SoSyM manuscript No.
(will be inserted by the editor)

Formal Semantics of Consistent EMF Model Transformations by Algebraic Graph Transformation

Enrico Biermann[1], **Claudia Ermel**[1], **Gabriele Taentzer**[2]

[1] Technische Universität Berlin, Germany
e-mail: {enrico,lieske}@cs.tu-berlin.de
[2] Philipps-Universität Marburg, Germany
e-mail: taentzer@mathematik.uni-marburg.de

Received: date / Revised version: date

Abstract Model transformation is one of the key activities in model-driven software development. An increasingly popular technology to define modeling languages is provided by the Eclipse Modeling Framework (EMF). Several EMF model transformation approaches have been developed, focusing on different transformation aspects. To validate model transformations wrt. functional behavior and correctness, a formal foundation is needed. In this paper, we define EMF model transformations as a special kind of typed graph transformations using node type inheritance. Containment constraints of EMF model transformations are translated to a special kind of EMF model transformation rules such that their application leads to consistent transformation results only. Thus, we identify a kind of EMF model transformations which behave like algebraic graph transformations. As a consequence, the rich theory of algebraic graph transformation can be applied to these EMF model transformations to show functional behavior and correctness. Furthermore, we propose parallel graph transformation as a suitable framework for modeling EMF model transformations with *multi-object structures*. Multi-object structures at transformation rule level provide a flexible way to describe the transformation of structures with a flexible number of recurring structures, dependent on concrete model instances. Parallel graph transformation means parallelizing the application of model transformation rules at several occurences at once. We illustrate our approach by selected refactorings of simplified statechart models. Finally, we discuss the implementation of our concepts in a tool environment for EMF model transformations.

1 Introduction

Model-driven software development is considered as a promising paradigm in software engineering. Models are ideal means for abstraction and enable developers to master the increasing complexity of software systems. Since models are central artifacts in model-driven software development, the quality of generated software is directly dependent on the quality of models. Modifying models (i.e. performing model refactoring [**?**]) to improve the understandability and refine the model structure is an important part of model development. Throughout this work, we use model refactoring as an example for precisely specified model transformations.

The Eclipse Modeling Framework (EMF) [**?**] has evolved to a defacto standard technology to define models and modeling languages. EMF provides a modeling and code generation framework for Eclipse applications based on structured data models. The modeling approach is similar to that of MOF, actually EMF supports Essential MOF (EMOF) as part of the OMG MOF 2.0 specification [**?**].

Containment relations, i.e. aggregations, define an ownership relation between objects. Thereby, they induce a tree structure in model instantiations. In MOF and EMF, this tree structure is further used to implement a mapping to XML, known as XMI (XML Meta data Interchange) [**?**]. Containment always implies a number of constraints for model instantiations that must be ensured at run-time. As semantical constraints for containment edges, the MOF specification states the following:

- *"An object may have at most one container."*
- *"Cyclic containment is invalid."*

As mentioned earlier, EMF provides full implementations of instance models. These implementations always ensure these constraints. A third constraint may be useful for storing EMF models:

- *"There is a distinguished object, the root object, which contains (transitively) all other model objects."*

Model-driven development relies heavily on model transformations. EMF models can be manipulated by several approaches to rule-based model transformations for example ATL [**?**], EWL [**?**], Tefkat [**?**], VIATRA2 [**?**] and MOMEMT [**?**]. A transformation framework for EMF models which follows the concepts of algebraic graph transformation [**?**] as far as possible, is presented in [**?**]. However EMF model transformation does not always behave like algebraic graph transformation. The main reason is the difficulty to always satisfy the containment constraints of EMF.

As first main contribution of this paper, we define EMF models as typed graphs with containment edges. We identify a kind of model transformation rules leading to consistent EMF model graphs, if applied as normal graph transformation rules to consistent EMF model graphs. Thus, we identify a kind of EMF model transformations which behave like algebraic graph

transformations. The advantage of this approach is that we can apply the rich theory of algebraic graph transformation to EMF model transformations for validation. The theory provided by graph transformation helps us to identify a certain refactoring order by analyzing dependencies between refactoring rules. Moreover, refactoring steps might be in conflict to each other. Here, the graph transformation theory is helpful in analyzing potential conflicts between refactoring rules (see also [**?**]). Last but not least, a complex refactoring cannot be realized by just one rule. Therefore, termination is another issue which will be shown by checking sufficient termination criteria. This work has been introduced in [**?**] and is extended here.

Although graph transformation is an expressive, graphical and formal means to describe computations on graphs, it has its limitations. For example, when performing model refactorings, a restructuring action is often accompanied by update-actions on all involved model elements. For example, pulling up an attribute to a super class implies the deletion of such an attribute from all sub classes. A simple example are transformations of objects of the same class occurring multiple times which all have the same properties (e.g. being contained in the same container, or referencing the same objects). Such objects are called *multi-objects*. One way to transform multi-objects is the sequential application of rules, such that we have to explicitly encode an iteration over all the actions to be performed. Actually, this is neither the most natural nor efficient way to express the semantics. Thus, it is necessary to have a more powerful means to express parallel actions.

As second contribution of this paper, we propose the use of parallel graph transformation concepts, originally proposed in [**?**] and extended to synchronized, overlapping rules in [**?**] to define EMF model transformations with multi-object structures. The essence of parallel graph transformation is that (possibly infinite) sets of rules which have a certain regularity, so-called *rule schemes*, can be described by a finite set of *multi-rules* modelling the elementary actions. For the description of such rule schemes the concept of amalgamating rules at *kernel rules* [**?**] is used in this paper to describe the application of multi-rules in variable contexts. Therefore, the transformation of multi-object structures can be described in a general way. In order to respect the special restrictions of EMF models (imposed by the containment hierarchy), we lift the concept of amalgamated graph transformation to amalgamated EMF transformation by showing that also amalgamated EMF model transformations always lead to consistent EMF model graphs.

Our approach is illustrated by selected refactorings of simplified statechart models. The abstract syntax definition of statechart models mostly follows that in the UML2 EMF model [**?**], but does not consider regions, state actions, and structured transition inscriptions. Thus, we concentrate on the main statechart structure. Structural improvements of statecharts might comprise a number of refactoring steps. Amalgamated EMF model transformation enables us to express refactorings which involve a variable number of EMF object structures with a certain property.

This article is structured as follows: Based on the definition of EMF models as graphs in Section 2, EMF transformation rules are defined by a restricted form of graph transformation rules in Section 3. Thereafter, EMF model transformations can be defined as consistent graph transformations in Section 4. The basic transformation approach is extended by multi-object structures in Section 5. After having formally defined EMF model transformation by graph transformation concepts, we outline different kinds of verification techniques which are available for graph transformation and discuss their basic application to EMF transformations in Section 6. Section 7 presents the current tool support available for our EMF transformation approach. Finally in Section 8 and 9, we discuss related work and give conclusions.

2 Informal Introduction to Consistent EMF Model Transformation

ier die Beispiele 1)
2-5

3 EMF Models as Graphs

In this section, we start to lay the basis for the application of graph transformation theory to EMF model transformations. As first step, we consider EMF instance models[1] as typed graphs with containment edges. Typing is expressed by a so-called type graph. It has some similarities to a metamodel, but does not contain multiplicities and other constraints. Those have to be expressed by graph constraints, as done in [**?**].

Since containment plays a special role in EMF models, we distinguish a special kind of edge types defining containments. For being able to check for containment cycles later, we identify containment types (called *cycle-capable*) which may be part of containment cycles, summarized in C_{Cycle}[2]. Type graphs with containment types are called sound, if there is a root type which transitively contains all other concrete types.

Definition 1 (Graph and graph morphism) *A graph $G = (G_N, G_E, s_G, t_G)$ consists of a set G_N of nodes, a set G_E of edges, as well as source and target functions $s_G, t_G : G_E \to G_N$.*

Given two graphs G and H, a pair of functions (f_N, f_E) with $f_N : G_N \to H_N$ and $f_E : G_E \to H_E$ forms a graph morphism *$f : G \to H$, if it has the following properties:*

1. *$\forall e \in G_E : f_N \circ s_G(e) = s_H \circ f_E(e)$, with $s_G(e) \in G_N$, and*

[1] Note that the EMF community uses the terms "EMF model" for metamodel and "EMF instance model" for a model conforming to a metamodel.

[2] Cycle-capable containment types would be containment loops in a flattened type graph without inheritance.

