Visual Modeling of Controlled EMF Model Transformation using HENSIN

Enrico Biermann, Claudia Ermel, Johann Schmidt and Angeline Warning

13 pages
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 HENSHIN1 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 format2. 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/emf/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 H\textsc{ENSHIN} transformation engine provides classes that can freely be integrated into existing Java projects relying on EMF.

\textbf{Figure 1} shows the basic GUI of our H\textsc{ENSHIN} tool before the extensions presented in this paper. The tree view \([1]\) allows the modeler to import EMF \texttt{EPackages} containing the basic meta-model(s) defining the domain of the transformation. The initial model is edited in a visual editor \([2]\). In the rule editor \([3]\), 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 \textbf{Figure 1} defines an operation which adds a \texttt{Request} object and links it to existing \texttt{Departure} and a \texttt{Destination} objects. The property view \([4]\) shows additional information for selected objects. Note that all information edited using the editors in \([2]\), \([3]\) and \([4]\) can also be obtained via the tree view \([1]\).

![H\textsc{ENSHIN} GUI with visual editors for graphs and rules.](image)

The rule shown in \([3]\) can now be applied to the current model in \([2]\) leading to the transformed graph shown in \textbf{Figure 2}, where a \texttt{Request} object has now been created and linked to the \texttt{Departure} object named "Berlin" and the \texttt{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 A\textsc{GG} \cite{AGG09}. This is useful for

\(^3\) like in our running example, the simulation of a personal mobility manager based on a web service.
validation of consistent EMF model transformations which behave like algebraic graph transformations \cite{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 \cite{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 \cite{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 \((\text{LHS} \xrightarrow{r} \text{RHS})\) is a pair of graphs \((\text{LHS}, \text{RHS})\), called left-hand side and right-hand side, and an injective (partial) graph morphism \(r: \text{LHS} \rightarrow \text{RHS}\). 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 \( \langle LHS \xrightarrow{r} RHS \rangle \) 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 \( \langle LHS \xrightarrow{r} RHS \rangle \) 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 \( G \xrightarrow{m} H \) 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 \( G_0 \xrightarrow{m_1} G_1 \xrightarrow{..} G_n \) of direct graph transformations is called graph transformation, denoted as \( G_0 \xrightarrow{..} G_n \).

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

<table>
<thead>
<tr>
<th>EMF notion</th>
<th>Graph term</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMF model</td>
<td>Type graph with attribution, inheritance, multiplicities. Edges can be marked as containments.</td>
</tr>
<tr>
<td>Instance model</td>
<td>TypedList, attributed graph with containment edges</td>
</tr>
<tr>
<td>Class</td>
<td>Node in type graph</td>
</tr>
<tr>
<td>Object</td>
<td>Node in typed graph</td>
</tr>
<tr>
<td>Association</td>
<td>Edge in type graph (with possible multiplicities or containment mark)</td>
</tr>
<tr>
<td>Reference</td>
<td>Edge in typed graph that satisfies multiplicity and containment constraints</td>
</tr>
</tbody>
</table>
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](image)

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 ResponseJamStatus 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
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 HENSIN transformation units to encapsulate and control the order of rule applications.

4 HENSIN Transformation Units

HENSIN transformation units may be arbitrarily nested inside each other. The most basic unit is a transformation rule. A HENSIN 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](image)

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 ”×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.

The last child unit of mainUnit is the IndependentUnit removeUnusedData (with a die as icon symbol). This unit contains two rules, DeleteJamStatus and DeleteUnusedRoute 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 ∃(a,c) where a : P → 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 $\land_{j \in J} c_j$ 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 $\exists (a, \text{true})$, $\forall (a, c)$ abbreviates $\neg \exists (a, \neg c)$, false abbreviates $\neg \text{true}$, $\lor_{j \in J} c_j$ abbreviates $\neg \land_{j \in J} \neg c_j$, and $c \implies d$ abbreviates $\neg c \lor d$. A graph condition $ac$ is called application condition of rule $r : L \rightarrow R$ 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 $p : P \rightarrow G$ satisfies $ac$, written $p \models ac$, if $ac = \text{true}$. A morphism $p : P \rightarrow G$ satisfies condition $ac = \exists (a, c)$ if there is an injective graph morphism $q : C \rightarrow G$ 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 $L \rightarrow R$ is applicable only if the application condition $ac$ is satisfied for its match $m : L \rightarrow G$, 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 \(^1\) of Figure 7, the empty rule is shown together with the outermost condition graph (condition over LHS). In the tree view of \(^1\), it can be seen that we require $\neg \exists \text{Route}$, 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: $(\neg \exists \text{HasNoJamStatus} \lor \exists \text{IsFree})$. This formula can be seen in the tree view of \(^2\), 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 $\neg \exists \text{HasNoJamStatus}$ in \(^3\) or for the formula $\exists \text{IsFree}$ in \(^4\). 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.
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 [?]

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


On the TTC webpage http://planet-research20.org/ttp2010/ a Share demo of HENSHIN can be found as well.


