally, $\exists a$ abbreviates $\exists(a, true)$, $\forall(a,c)$ abbreviates $\neg\exists(a,\neg c)$, *false* abbreviates $\neg true$, $\vee_{j\in J} c_j$ abbreviates $\neg \wedge_{j\in J} \neg c_j$, and $c \implies d$ abbreviates $\neg c \vee d$. A graph condition ac is called *application condition* of rule $r : L \to R$ if ac is a graph condition over L; an application condition of the form $\neg\exists a$ is usually called *negative application condition*.

A condition is satisfied by a morphism into a graph if the required structure exists, which can be verified by the existence of suitable morphisms.

Definition 3 (Satisfaction of conditions) Given a graph condition ac, a morphism $p : P \to G$ satisfies ac, written $p \models ac$, if $ac = true$. A morphism $p : P \to G$ satisfies condition $ac = \exists(a,c)$ if there is an injective graph morphism $q : C \to G$ such that $q \circ a = p$ and q satisfies c. The satisfaction of conditions by graphs and morphisms is extended to Boolean conditions in the usual way. A rule $L \to R$ is applicable only if the application condition ac is satisfied for its match $m : L \to G$, i.e. if $m \models ac$.

Let us consider once more the *ConditionalUnit* `decideMeans` from our PMM example (see Figure 6). Here, the condition `AllRoutesJammed` is expressed by an empty rule[4] with a nested application condition, shown in Figure 7.

In view $\boxed{1}$ of Figure 7, the empty rule is shown together with the outermost condition graph (condition over *LHS*). In the tree view of $\boxed{1}$, it can be seen that we require $\neg\exists Route$, i.e. a morphism from graph `Route` (consisting of a single `Route` node) into the host graph must not exist for the rule to be applicable. Since this application condition is nested, we require a further condition for the `Route` graph, formulated as disjunction (OR-construct) over two more conditions: $(\neg\exists\ HasNoJamStatus \vee \exists\ IsFree)$. This formula can be seen in the tree view of $\boxed{2}$, as well as in the corresponding visual hierarchical view where the formula is depicted as an OR block with two compartments. Clicking on one of the two parts of the disjunction in the visual view (or on one of the two OR branches in the tree view) opens the next level, either for the formula $\neg\exists HasNoJamStatus$ in $\boxed{3}$ or for the formula $\exists IsFree$ in $\boxed{4}$. Here, we have arrived at the basic level of graph morphisms. The complete nested application condition means that the empty rule is applicable (returns *true*) if there exists no route that has either no `JamStatus` node or that has a `JamStatus` node with attribute `jam=false`. Recall that in this case (all routes are jammed) unit `decideMeans` (see Figure 6) applies rule `switchToBike`, otherwise a route is chosen for the car as transport means.

6 Related Work

There are a number of model transformation engines which can modify models in EMF format such as ATL [JK05], EWL [KPPR07], Tefkat [LS05], VIATRA2 [VB07], MOMENT [Bor07].

[4] Note that we allow arbitrary transformation units as conditions in *ConditionalUnits*. While this may lead to side effects if a unit different from the empty rule is used, the conceptual advantage is that components of HENSHIN transformation units always are transformation units in turn.