2. $\forall e \in G_E : f_N \circ t_G(e) = t_H \circ f_E(e)$*, with* $t_G(e) \in G_N$*.*

If f_N *and* f_E *are inclusions, then* G *is called a subgraph of* H*, denoted by* $G \subseteq H$*.*

Definition 2 (Type graph) *A type graph* $TG = (T, I, A, C)$ *consisting of graph* $T = (T_N, T_E, s_T, t_T)$*, a graph* I*, called* inheritance graph *with* $I_N = T_N$ *and* $I_E \cap T_E = \emptyset$*, a set* $A \subseteq T_N$ *of* abstract nodes*, and a set* $C \subseteq T_E$ *of* containment edges*.*

For each node n *in* I_N *the* inheritance clan *is defined by* $clan_I(n) = \{n' \in I_N \mid \exists \text{ path } e_1, \ldots, e_i \in I_E \text{ with } i \leq 1 \text{ such that } s_I(e_1) = n', t_I(e_k) = s_I(e_{k+1}) \text{ with } 1 \leq k < i, \text{ and } t_I(e_i) = n\} \cup \{n\}$*.*

Furthermore we define a containment relation[3]

$$contains_{TG} = \{(n, m) \in T_N \times T_N \mid \exists c \in C \wedge x, y \in T_N : s(c) = x \text{ with } n \in clan(x) \wedge t(c) = y \text{ with } m \in clan(y)\} \cup \{(x, y) \in T_N \times T_N \mid \exists z \in T_N : (x \; contains_{TG} \; z \wedge z \; contains_{TG} \; y)\}$$

Based on $contains_{TG}$ *we create a relation containing cyclic containments only:*

$cycle_{TG} = \{(x, y) \in contains_{TG} | (y, x) \in contains_{TG}\}$*. Then a subset of containment edges, called* cycle-capable containment edges*, is defined whose instances might be part of containment cycles:* $C_{Cycle} = \{c \in C(TG) | \exists v_s \in clan(s(c)) \wedge \exists v_t \in clan(t(c)) : (v_s, v_t) \in cycle_{TG}\}$*.*

Type graph TG *is called* sound*, if there is a type node* $r \in T_N - A$*, called* root type*, such that the following holds:* $\forall n \in T_N - A - \{r\} : (r, n) \in contains_{TG}$

Example 1 (EMF model for simplified statecharts) In the running example, we consider the refactoring of a simplified form of statecharts. In Fig. **??** an EMF model for a simplified statecharts variant is shown. Events, actions, and parallel regions are not shown. Nevertheless, this model is interesting from the containment point of view: The type graph is sound, since type $StateMachine$ contains all concrete types, i.e. is the root type. Type $Vertex$ is contained in $StateMachine$ as well as in $State : Exp$. Since $State : Exp$ inherits from $Vertex$, it is again contained in $State : Exp$. Due to the same reason, $Pseudostate$ is contained in $State : Exp$. Moreover, $Transition$ and $FinalState$ are contained in $State : Exp$. Please note that $NamedElement$ is abstract and does not have to be contained in the root type. The containment edge of types $superState\_subState$ can be part of containment cycles because a $State : Exp$ is either the source of the edge or it is in the clan of $Vertex$ (the target of the edge). Pairs $(State, State)$, $(State, FinalState)$, and $(FinalState, State)$ are in the equivalence relation. Thus, a $State : Exp$ (resp. $FinalState$) may contain $States$ (resp. $FinalStates$) and form containment cycles.

Please note that the multiplicities shown in Fig. **??** are not formalized by type graphs, but have to be expressed by additional graph constraints [**?**] (not shown here).

[3] If there is no confusion, we use infix notation for $contains_{TG}$, e.g. $(x \; contains_{TG} \; y)$ instead of $(x, y) \in contains_{TG}$.

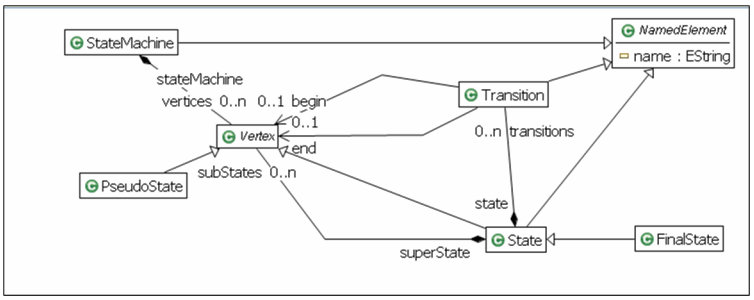

Figure 1 EMF model for simplified statecharts

Now we define EMF instance models as typed graphs where each object node has at most one container and no containment cycles do occur. Graphs fulfilling these requirements are called *graphs with containment*. Although EMF instance models do not need to be rooted in general, this property is important for storing them, or more general, to define the model's extent.

Definition 3 (Typed Graph and clan morphism) *Given a type graph* $TG = (T, I, A, C)$ *and a graph* G*. A* clan morphism $type_G \colon G \to TG$ *consists of a pair of functions* $(type_{G_N}, type_{G_E})$ *with* $type_{G_N} : G_N \to T_N$ *and* $type_{G_E} : G_E \to T_E$ *such that*

$$type_{G_N} \circ s_G(e) \in clan_I(s_T \circ type_{G_E}(e)) \text{ and } type_{G_N} \circ t_G(e) \in clan_I(t_T \circ type_{G_E}(e)).$$

If $type_G$ *exists, graph* G *is typed over* TG *and* $type_G$ *is called* typing morphism.

type is called concrete *if* $\forall n \in G_N : type_{G_N}(n) \notin A$*. Given a second clan morphism* $type' \colon G \to TG$*,* $type'$ *is* finer *than type, if*

$$type'_{G_N}(n) \in clan_I(type_{G_N}(n)) \text{ for all } n \in G_N \text{ and } type'_E = type_E.$$

Definition 4 (Graph with containment (C-graph)) *A graph with containment, short C-graph, is a graph* G *with a distinguished set of containment edges* $G_C \subseteq G_E$*. The containment edges induce the following transitive binary relation* $contains_G$*:*

- $contains_G = \{(x, y) \in G_N \times G_N \mid \exists e \in G_C : (s_G(e) = x \wedge t_G(e) = y)\ \} \cup$
  $\{(x, y) \in G_N \times G_N \mid \exists z \in G_N : (x \ contains_G \ z \wedge z \ contains_G \ y)\}$

All containment edges must fulfill the following properties (containment constraints):

- $e_1, e_2 \in G_C : t_G(e_1) = t_G(e_2) \ \Rightarrow \ e_1 = e_2$ *(at most one container).*
- $(x, x) \notin contains_G$ *for all* $x \in G_N$ *(no containment cycles).*

If G *is typed over* $TG = (T, I, A, C)$*, there is a clan morphism* $type_G \colon G \to TG$ *which is consistent with containment, i.e.* $\forall e \in G_C : type_{G_E}(e) \in C$*. Moreover, the set of containment edges* G_C *is induced by the typing morphism, i.e.* $G_C = \{e \in G_E | type_{G_E}(e) \in C\}$*.*

Please note that type graph TG is not a C-graph.

Definition 5 (Rooted graph) *A C-graph G is called* rooted*, if there is a node $r \in G_N$, called* root node*, such that $\forall x \in G_N$ with $x \neq r$: r contains$_G$ x.*

Example 2 (EMF instance model) Fig. **??** shows the main parts of a simple statechart modeling a phone. The containment can be well seen in the tree-like representation. However, source and target states of transitions are not shown in the tree view, but in a separate properties view (at the right bottom). It shows the properties of transition *caller hangs up* from state *DialTone* to state *Idle*. In addition, a purely graphical view of the same statechart is depicted on the right which is especially helpful to understand the transition structure inside of state *Phone*. This instance model fulfills both containment constraints and is rooted by a node of type *StateMachine*.

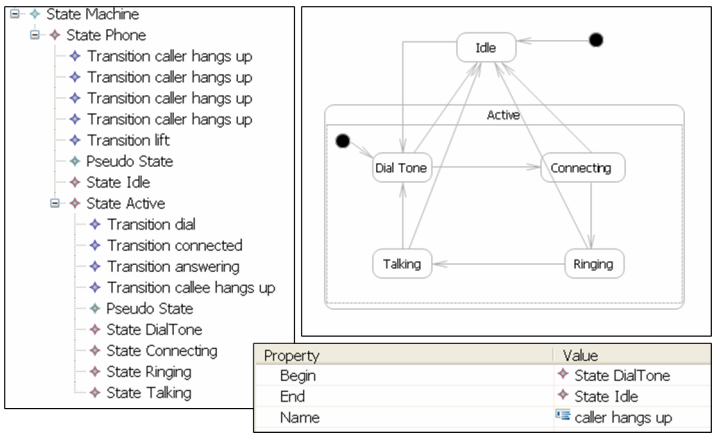

Figure 2 EMF instance model of a simple phone

In this paper, we concentrate on the structural issues of EMF models and their formalization. Needless to say that objects also may have typed attributes. They can be formalized by attributed graphs as done in [**?**]. For the formalization of EMF transformation by graph transformation, attributes will play a minor role.

4 EMF Transformation Rules as Consistent Graph Rules

In this section, we start to define a special kind of graph transformation which formalizes a form of EMF model transformation leading always to EMF models consistent with typing and containment constraints. For that purpose, the form of allowed transformation rules has to be restricted. Consistent transformation rules allow the following kinds of actions which change containments:

1. Create a new object node and connect it immediately to its container.
2. Delete a containment relation together with its target object node or change the container.
3. Create a containment relation with the target object node or change the container.
4. Create cycle-capable containment edges only, if the old and the new containers are both transitively contained in the same container.

Before we are able to precisely define consistent graph rules, we have to define relations between typed C-graphs, so-called C-graph morphisms. They define mappings of nodes and edges respectively, such that they are compatible with typing, source, and target functions. C-graph morphisms are needed to define graph rules such that the left (LHS) and the right-hand sides (RHS) are specified as well as the mapping from left to right. While the LHS defines the pattern to be found in the model, its relation to RHS formulates the actions to be performed. All object nodes and edges which occur in LHS but not in RHS are deleted, while all elements occurring in the RHS and not in the LHS are newly created. Elements occurring in both LHS and RHS have to be identified but are not changed. Moreover, negative application conditions (NACs) can be formulated. A NAC consists of an extension of the LHS where the structure not being part of the LHS is prohibited to occur in the model.

Definition 6 (C-graph morphism) *Given two C-graphs G, H, a pair of functions (f_N, f_E) with $f_N : G_N \to H_N$ and $f_E : G_E \to H_E$ forms a valid C-graph morphism $f : G \to H$, if it has the following properties:*

- $f_N \circ s_G(e) = s_H \circ f_E(e)$, $f_N \circ t_G(e) = t_H \circ f_E(e)$, *and*
- $\forall e \in G_C \;\Rightarrow\; f_E(e) \in H_C$ *(containment edges are preserved).*

If G and H are typed over TG, f must be type compatible, i.e. $type_G = type_H \circ f$.
If f_N and f_E are inclusions, then G is called a subgraph of H, denoted by $G \subseteq H$.