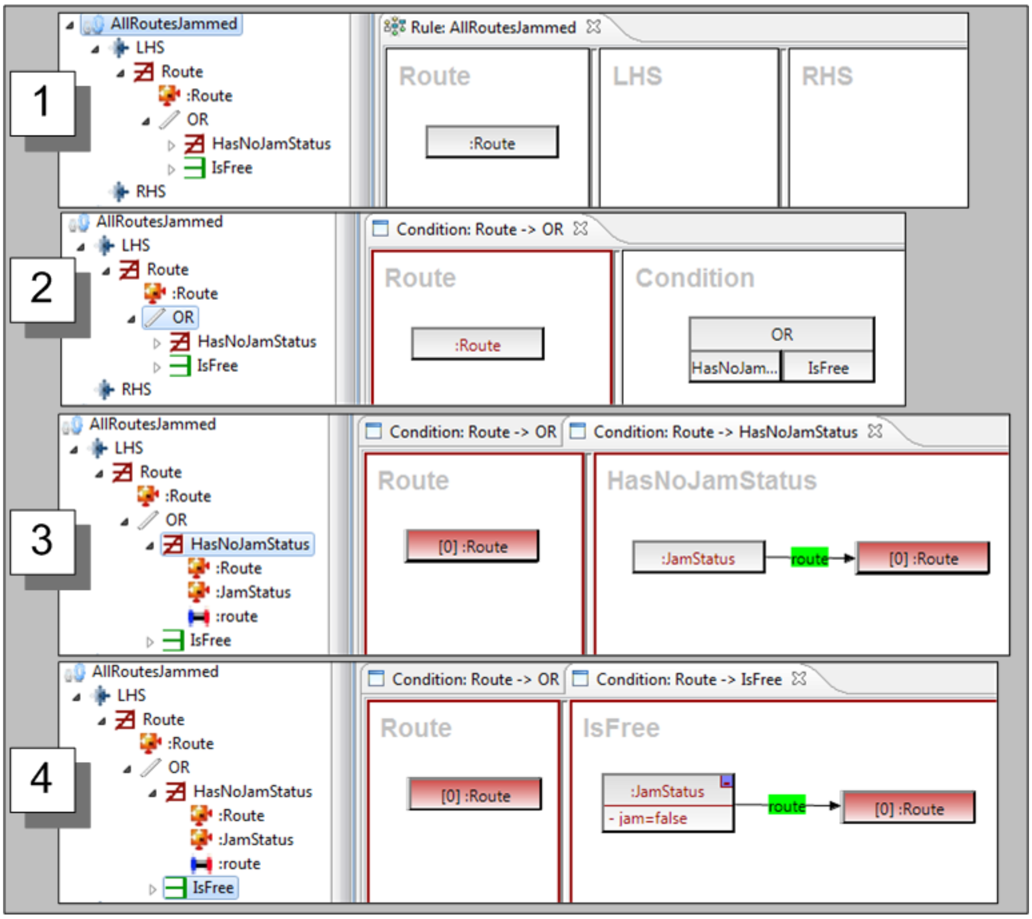

Figure 7: Empty rule `AllRoutesJammed` with application condition

For ATL, a formal semantics based on Maude has been introduced recently [TV10]. Formal semantics defined in Maude for MOMENT and for ATL might be exploited for analyzing EMF model transformations. None of these tool environments supports visual editing of control structures.

Graph transformation tools like PROGRES [SWZ99], AGG [AGG09], FuJaBa [FNTZ00] and MoTMoT [FOT10] feature visual editors which also support the definition of control structures, e.g. by story diagrams in FuJaBa, which were extended by implicit control in [MV08]. The tool GrGen.NET [GK08] also supports the arbitrary nesting of application conditions but is based on a textual specification language. MoTMoT (Model driven, Template based, Model Transformer) is a compiler from visual model transformations to repository manipulation code. The compiler takes models conforming to a UML profile for Story Driven Modeling as input and outputs Java Metadata Interface (JMI) code. Control structures are expressed by activity diagrams. Since the MoTMoT code generator is built using AndroMDA, adding support for other repository platforms (like EMF) is possible in principle and consists of adding a new set of code templates.

To the best of our knowledge, none of the existing EMF model transformation approaches (based on graph transformation or not) support confluence and termination analysis of EMF model transformation rules yet. Here, the HENSHIN approach and tool environment serves as a bridge to make well-established tool features and formal techniques for graph transformation available for model-driven development based on EMF.