The conditions in Def. **??** are due to the use of abstract types for rule elements and express that *1.* retyping of elements is not allowed, *2.* newly created object nodes must be concretely typed, and *3.* node types occurring in NACs may be finer than in the LHS.

Definition 7 (Graph rule) *A graph rule typed over a type graph $TG = (T, I, A, C)$ is given by $p = (L \supseteq K \subseteq R,\ type,\ NAC)$, where L, K and R are C-graphs, type is a triple of typing morphisms $type = (type_L \colon L \to TG,\ type_K \colon K \to TG,\ type_R \colon R \to TG)$, and NAC is a set of pairs $nac_i = (N_i, type_{N_i}), i \in \mathbb{N}$ with $L \subseteq N_i$, and $type_{N_i} \colon N_i \to TG$ a typing clan morphism, such that the following conditions hold:*

1. $type_L \supseteq type_K \subseteq type_R$
2. $type_{R_N}(R'_N) \cap A = \emptyset$ *where* $R'_N := R_N - K_N$, *and*

3. $type_{N_i}$ is finer than $type_L$ *for all* $(N_i, type_{N_i}) \in NAC$

Moreover, we define $L'_C := L_C - K_C$, $R'_C := R_C - K_C$, *and* $L'_N := L_N - K_N$.

In the following definition, we formalize all actions that preserve consistent containment relations which have been described in the beginning of this section.

Definition 8 (Consistent graph rule) *Let* $p = (L \supseteq K \subseteq R,\ type,\ NAC)$ *be a graph rule as defined in* **??**. *Rule* p *is* consistent *wrt. containment if the following constraints are satisfied:*

1. *(node creation)* $\forall n \in R'_N$ *with* $type_R(n) = t_{TG}(c)$ *for some* $c \in TG_C$*:* $\exists e \in R'_C$ *with* $t_R(e) = n$,
2. *(containment edge deletion)* $\forall e \in L'_C$ *with* $t_L(e) = n$*:*
   $n \in L'_N \quad \vee \quad (n \in K_N \wedge \exists e' \in R'_C$ *with* $t_R(e') = n)$
3. *(containment edge creation)* $\forall e \in R'_C$ *with* $t_R(e) = n$*:*
   $n \in R'_N \quad \vee \quad (n \in K_N \wedge \exists e' \in L'_C$ *with* $t_L(e') = n)$
4. *(creation of cycle-capable containment edges)*
   $\forall e \in R'_C$ *with* $s_R(e) = n \wedge t_R(e) = m : \exists e' \in L'_C$ *with* $s_L(e') = o \wedge t_L(e') = m : ((o, n) \in contains_L \wedge (m, n) \notin contains_L) \vee (n, o) \in contains_L$

Example 3 (Refactoring rules) In Figs. **??** to **??**, we show three statecharts refactorings specified as EMF rules. Please note that object nodes with the same number mean that they are equal. The first one in Fig. **??** removes an isolated state from a composite state. Isolated means that no state or transition is contained and no transition starts or ends at this state. These constraints are guaranteed by the so-called *dangling condition* which allows the application of a graph transformation rule only if no edges are connected to nodes which are deleted by the rule (see Def. **??**). This rule is consistent, since the isolated state node is deleted together with its containment edge. Two transitions with the same source, the same target, and

Figure 3 Refactoring "Remove isolated state"

the same name are considered as redundant and can be removed by the graph rule in Fig. **??**. This rule is consistent, since the redundant transition node is deleted together with its containment edge. The third refactoring folds two transitions with the same name if the original target states are contained in the same super state. This refactoring consists of two rules, depicted in Fig. **??** which are applied as long as possible. While the upper rule folds two transitions both going to substates, the rule at the bottom folds a transition to a substate with the transition to its superstate having

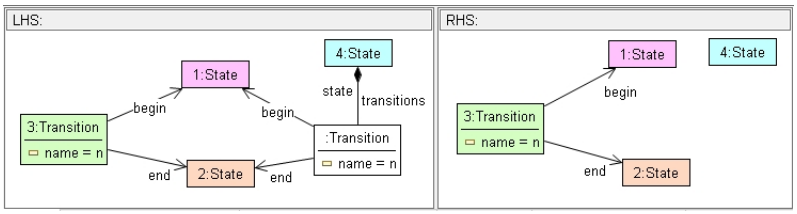

Figure 4 Refactoring "Delete redundant transition"

the same name. The application of both rules together allows folding of an arbitrary number of such transitions. The upper rule in Fig. **??** has to be equipped with a NAC isomorphic to the right-hand side to be sure that a transition like the new one does not already exist. This rule is consistent, since both transitions are deleted with their containment edges, while the new transition comes along with a new containment edge. The rule at the bottom of Fig. **??** is also consistent, since a transition is deleted together with its containment edge.

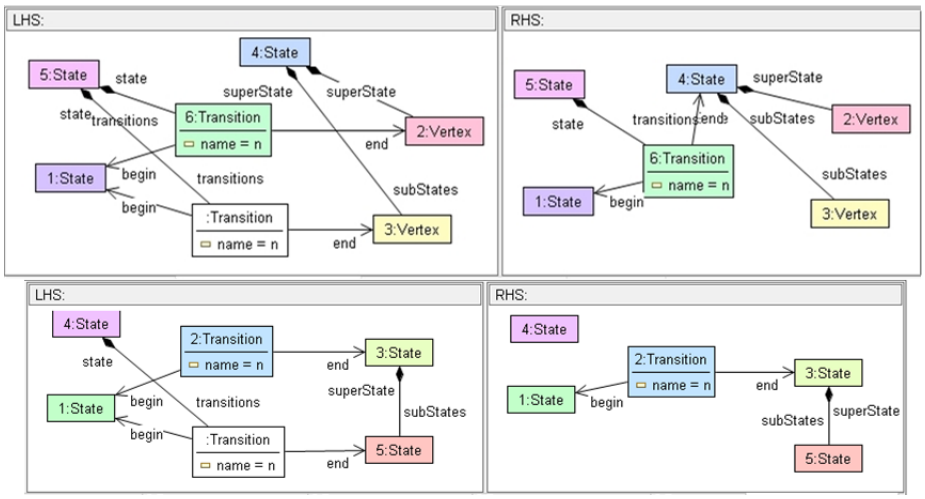

Figure 5 Refactoring "Fold incoming transitions"

The last refactoring moves a state out of a composite state up to the surrounding one. This is possible, if all adjacent transitions already belong to one of the surrounding states. This condition is checked by two similar NACs, one of which is shown together with the rule in Fig. **??**. It states that there must not be a transition adjacent to the state to be move up which still belongs to the old container state. The second NAC differs from the first one only in checking a similar condition for incoming transitions. This rule changes the containment relation of a state, i.e. its containment edge is deleted and a new one is created. Therefore, cond. 3 and 4 of Def. **??**

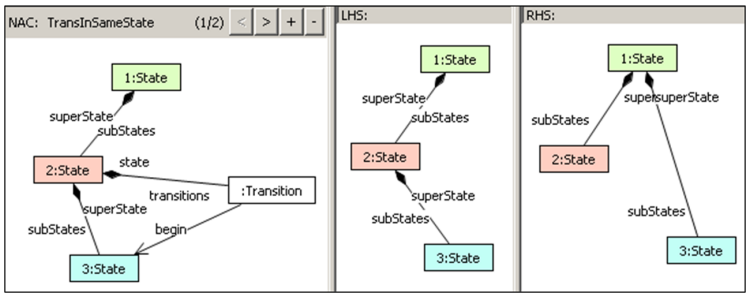

Figure 6 Refactoring "Move State Up"

are fulfilled. Since *superState* is cycle-capable, Cond. 5 has to be checked, too. The edge from *State* 1 to 3 is newly created, thus there has to be a containment relation from *State* 1 to *State* 2 in LHS and no containment relation from *State* 3 to 1. These two conditions are fulfilled, thus this rule is also consistent.

Summarizing, all refactorings can be formalized as consistent graph rules, since containment edges are deleted only if their corresponding containments are deleted. All newly created object nodes are created together with their containment edges. In the last refactoring rule, the container is changed consistently.

5 EMF Model Transformations as Consistent Graph Transformations

Having specified the kind of transformation rules which obey containment constraints, we define EMF model transformations as sequences of consistent graph transformation steps now. Thereafter, the first result of this contribution is presented. We show that each graph transformation step applying a consistent graph rule keeps the consistency of containment edges. Furthermore, we show that the application of consistent graph rules does not destroy roots.

A consistent rule may be applied to a graph (cf. Def. **??**) if

1. nodes are deleted only, if all adjacent edges occur in the rule as items to be deleted,
2. rule items may be matched with one and the same graph item only if they are preserved,
3. more abstractly typed nodes may be mapped to finer typed nodes, and
4. all NACs are fulfilled where NAC-nodes may also be more abstractly typed than graph nodes.

Definition 9 (Matching and application of graph rules) *Let* $p = (L \supseteq K \subseteq R,\ type,\ NAC)$ *be a graph rule as defined in* **??**, $(G, type_G)$ *a typed C-graph with* $type_G \colon G \to TG$ *being a concrete clan morphism, and* $m \colon L \to G$

a C-graph morphism. Then m is a match with respect to p and $(G, type_G)$, if

1. *m fulfills the so-called* dangling condition*, i.e. $\forall n \in L'_N : \nexists e \in G_E - m_G(L_E)$ with $s_G(e) = m_N(n) \vee t_G(e) = m_N(n)$*
2. *m fulfills the* identification condition *for nodes, i.e. $\forall x_1, x_2 \in L_N$ with $m_N(x_1) = m_N(x_2) : x_1, x_2 \in K_N$ (analogously for edges)*
3. *$type_G \circ m$ is finer than $type_L$.*
4. *m satisfies NAC, i.e. for each $nac_i = (N_i, type_{N_i}) \in NAC, i \in J$ there does not exists a C-graph morphism $o_i \colon N_i \to G$ such that $o_i|_L = m$ and $type_G \circ o_i$ is finer than $type_{N_i}$.*

Given a match m, rule p can be applied to $(G, type_G)$ which means to replace the matched part $m(L)$ by the corresponding right-hand side R of the rule. By $(G, type_G) \overset{p,m}{\Longrightarrow} (H, type_H)$, we denote the direct graph transformation *where rule p is applied to G at match m leading to the result graph $(H, type_H)$. The construction of a direct transformation is a double-pushout (DPO) which is shown in the diagram to the right with pushouts (PO_1) and (PO_2) in the category formal of typed graphs. Graph D is the intermediated graph after removing $m(L)$, and H is constructed as gluing of D and R along K (see [?]). Note that typing morphisms are not shown.*

The transformation definition in [**?**] is based on the double-pushout construction in the category of typed graphs and graph morphisms which is unique up to isomorphism. Result graph H is constructed as by taking the original graph G, deleting all items in the LHS and not in the RHS, and then adding all RHS-items not being in the LHS, disjointly. That means all newly created items get new identifiers:

- $H_N = G_N - m_N(L_N - K_N) \uplus (R_N - K_N)$
- $H_E = (G_E - m_E(L_E - K_E) \uplus (R_E - K_E)$
- $s_H(e) =$ if $e \in G_E$ then $s_G(e)$ else: if $s_R(e) \in K_N$ then $m_N(s_R(e))$ else $s_R(e)$
- analogously for t_H
- $type_{H_N}(n) =$ if $n \in G_N$ then $type_{G_N}(n)$ else $type_{R_N}(n)$
- analogously for $type_{H_E}$

Example 4 (Result model) Fig. **??** shows the main parts of the simple statechart in Fig. **??** after applying refactoring *Fold incoming transitions*. All the transitions going to state *Idle* have been replaced by only one from state *Active*. This is possible, since all these transitions had the same name. Again the tree-like and the diagram view are shown, together with the properties view for the new transition *caller hangs up*.

Theorem 1 (Consistent graph transformation step) *Given a consistent graph rule $p = (L \supseteq K \subseteq R, type, NAC)$ and a match $L \xrightarrow{m} G$ to a C-graph G which is concretely typed by $type_G \colon G \to TG$. Then, the result*

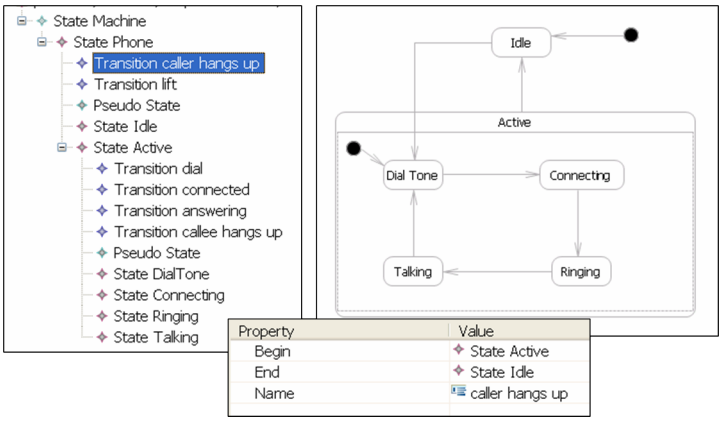

Figure 7 EMF instance model of Fig. **??** after refactoring

graph $(H, type_H)$ *of direct transformation* $(G, type_G) \overset{p,m}{\Longrightarrow} (H, type_H)$ *is a C-graph.*

Proof

To be shown:
1. $\forall e_1, e_2 \in H_C : t_H(e_1) = t_H(e_2) \implies e_1 = e_2$
2. $(x, x) \notin contains_H$ for all $x \in H_N$

1. Assuming $\exists e_1, e_2 : t_H(e_1) = t_H(e_2)$ and $e_1 \neq e_2$

Case 1: $e_1 \in G_C$ and $e_2 \in G_C$:
Since G is C-Graph, we have $e_1 = e_2$. This contradicts the assumption.

Case 2: $e_1 \in G_C$ and $e_2 \in R'_C$:
Subcase 1 $t(e_2) \in R'_N$:
e_1 is in G and is preserved by p and $t(e_1)$ is preserved as well. So $t_H(e_1) = t_H(m'(e_2))$ contradicts $t(e_2) \in R'_N$.
Subcase 2 $\exists e'_2 \in L'_C$ with $t(e'_2) = t(e_2)$ (due to Cond. 3 and 4 in Def. **??**): $t(e'_2) = t(e_2) = t(e_1) \implies e_1 = e'_2 \in L'_C$ because L is a C-Graph. Therefore e_1 is deleted which contradicts its existence in H.

Case 3: $e_1 \in R'_C$ and $e_2 \in R'_C$.
Due to Cond. 1, 3 and 4 in Def. **??**, we have three subcases:
Subcase 1 $t(e_1) \in R'_C$ and $t(e_2) \in R'_C$:
$t(e_1) = t(e_2) \implies e_1 = e_2$ because R is C-graph.
Subcase 2 $t(e_1) \in R'_C$ and $\exists e'_2 \in L'_C$ with $t(e'_2) = t(e_2)$
$t(e_2)$ is preserved by p and $t(e_1)$ is created. Therefore $t(e_1) \neq t(e_2)$ which contradicts the assumption.
Subcase 3 $\exists e'_1 \in L'_C$ with $t(e'_1) = t(e_1)$ and $\exists e'_2 \in L'_C$ with $t(e'_2) = t(e_2)$
R is a C-Graph and e_1 and $e_2 \in R_C$. So $t_R(e_1) = t_R(e_2)$ implies $e_1 = e_2$ and further implies $t_H(m'(e_1)) = t_H(m'(e_2))$

2. Assuming $\exists m \in H_N : (m,m) \in contains_H$. Then, there is a newly generated containment path $n \xrightarrow{*} m$ such that $(m,n) \in contains_G \wedge (n,m) \notin contains_G$. Due to Cond. 5 of Def. **??**, we have to consider the following two cases for the creation of the last edge $e \in R'_C$ of the path $n \xrightarrow{*} m$, i.e. either an edge $e' \in L'_C$ is "shifted down" (Case 1), or it is "shifted up" (Case 2) in the containment hierarchy.

Case 1 $\exists e' \in L'_C$ with $s(e') = o \wedge t(e') = m : (o,n) \in contains_L \wedge (m,n) \notin contains_L$
$(m,n) \in contains_G \wedge (o,n) \in contains_L \wedge (o,m) \in contains_L \implies (m,n) \in contains_L$ because of the uniqueness of a containment path (since G and L are C-graphs). This contradicts the assumption $\lightning$.

Case 2 $\exists e' \in L'_C$ with $s(e') = o \wedge t(e') = m : (n,o) \in contains_L$
$(o,m) \in contains_L \wedge (n,o) \in contains_L \implies (n,m) \in contains_L \implies (n,m) \in contains_G$ contradicts the assumption $\lightning$.

Theorem 2 (Rooted graph transformation step) *Given a consistent graph rule $p = (L \supseteq K \subseteq R, type, NAC)$ and a match $L \xrightarrow{m} G$ to a rooted graph G which is concretely typed by $type_G \colon G \to TG$ and satisfies NAC. Then, the result graph $(H, type_H)$ of the direct transformation $(G, type_G) \overset{p,m}{\Longrightarrow} (H, type_H)$ is rooted.*

Proof Assume that $G \overset{p,m}{\Longrightarrow} H$ with G being a rooted graph and H a non-rooted graph. Root node x of G is preserved by each consistent graph rule due to cond. 1 of Def. **??**. Hence, $x \in D \implies x \in H$ with x being root node in H. Since H is non-rooted, there exists at least one other node $y \in H$ with $x \neq y$ and $(x,y) \notin contains_H$. There are two possible ways how this situation may result from applying rule p:

1. Node y is generated by p: $\exists y \in R'$.
2. The containment edge e_G with $t(e_G) = y$ is deleted: $\exists e_L \in L' : m(e_L) = e_G$.

Case 1: According to Cond. 2 in Def. **??**, for the newly generated node y, there is a containment edge $e_R \in R'_C$ with $t_R(e_R) = y$. The source node of this newly generated containment edge is either another new node $z_R \in R'$ or an already existing node $n_K \in K$. Since only a finite number of new nodes can be generated by p, there is always one already existing node $n_K \in K$, which is the uppermost node of the newly generated containment path with y at the end, i.e. $(n_K, y) \in contains_R$. By comatch $R \xrightarrow{m'} H$ we get $n = m'(n_K)$ and $(n,y) \in contains_H$. Since n is preserved by p, we have $m(n_K) = n$ such that $(x,n) \in contains_G \implies (x,n) \in contains_H$. From $(x,n) \in contains_H$ and $(n,y) \in contains_H$ we conclude that $(x,y) \in contains_H$ which contradicts the assumption $\lightning$.

Case 2:
Subcase 2.1: Target node $t(e_G) = y$ is deleted together with e_G. Due to the

dangling condition, rule p may be applied only if y does not itself contain a node: $\nexists z : (y, z) \in contains_G$, or if all nodes contained recursively in y are also deleted by the rule: $\forall z : (y, z) \in contains_G : \exists z_L \in L'$ with $m(z_L) = z$. So, no nodes from the disconnected subtree (starting with y) remain in the graph which contradicts the assumption $\lightning$.