7 Conclusion

In this paper, we presented two extensions for supporting controlled EMF transformations in our EMF transformation environment HENSHIN. The first extension supports the visual definition of HENSHIN transformation units which may be hierarchically nested (the basic unit being a rule) and which restrict the possible rule application sequences in a suitable way. The second extension concerns the definition of application conditions for transformation rules. Such conditions may be nested as well, and they may be combined by logical connectors such as AND and OR. We illustrated the usage of the extended HENSHIN environment by a simulation example of a personal mobility manager (PMM). Apart from the PMM example, HENSHIN has been applied also for larger case studies, e.g. for model refactorings [ABJ+10] and model-to-model transformations such as the *Ecore2Genmodel* case study of the Transformation Tool Contest 2010 [?][5].

The extended HENSHIN environment provides visual views for all control structures and conditions supporting zooming into deeper nesting levels. Thus, the visualization is independent of the complexity of the (nested) control structures, as only two levels are shown at a time. Both tree view editing and visual editing is supported at all levels. For editing formulas within application conditions, from the user's perspective an additional textual view of a complete formula might be desirable [GP96]. The integration of such a textual formula view in HENSHIN is work in progress.

A special kind of transformation units in HENSHIN are *AmalgamationUnits*, which are useful to specify *forall*-operations on recurring model patterns. An *AmalgamationUnit* is a multi-rule scheme containing the model pattern and a fixed kernel rule part. An amalgamated rule, induced by such a scheme, is a kind of parallel rule synchronized at the kernel rule part. Its application modifies all recurring instances of the model pattern in one step. The development of a visual editor within HENSHIN for *AmalgamationUnits* is work in progress.

Furthermore, on the theoretical side we aim to lift confluence and termination analysis results from the rule level to the level of transformation units.

References

[ABJ+10] T. Arendt, E. Biermann, S. Jurack, C. Krause, G. Taentzer. Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations. In *Proc. of the ACM/IEEE 13th Intern. Conf. on Model Driven Engineering Languages and Systems (MoDELS'10)*. LNCS 6394, pp. 121–135. 2010. http://tfs.cs.tu-berlin.de/publikationen/Papers10/ABJ+10.pdf

[AGG09] TFS-Group, TU Berlin. AGG. 2009. http://tfs.cs.tu-berlin.de/agg.

[5] On the TTC webpage http://planet-research20.org/ttc2010/ a Share demo of HENSHIN can be found as well.

[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.), *Proc. of the International Conference on Model Driven Engineering Languages and Systems (MoDELS'06)*. LNCS 4199, pp. 425–439. Springer, Berlin, 2006.
http://tfs.cs.tu-berlin.de/publikationen/Papers06/BEK+06a.pdf

[BEL+10] E. Biermann, C. Ermel, L. Lambers, U. Prange, G. Taentzer. Introduction to AGG and EMF Tiger by Modeling a Conference Scheduling System. *Int. Journal on Software Tools for Technology Transfer* 12(3-4):245–261, Juli 2010.
http://www.springerlink.com/content/p4n1g45627852743/

[BET08] E. Biermann, C. Ermel, G. Taentzer. Precise Semantics of EMF Model Transformations by Graph Transformation. In Czarnecki (ed.), *Proc. ACM/IEEE 11th International Conference on Model Driven Engineering Languages and Systems (MoDELS'08)*. LNCS 5301, pp. 53–67. Springer, 2008.
http://tfs.cs.tu-berlin.de/publikationen/Papers08/BET08.pdf

[Bor07] A. Boronat. *MOMENT: A Formal Framework for Model Management*. PhD thesis, Universitat Politècnica de València, 2007.