Subcase 2.2: Target node $t(e_G) = y$ is preserved by rule p, and by Cond. 3 of Def. **??**, a new containment path $n \xrightarrow{*} y$ is created. According to Def. **??**, Cond. 3, the source node n of the first edge $e \in R'_C$ of path $n \xrightarrow{*} y$ is a node preserved by rule p, i.e. $n_K \in K$ with $m(n_K) = n = m'(n_K)$. Hence, we have $(x, n) \in contains_H$ and $(n, y) \in contains_H$ and conclude that $(x, y) \in contains_H$ which contradicts the assumption $\lightning$.

6 Consistent EMF Model Transformations with Multi-Object Structures

In this section, we lift the essential concepts of parallel graph transformation [**?**] to EMF model transformation and also lift the consistency result for EMF model transformations from Section **??** to transformations with multi-object structures which we also call amalgamated EMF transformations.

Using amalgamated graph transformation, a system state modeled by a graph can be changed by several actions executed in parallel. Since graph transformation is rule-based without restrictive execution prescription, amalgamated graph transformation offers the possibility for massively parallel execution. The synchronization of amalgamated rule applications is described by common subrules, called *kernel rules*.

The simplest type of parallel actions is that of *independent actions*. If they operate on different objects they can clearly be executed in parallel. If they overlap just in reading actions on common objects, the situation does not change essentially. In graph transformation, this is reflected by a *parallel rule* which is a disjoint union of rules. The overlapping part, i.e. the objects which occur in the match of more than one rule, is handled implicitly by the match of the parallel rule. As the application of a parallel rule can model the parallel execution of independent actions only, it is equivalent to the application of the original rules in either order [**?**].

If actions are not independent of each other, they can still be applied in parallel if they can be synchronized by subactions. If two actions contain the deletion or the creation of the same substructure, this operation can be encapsulated in a separate action which is a common subaction of the original ones. A common subaction is modelled by the application of a *kernel rule* of all additional actions (modelled by *multi-rules*). The application of rules synchronized by kernel rules is then performed by gluing multi-rule instances at their kernel rule which leads to the corresponding *amalgamated rule*. The application of an amalgamated rule is called *amalgamated graph transformation*.

Formally, the synchronization possibilities of actions (multi-rule applications) are defined by an interaction scheme. For consistent amalgamated EMF transformations (also called EMF model transformations with multi-object structures), we need consistent interaction schemes where all rules are consistent.

Definition 10 (Consistent Interaction Scheme)
An interaction scheme $IS = (r_k, M)$ *consists of rule* r_k *called* kernel rule *and a set* $M = \{r_i | 1 \leq i \leq n\}$ *of rules called* multi-rules *with* $r_k \subseteq r_i$ *for all* $1 \leq i \leq n$.[4] *IS is* consistent, *if all rules are consistent.*

Example 5 An example of the construction of an amalgamated EMF transformation rule from an interaction scheme is given in Fig. **??**.

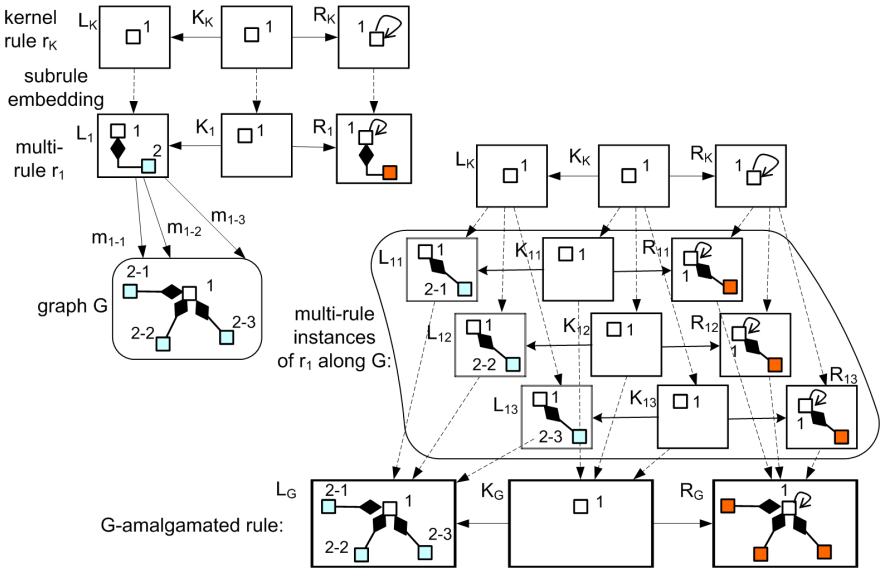

Figure 8 Construction of an amalgamated graph rule

The common sub-action (adding a loop to a object 1) is modelled by kernel rule r_K. We have only one multi-rule r_1 modelling that at each possible match that has an additional blue object 2 contained in object 1, object 2 shall be deleted together with its containment edge, and a new red object shall be inserted together with a containment edge such that the new red node is contained in object 1. Both the kernel-rule and the multi-rule are consistent, and we have a subrule embedding from the kernel rule to the multi-rule given by the three C-graph morphisms $L_K \to L_1, K_K \to K_1$ and $R_K \to R_1$. Given graph G, we have obviously three different matches from

[4] $r_k \subseteq r_i$ is valid if $L_k \subseteq L_i$, $K_k \subseteq K_i$, and $R_k \subseteq R_i$ as well as $type_{X_k} = type_{X_{i|X_k}}$ for $X = L, K, R$.

the multi-rule r_1 to G which overlap in the match of the kernel rule to G. Hence, we have three multi-rule instances, each of them with a different match to G. Gluing the multi-rule instances at their common kernel rule, we get the amalgamated rule with respect to G, shown at the bottom of Fig. **??**. The amalgamated rule contains the common action and, additionally, all actions from the multi-rules that do not overlap. Dashed arrows in Fig. **??** indicate rule embedding morphisms, embedding the kernel rule into the corresponding instances of the multi-rules, and the multi-rules into the amalgamated rule.

Definition 11 (Amalgamated EMF model transformation rule) *Given an interaction scheme $IS = (r_k, \{r_i | i \in I\})$ and match m_k for the kernel rule r_K to C-graph $(G, type_G)$, IS is applied at m_K by constructing another interaction scheme $IS' = (r_k, \{r_j | 1 \leq j \leq n\})$ called* interaction scheme instance *of IS, with each r_j being a copy (a rule instance, i.e. a new rule with $L_i \cap L_j = L_k, K_i \cap K_j = K_k$, and $R_i \cap R_j = R_k$) of some r_i for $i \in I$. Each copy r_j of rule r_i is constructed by a different match $m_{ij} : L_i \to G$, i.e. for each two rule instances r_j, r_l for all $1 \leq j < l \leq n$ which are copies of the same r_i, we have that $m_j(L_j) \neq m_l(L_l)$.*

There are maximal many rule instances r_j in the sense that each multi-rule match $m_i(L_i), i \in I$ corresponds to the match of one of its rule instances r_j: $\forall m_i : L_i \to G\ \exists m_j : L_j \to G$ s.t. $m_i(L_i) = m_j(L_j)$.

An amalgamated transformation rule *$r_A = (L_A \supseteq K_A \subseteq R_A, type, NAC)$, shortly* amalgamated rule*, is a rule where the left-hand sides of all multi-rule instances in IS' are glued over the kernel left-hand side L_K yielding L_A. Similarly, K_A and R_A are constructed. NAC is the union of all NAC_j and NAC_k. type is glued from the typing morphisms of all rule instances. Morphism m_A, called* amalgamated match *of r_A to G, is constructed by gluing all m_j which overlap at m_k.*

We speak of *amalgamated EMF model transformation* (or, alternatively, of *EMF model transformation with multi-object structures*) if the definition of the EMF model transformation is given by consistent interaction schemes. In this case, the application of an amalgamated EMF model transformation rule defines an EMF model transformation step. Note that a special interaction scheme consisting of a kernel rule only yields in only one amalgamated rule equal to the kernel rule.

In order to show that EMF instance graphs resulting from amalgamated EMF model transformation are consistent (Theorem **??**), we construct the amalgamated rule from a given consistent interaction scheme and show that this amalgamated rule is a consistent graph rule. Afterwards, we can apply Theorem **??**.

Theorem 3 *The construction in Def.* **??** *yields a unique amalgamated rule up to isomorphism.*

Proof The proof is given in [**?**].

Theorem 4 *Let $IS = (r_k, \{r_j | 1 \leq j \leq n\})$ be a consistent interaction scheme instance and $m_k : L_k \to G$ a match from r_k to a typed C-graph $(G, type_G)$. Then, the amalgamated transformation rule r_A resulting from the construction acc. to Def.* **??** *is consistent.*

Proof
W.l.o.g. we assume an interaction scheme where each multi rule is match exactly once. If I is not such an interaction scheme from the beginning, we can always construct an interaction scheme I' with this property by deleting those multi rules which are not applied and by copying those multi-rules which are applied more than once as often as needed.
Case $n = 0$**:** No multi-rule is applied. The amalgamated rule r_A is equal to the kernel rule r_K, which is consistent by assumption (I is consistent).

Case $n = 1$**:** There is one application of a multi-rule. The amalgamated rule r_K is equal to this multi-rule, thus it is consistent by assumption.

Case $n > 1$**:** We have to show that the amalgamated rule r_A satisfies all five consistency constraints for EMF rules according to Def. **??**:

1. (node creation) To show: $\forall n \in R'_{A_N}$ with $type_R(n) \in clan_I(t_{TG}(c))$ for some $c \in TG_C$: $\exists e \in R'_{A_C}$ with $t_{R_A}(e) = n$.
   W.l.o.g. $n \in R'_{j_N}$: Then, there is a unique $e \in R'_{j_C}$ with $t_{R'_{j_C}}(e) = n$ if $type_{R_j}(n) \in clan_I(t_{TG}(c))$ for some $c \in TG_C$, since r_j is consistent. There cannot be another $e \in R'_{A_C}$ with $t_{R_A}(e) = n$, since the construction of the amalgamated rule instances results in an overlap of multi-rules in the kernel rule only. (Note that the amalgamated match may glue multi-rule matches outside of kernel match.)
2. (containment edge deletion) To show: $\forall e \in L'_{A_C}$ with $t_{L_A}(e) = n$:
   $n \in L'_{A_N} \quad \vee \quad (n \in K_{A_N} \wedge \exists e' \in R'_{A_C}$ with $t_{R_A}(e') = n)$.
   W.l.o.g. $e \in L'_{j_C}$ with $t_{L_j}(e) = n$. Then, $n \in L'_{j_N} \vee (n \in K_{j_N} \wedge \exists e' \in R'_{j_C}$ with $t_{R_j}(e') = n)$, since r_j is consistent.
3. (containment edge creation) To show: $\forall e \in R'_{A_C}$ with $t_{R_A}(e) = n$:
   $n \in R'_{A_N} \quad \vee \quad (n \in K_{A_N} \wedge \exists e' \in L'_{A_C}$ with $t_{L_A}(e') = n)$
   W.l.o.g. $\forall e \in R'_{j_C}$ with $t_{R_j}(e) = n$. Then, $n \in R'_{j_N} \vee (n \in K_{j_N} \wedge \exists e' \in L'_{j_C}$ with $t_{L_j}(e') = n)$, since r_j is consistent.
4. (creation of cycle-capable containment edges)
   To show: $\forall e \in R'_{A_{C_{Cycle}}}$ with $s_{R_A}(e) = n \wedge t_{R_A}(e) = m : \exists e' \in L'_{A_C}$ with $s_{L_A}(e') = o \wedge t_{L_A}(e') = m :$ $((o, n) \in contains_{L_A} \wedge (m, n) \notin contains_{L_A}) \vee (n, o) \in contains_{L_A}$.
   W.l.o.g. $e \in R'_{j_{C_{Cycle}}}$ with $s_{R_j}(e) = n \wedge t_{R_j}(e) = m$.
   Then, there is $e' \in L'_{j_C}$ with $s_{L_j}(e') = o \wedge t_{L_j}(e') = m :$
   $((o, n) \in contains_{L_j} \wedge (m, n) \notin contains_{L_j}) \vee (n, o) \in contains_{L_j}$.
   In addition, we have to show that there is no $(m, n) \in contains_{L_j}$ for some $l \neq j$. Since r_j and r_l overlap in r_K only, $m, n \in L'_{K_N} \subseteq L'_{j_N}$ and $(m, n) \notin contains_{L_j} \implies (m, n) \notin contains_{L_l}$.

Corollary 1 *Given a consistent interaction scheme* $IS = (r_k, \{r_i | 1 \leq i \leq n\})$ *and typed C-graph* $(G, type_G)$*. Then after applying interaction scheme* IS *to* $(G, type_G)$*, the result graph* $(H, type_H)$ *is a typed C-graph as well.*

Proof Due to Theorem **??**, the amalgamated rule constructed from I is consistent. By Theorem **??**, consistent rules preserve C-graphs. Hence, the result graph H is again a C-graph.

Corollary 2 *Given a consistent interaction scheme* IS *like in Corollary* **??**. *Then, if* G *is a* rooted *C-graph, the result graph* H *after applying the interaction scheme* IS *to* G *is a* rooted *C-graph as well.*

Proof Due to Theorem **??**, the amalgamated rule constructed from IS is consistent. By Theorems **??** and **??**, we know that consistent rules preserve C-graphs and the rootedness of C-graphs. Hence, the result graph H is a rooted C-graph.

In addition to specify how multi-rules should be synchronized, we must decide where and how often a set of multi-rules should be applied. The basic way to synchronize complex parallel actions is to require that a rule should be applied at *all different matches* it has in a given graph (expressing massively parallel execution). In this paper, we restrict the *covering of* G (the image of all different matches from instances of multi-rules in G) to all different matches of multi-rules that overlap in the match of their common kernel rule and do not overlap anywhere else. For more complex covering constructions see [**?**].

Example 6 (Refactoring rules using multi-object structures)

Using rules with multi-object structures, certain refactorings can be expressed in a much simpler way. Let us consider, for instance, the refactoring *FoldIncomingTransitions* (see Fig. **??**). Using multi-object structures, we only need one rule instead of two. Moreover, the focus of the refactoring (to delete *all* transitions carrying the same name to sub-states and keep only *one* transition to their common super-state) becomes more clear.

Fig. **??** shows the corresponding interaction scheme with multi-object structures for the transitions to sub-states and their target states. All those transitions are deleted in one step, only one of them is preserved (which is transition 6) but redirected to the former super-state of the transitions. Note that in Fig. **??** and in the following screenshots of interaction schemes we use an integrated notation where we define the kernel rule and one multi-rule within one rule diagram. This is possible since each of our interaction schemes consists of a kernel rule and one multi-rule only. We distinguish objects belonging to the multi-rule only by drawing them as multi-objects (with indicated multiple boxes instead of simple rectangles). The kernel rule consists of all simple objects which are not drawn as multiple boxes. All arcs adjacent to multi-objects belong to the multi-rule only(and not to the kernel rule).

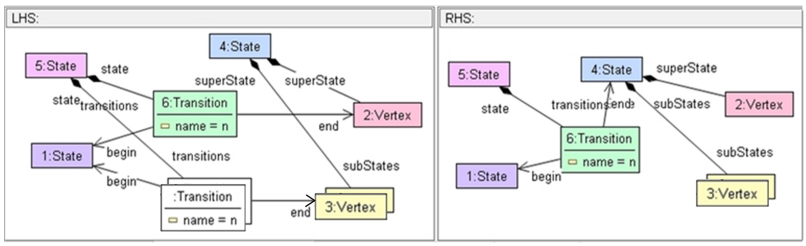

Figure 9 Refactoring "Fold incoming transitions" as interaction scheme

As second example we consider a complex refactoring *Remove Unreachable States* where an unreachable state is a state having no transitions from any other states pointing to it. This refactoring consists of two phases, applied to all unreachable states: at first, all transitions going out from an unreachable state are removed. After this step, all unreachable states are isolated, i.e. there are no transitions connected to it. Hence, in the second phase, all isolated states are removed.

The deletion of outgoing transitions from unreachable states comprises also the deletion of transition loops. This has to be dealt with by a separate interaction scheme due to injective matching. The scheme for the deletion of transition loops is shown in Fig. **??**. We use a multi-object to denote all transition loops of an unreachable state. The condition that the state is unreachable is modeled by the negative application condition which forbids the existence of any transition having this state as target and another state as source. The scheme has to be applied as often as possible (i.e. for all unreachable states with one or more transition loops). The scheme "Remove transition loops from unreachable state" is consistent, as the transition loops are deleted always together with their respective containment edge.

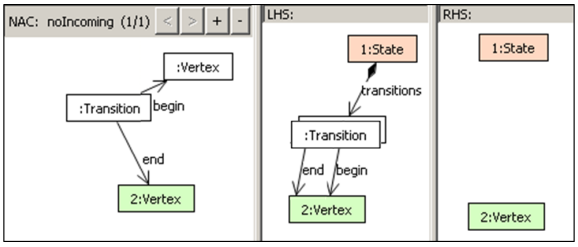

Figure 10 Refactoring "Remove transition loops from unreachable state" as interaction scheme

Fig. **??** shows the second interaction scheme for removing all remaining outgoing transitions from an unreachable state. We use multi-objects to denote all outgoing transitions and all target states reached by them from the unreachable state. The interaction scheme "Remove transitions

from unreachable state" is consistent, since the transition nodes are deleted together with their respective containment edges.

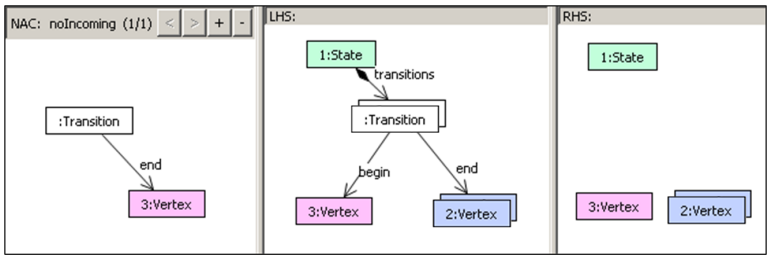

Figure 11 Refactoring "Remove transitions from unreachable state" as interaction scheme

Having now isolated all unreachable states by removing all transition loops and outgoing transitions, it remains to delete the isolated states. In Section **??**, Fig. **??**, we had defined a rule to remove an isolated state. This rule had to be applied as often as possible to remove all isolated states. The interaction scheme "Remove isolated states" in Fig. **??** has to be applied only once to solve the same task. It uses a multi-object to model all isolated states in the graph. Like in the simple rule in Fig. **??**, the check for isolation is done implicitly by the dangling condition which prevents the deletion of object vertices being source or target state of transitions or container of child states. The interaction scheme "Remove isolated states" is consistent, since the isolated vertex nodes are deleted together with their respective containment edges.