[EEPT06] H. Ehrig, K. Ehrig, U. Prange, G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. EATCS Monographs in Theor. Comp. Science. Springer, 2006.
http://www.springer.com/3-540-31187-4

[EHL+10a] H. Ehrig, A. Habel, L. Lambers, F. Orejas, U. Golas. Local Confluence for Rules with Nested Application Conditions. In Ehrig et al. (eds.), *Proceedings of Intern. Conf. on Graph Transformation (ICGT' 10)*. LNCS 6372, pp. 330–345. Springer, 2010.
http://tfs.cs.tu-berlin.de/publikationen/Papers10/EHL+10.pdf

[EHL10b] H. Ehrig, A. Habel, L. Lambers. Parallelism and Concurrency Theorems for Rules with Nested Application Conditions. *Electr. Communications of the EASST* 26:1–24, 2010.
http://journal.ub.tu-berlin.de/index.php/eceasst/issue/view/36

[EMF08] Eclipse Consortium. Eclipse Modeling Framework (EMF) – Version 2.4. 2008.
http://www.eclipse.org/emf.

[EMT09] TFS-Group, TU Berlin. EMF Tiger. 2009. http://tfs.cs.tu-berlin.de/emftrans.

[FNTZ00] T. Fischer, J. Niere, L. Torunski, A. Zündorf. Story Diagrams: A new Graph Rewrite Language based on the Unified Modeling Language. In Engels and Rozenberg (eds.), *Proc. of the* 6^{th} *International Workshop on Theory and Application of Graph Transformation (TAGT)*. LNCS 1764, pp. 296–309. Springer, Berlin, 2000.

[FOT10] FOTS-Group, University of Antwerp. MoTMoT: Model driven, Template based, Model Transformer. 2010. http://www.fots.ua.ac.be/motmot/index.php.

[GK08] R. Geiß, M. Kroll. GrGen.NET: A Fast, Expressive, and General Purpose Graph Rewrite Tool. In Schürr et al. (eds.), *Proc. 3rd Intl. Workshop on Applications of Graph Transformation with Industrial Relevance (AGTIVE'07)*. LNCS 5088. Springer, 2008.

[GP96] T. R. G. Green, M. Petre. Usability Analysis of Visual Programming Environments: a 'cognitive dimensions' framework. *Journal of Visual Languages and Computing* 7(2):131–174, 1996.

[HP09] A. Habel, K.-H. Pennemann. Correctness of high-level transformation systems relative to nested conditions. *Mathematical Structures in Computer Science* 19:1–52, 2009.

[JK05] F. Jouault, I. Kurtev. Transforming Models with ATL. In *MoDELS Satellite Events*. LNCS 3844, pp. 128–138. Springer, Berlin, 2005.

[KPPR07] D. Kolovos, R. Paige, F. Polack, L. Rose. Update Transformations in the Small with Epsilon Wizard Language. *Journal of Object Technology* 6(9):53–69, 2007.

[LMEP08] L. Lambers, L. Mariani, H. Ehrig, M. Pezze. A Formal Framework for Developing Adaptable Service-Based Applications. In Fiadeiro and Inverardi (eds.), *Proc. Fundamental Approaches to Software Engineering (FASE'08)*. LNCS 4961, pp. 392–406. Springer, 2008.

[LS05] M. Lawley, J. Steel. Practical Declarative Model Transformation with Tefkat. In *MoDELS Satellite Events*. LNCS 3844, pp. 139–150. Springer, Berlin, 2005.

[MV08] B. Meyers, P. Van Gorp. Towards a Hybrid Transformation Language: Implicit and Explicit Rule Scheduling in Story Diagrams. In *Proceedings of the 6th International Fujaba Days*. 2008.

[SWZ99] A. Schürr, A. Winter, A. Zündorf. The PROGRES-Approach: Language and Environment. In Ehrig et al. (eds.), *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 2: Applications, Languages and Tools*. Pp. 487 – 550. World Scientific, River Edge, NJ, USA, 1999.

[TV10] J. Troya, A. Vallecillo. Towards a Rewriting Logic Semantics for ATL. In Tratt and Gogolla (eds.), *Proc. of the Intern. Conf. on Model Transformation (ICMT'10)*. LNCS 6142, pp. 230–244. Springer, 2010.

[VB07] D. Varró, A. Balogh. The model transformation language of the VIATRA2 framework. *Science of Computer Programming* 68(3):214–234, 2007.