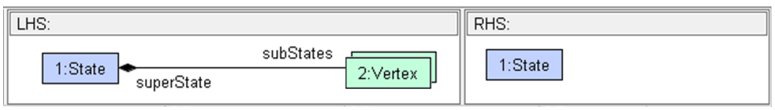

Figure 12 Refactoring "Remove isolated states" as interaction scheme

7 Towards Formal Validation of EMF Transformations

Having clarified the kind of EMF model transformations that can be formalized by algebraic graph transformation, we sketch how the rich graph transformation theory [**?**] can be applied to EMF model transformations. The theory is presented informally, just to give an idea how it can be applied in the context of EMF model transformations. In case of EMF transformations with multi-object structures the theory has to be applied on the level of amalgamated rule application. Some formal analysis techniques are supported by a general-purpose graph transformation tool, called AGG. To be

able to use it for EMF transformations, we have to translate an EMF transformation system (consisting of an EMF model and a set of EMF rules) to AGG first.

7.1 Conflicts and Dependencies

Graph transformation theory allows us to compute conflicts and dependencies of transformations by relying on the idea of critical pair analysis. The general-purpose graph transformation tool AGG [**?**] provides an algorithm implementing this analysis.

Critical pair analysis is known from term rewriting and can be used to check if a rewriting system contains conflicting computations. Critical pairs formalize the idea of showing a conflicting situation in a minimal context. From the set of all critical pairs we can extract the objects and links which cause conflicts or dependencies.

To construct minimal critical graphs, we basically consider all overlapping graphs of the left-hand sides of two rules with the obvious matches. If one of the rules contains NACs, extensions of the left-hand sides by parts of the corresponding NACs also have to be considered for the construction of overlapping graphs. The reasons why graph rules can be in conflict are threefold:

1. One rule application deletes a graph item which is in the match of another rule application (*delete-use* conflict).
2. One rule application produces a graph item that gives rise to a graph structure that is forbidden by a NAC of another rule application (*produce-forbid* conflict).
3. One rule application changes attributes being in the match of another rule.

AGG supports critical pair analysis for typed attributed graph transformations. As an important observation of the critical pair analysis, we can conclude that there is a preferred order for rule applications and reduce the number of actual conflicts in a transformation sequence. This is important for complex transformation sequences consisting of a number of steps to reach a certain goal. If all critical pairs can be resolved such that they can lead to the same result, the transformation system is called *locally confluent*.

Example 7 (Analyzing EMF model refactorings) Since all refactoring rules in the running example are consistent graph rules, the critical pair analysis is also available for EMF model refactorings. In [**?**], we have shown that critical pair analysis can be used to detect conflicts and dependencies between refactorings of class models. A formal specification of refactorings as graph transformation rules allows us to reason about dependencies between different types of refactorings. Due to the results presented in this paper, such an analysis is now available for EMF model refactorings.

Consider for example the application of refactoring rule *MoveStateUp* in Fig. **??** two times in parallel. It might happen that a state (2) which shall be moved out has a substate (6) that shall also be moved out. In that case, the substates relation between state (2) and its superstate (1) has to be deleted and is also used. Thus, we can identify a delete-use conflict here. Fig. **??** shows the critical pair for this conflict[5]. This conflict can be resolved by moving inner states up at first.

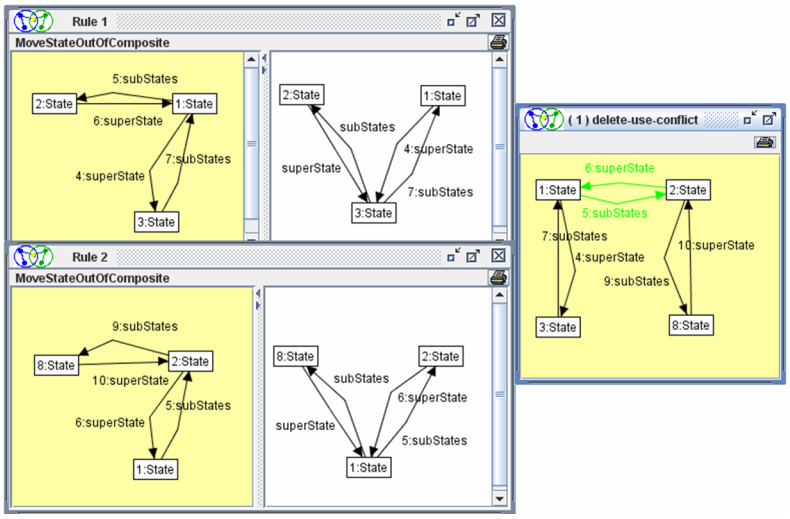

Figure 13 Conflicting applications of Rule *MoveStateUp* two times

7.2 Termination Criteria

Besides conflicts and dependencies, we can also apply the developed termination criteria for graph transformation to consistent EMF model transformations. A locally confluent and terminating transformation system is confluent in general and thus shows a functional behavior. That means for each input there is a unique transformation result. In general, termination is undecidable for graph transformation systems. But if a system meets certain termination criteria, we can conclude that it is terminating.

The proof for termination of graph transformation systems [**?**] is based on layered graph transformation systems with deletion and non-deletion layers. Informally, the deletion layer conditions express that the last creation of a node of a certain type should precede the first deletion of a node of the same type. Furthermore, each rule application should delete more items of

[5] Note that rules and graphs are no longer shown as EMF instance models but as AGG graphs.

a certain type than it creates. Non-deletion layer conditions ensure that if an element of a certain type occurs in the LHS of a rule then all elements of the same type were already created in previous layers.

Termination of statechart refactorings Considering again the refactoring rules in Section **??**, we like to check if the first three refactorings can be applied automatically to optimize a statechart as far as possible. Since the first three rules are deleting ones and each rule decreases the number of model elements, the deletion layer conditions are satisfied and thus the refactoring is terminating. Refactoring rule "Move State Up" is of a different kind. It does not make sense to apply this refactoring automatically, since that would lead to statecharts where all states are contained in the uppermost state. Thus, we do not have to check the termination of applying this rule as long as possible.

8 Implementation of EMF Transformation in EMF Henshin

Recently, EMF Henshin [**?**,**?**] (**T**ransformat**I**on **GE**ne**R**ation) has been developed as an Eclipse plug-in supporting modeling and execution for EMF model transformations, based on structured data models and graph transformation concepts.

The main goal is to provide an EMF transformation approach which is precisely defined such that verification of EMF model transformation becomes possible. Rule applications change an EMF model instance in-place, i.e. an EMF instance model is modified directly, without copying it before. Moreover, control of rule applications is supported by EMF Henshin, as well as pre-definition of (parts of) the match. EMF Henshin currently consists of a *graphical editor* for visually defining EMF model transformation rules and a transformation engine which executes the defined rules. The transformation engine provides classes which can freely be integrated into existing projects which rely on EMF models. Another advantage is that EMF Henshin directly operates on those models.

Fig. **??** shows a screenshot of the EMF Henshin meta model. It shows that the objects of graphs, namely nodes, edges and attributes are typed over *EClass*, *EReference*, and *EAttribute*, which are classes of the Ecore model. Moreover, it is possible to define control structures (transformation units) over rules. For example, there are constructs for non-deterministic rule choices (*IndependentUnit*) or rule priority (*PriorityUnit*). Those constructs can be nested arbitrarily to define complex control structures. Furthermore, passing of model elements and parameters from one rule to another is possible by using input and output parameters (not shown in Fig. **??**). These can be used to model object flow between transformation steps. For amalgamated EMF transformations, a special control unit exists (*AmalgamatedUnit*) which represents a basic interaction scheme consisting of one kernel and one multi-rule. An extension to define more general interaction schemes with multi-object structures is work in progress.

Currently there exist two implementations of the transformation engine. One is written in Java while the other one translates the transformation rules to AGG [?,?]. This is useful for the verification of consistent EMF model transformations which behave like algebraic graph transformations, e.g. to show functional behavior and correctness (see Section ??).

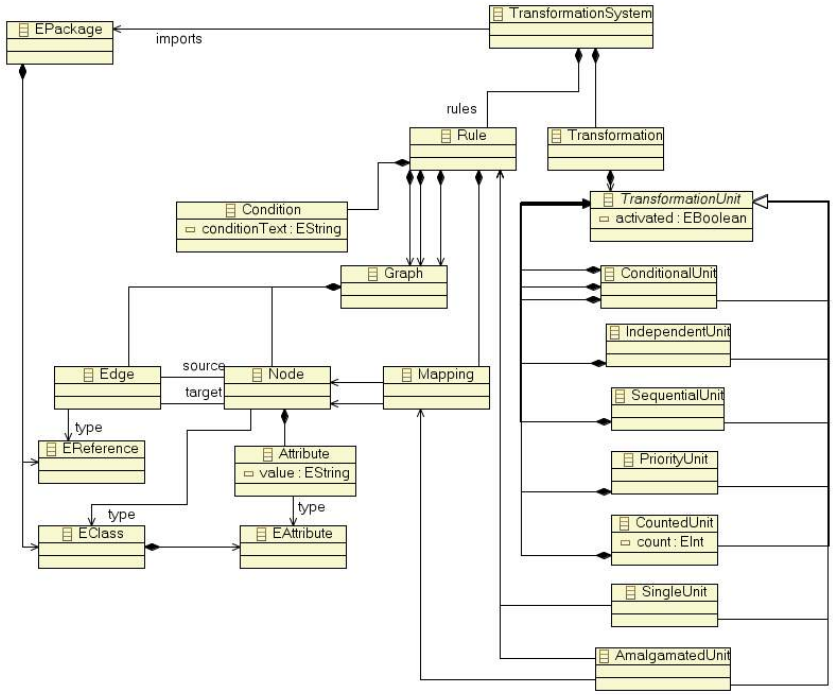

Figure 14 EMF HENSHIN meta model

The EMF HENSHIN transformation engine provides an API to apply rules to EMF model instances in user applications. To this end, rules defined in EMF HENSHIN need to be passed to class *GenericRule*. Once instanciated, partial matches for a rule application can be set (including the initialization of parameters). A rule is applied by calling the *execute()*-method of *GenericRule*, where optional input parameters and matches are respected. After a rule application, the output parameters of the rule may be read and used for later rule applications. Transformation units are executed in a similar way by using class *GenericUnit*.

9 Related Work

There are a number of model transformation engines which can modify EMF models: ATL [?], EWL [?], Tefkat [?], VIATRA2 [?], MOMEMT [?],

etc. Each of these projects can be used to specify model transformations such as the statechart refactorings presented. However, an approach which allows to transform EMF models in-place is preferred. In contrast to most model transformation engines, MOMEMT as well as our approach have a formal basis. MOMENT is based on Maude which might be exploited for validation of EMF model transformations. To the best of our knowledge, none of the existing transformation approaches supports confluence and termination analysis of EMF model transformations yet.

There are two tool-based approaches known to us which also realize amalgamated graph transformation: AToM^3 and GROOVE. AToM^3 supports the explicit definition of interaction schemes in different rule editors [**?**], whereas GROOVE implements rule amalgamation based on nested graph predicates [**?**]. A related conceptual approach aiming at transforming collections of similar subgraphs is presented in [**?**] by Gronmo et.al. The main conceptual difference is that we amalgamate rule instances whereas Gronmo et.al. replace all collection operators (multi-objects) in a rule by the mapped number of collection match copies. Similarly, Hoffmann et al. define a cloning operator in [**?**] where cloned nodes roughly correspond to multi-objects. Moreover, graph transformation tools PROGRES [**?**] and FuJaBA [**?**] feature so-called set nodes which can be duplicated as often as necessary. These two approaches are more restricted, since they focus on multiple instances of single nodes instead of graph parts. None of these related approaches are applied to the transformation of EMF models.

10 Conclusion

A precise specification of EMF model transformations can be advantageously used to validate important properties such as functional behavior. In this paper, we identified a variant of EMF model transformation that can be defined as algebraic graph transformation with node type inheritance. This result can be used to validate EMF model transformation based on the rich graph transformation theory available. These validation facilities are illustrated by selected model refactorings, formulated for a restricted form of statechart models.

The consistency constraints for transformation rules limit our approach, but not dramatically, since they mostly emerge directly from EMF containment constraints. However, the deletion of a containment relation without target object might be attractive as it could lead to a detachment of a complete subtree. This kind of implicit deletion is not allowed, hence the deletion of an object tree has to be performed explicitly by applying rules which delete stepwise. The extension of the basic approach by multi-object structures allows to specify more complex transformations in one step. To specify suitable pre-conditions, our approach allows to test for existence and non-existence of object structures and to check attribute values. Sometimes more complex pre-conditions are needed. It is up to future work to extend the approach to so-called nested conditions as presented in [**?**].

EMF Henshin [**?**] currently supports the generation of transformation code in Java and the translation of EMF models and rules to AGG [**?**], a tool environment for algebraic graph transformation. However, a validation tool for EMF model transformations which seamlessly integrates analysis techniques provided by AGG, is left to future work.

Amalgamated EMF transformation essentially extends the capabilities of EMF transformation based on simple graph transformation [**?**] by allowing parallel execution of synchronized EMF transformation rules. The possibility of synchronizing rules and of forcing a step to be, in a sense, maximally parallel is very useful for specifying refactoring operations. This is the case for a large number of refactorings, but also for other examples, e.g. the simulation of visual behavioral models, such as Statecharts with AND states. It has been shown in the paper that amalgamated EMF transformation using consistent rules and instance models again always leads to consistent EMF instance models which satisfy the containment constraints of EMF. We are currently extending EMF Henshin to the specification and execution of EMF transformations with multi-object structures.

Further valuable forms of quality assurance for EMF model transformation such as guidelines and refactoring as syntactical techniques as well as testing and debugging as semantical forms would make a comprehensive development environment for EMF model transformations perfect.

References

1. Mens, T., Tourwé, T.: A survey of software refactoring. Transactions on Software Engineering **30**(2) (February 2004) 126–139
2. Eclipse Consortium: Eclipse Modeling Framework (EMF) – Version 2.4. (2008) `http://www.eclipse.org/emf`.
3. Object Management Group: Meta Object Facility (MOF) Core Specification Version 2.0. `http://www.omg.org/technology/documents/modeling_spec_catalog.htm\#MOF` (2008)
4. Object Management Group: MOF 2.0 / XMI Mapping Specification. (2008) `http://www.omg.org/technology/documents/formal/xmi.htm`.
5. Jouault, F., Kurtev, I.: Transforming Models with ATL. In: Proc. Workshop Model Transformations in Practice at MoDELS'05, Montego Bay, Jamaica (2005)
6. Kolovos, D., Paige, R., Polack, F., Rose, L.: Update transformations in the small with epsilon wizard language. Journal of Object Technology **6**(9) (October 2007) 53–69 Special Issue. TOOLS EUROPE.
7. Lawley, M., Steel, J.: Practical Declarative Model Transformation With Tefkat. In: In Proc. Model Transformation in Practice Workshop, Models Conference. (2005)
8. Varró, D., Balogh, A.: The model transformation language of the viatra2 framework. Sci. Comput. Program. **68**(3) (2007) 214–234
9. Boronat, A.: MOMENT: A Formal Framework for Model Management. PhD thesis, Universtat Polit'e?cnica de Val'encia (2007)

10. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. EATCS Monographs in Theoretical Computer Science. Springer (2006)
11. Biermann, E., Ehrig, K., Köhler, C., Kuhns, G., Taentzer, G., Weiss, E.: Graphical Definition of In-Place Transformations in the Eclipse Modeling Framework. In Nierstrasz, O., Whittle, J., Harel, D., Reggio, G., eds.: Proc. 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS'06). Volume 4199 of lncs., springer (2006) 425–439
12. Mens, T., Taentzer, G., Runge, O.: Analysing refactoring dependencies using graph transformation. Software and System Modeling **6**(3) (September 2007) 269–285
13. Biermann, E., Ermel, C., Taentzer, G.: Precise Semantics of EMF Model Transformations by Graph Transformation. In Czarnecki, K., ed.: Proc. ACM/IEEE 11th International Conference on Model Driven Engineering Languages and Systems (MoDELS'08). Volume 5301 of LNCS., Springer (2008) 53–67
14. Ehrig, H., Kreowski, H.J.: Parallel graph grammars. In Lindenmayer, A., Rozenberg, G., eds.: Automata, Languages, Development. Amsterdam: North Holland (1976) 425–447
15. Taentzer, G.: Parallel and Distributed Graph Transformation: Formal Description and Application to Communication-Based Systems. PhD thesis, TU Berlin (1996) Shaker Verlag.
16. Böhm, P., Fonio, H.R., Habel, A.: Amalgamation of graph transformations: a synchronization mechanism. Journal of Computer and System Science **34** (1987) 377–408
17. Eclipse Consortium: Eclipse Model Development Tools. (2007) `http://www.eclipse.org/modeling/mdt`.
18. Rensink, A., Taentzer, G.: Ensuring structural constraints in graph-based models with type inheritance. In: Proc. Fundamental Approaches to Software Engineering (FASE). Volume 3442 of LNCS., Springer (2005) 64–79
19. TFS-Group, TU Berlin: AGG. (2009) `http://tfs.cs.tu-berlin.de/agg`.
20. Tiger Project Team, Technische Universität Berlin: EMF Tiger (2009) `http://tfs.cs.tu-berlin.de/emftrans`.
21. Taentzer, G.: AGG: A Graph Transformation Environment for Modeling and Validation of Software. In Pfaltz, J., Nagl, M., Boehlen, B., eds.: Application of Graph Transformations with Industrial Relevance (AGTIVE'03). Volume 3062 of LNCS. Springer (2004) 446 – 456
22. de Lara, J., Ermel, C., Taentzer, G., Ehrig, K.: Parallel Graph Transformation for Model Simulation applied to Timed Transition Petri Nets. In: Proc. Graph Transformation and Visual Modelling Techniques (GTVMT) 2004. (2004)
23. Rensink, A., Kuperus, J.H.: Repotting the geraniums: On nested graph transformation rules. In: Int. Workshop of Graph Transformation and Visual Modelling Techniques (GT-VMT'09). (2009)
24. Grønmo, R., Krogdahl, S., Møller-Pedersen, B.: A collection operator for graph transformation. In: Int. Conf. on Model Transformation (ICMT'09). (2009) To appear.
25. Hoffmann, B., Janssens, D., van Eetvelde, N.: Cloning and expanding graph transformation rules for refactoring. In: Int. Workshop on Graph and Model Transformation (GraMoT'05). Volume 152 of Electr. Notes on Theor. Comput. Sci., Elsevier (2006) 53–67

26. Schürr, A., Winter, A., Zündorf, A.: The PROGRES-approach: Language and environment. In Ehrig, H., Engels, G., Kreowski, J.J., Rozenberg, G., eds.: Handbook of Graph Grammars and Computing by Graph Transformation, Volume 2: Applications, Languages and Tools, World Scientific (1999)
27. Fischer, T., Niere, J., Torunski, L., Zündorf, A.: Story diagrams: A new graph rewrite language based on the unified modeling language. In Engels, G., Rozenberg, G., eds.: Proc. of the 6th International Workshop on Theory and Application of Graph Transformation (TAGT), Paderborn, Germany. Volume 1764 of LNCS., Springer (2000) 296–309
28. Habel, A., Pennemann, K.H.: Correctness of high-level transformation systems relative to nested conditions. Mathematical Structures in Computer Science **19** (2009) 1–52