Precise Semantics of EMF Model Transformations by Graph Transformation

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

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

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. We illustrate our approach by selected refactorings of simplified statechart models.

Keywords: model-driven software development, Eclipse Modeling Framework, model transformation, graph transformation.

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 [10]) 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 specifying model transformations.

The Eclipse Modeling Framework (EMF) [4] has evolved to one of the standard technologies to define 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) [13]. 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. A transformation framework for EMF models which follows the concepts of algebraic graph transformation [5] as far as possible, is presented in [2]. But 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.

In this paper, we define EMF models as typed graphs with containment edges. We identify a kind of model transformation rules which lead 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.

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 [6], 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. 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 [9]). 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.

2 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[3] 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 [11].

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

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}[4]. Type graphs with containment types are called sound, if there is a root type which transitively contains all other concrete types.

Definition 1 (Graph). *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 $s_G, t_G : G_E \to G_N$.*

Definition 2 (Type graph). *A type graph $TG = (T, I, A, C)$ consisting of graph $T = (N, E, s, t)$, a graph I, called* inheritance graph *sharing the same set of nodes N with T (but not the edges), a set $A \subseteq N$ of* abstract nodes*, and a set $C \subseteq E$ of* containment edges*.*
For each node n in I the inheritance clan *is defined by $clan_I(n) = \{n' \in N \mid \exists$ path $n' \xrightarrow{*} n$ in $I\}$ where path of length 0 is included, i.e. $n \in clan_I(n)$.*
Furthermore we define a containment relation[5]
$contains_{TG} = \{(n, m) \in N \times N \mid \exists c \in C \wedge x, y \in N : s(c) = x$ *with* $n \in clan(x)$
$\wedge\, t(c) = y$ *with* $m \in clan(y)\} \cup$
$\{(x, y) \in N \times N \mid \exists z \in N : (x\ contains_{TG}\ z \wedge z\ contains_{TG}\ y)\}$
Based on $contains_{TG}$ we create the equivalence relation
$equiv_{contains} = r(contains_{TG}) - \{(x, y) \in contains_{TG} | (y, x) \notin contains_{TG}\}$.
with r being the reflexive closure of $contains_{TG}$. Then we define
$C_{Cycle} = \{c \in C | \exists v_s \in clan(s(c)) \wedge \exists v_t \in clan(t(c)) : (v_s, v_t) \in equiv_{contains}\}$
as a subset of cycle-capable containment edges that might be part of containment cycles.
Type graph TG is called sound*, if there is a type node $r \in N - A$, called* root type*, such that the following holds: $\forall n \in N - A - \{r\} : (r, n) \in contains_{TG}$*

Example 1 (EMF model for simplified statecharts). In the running example, we consider the refactoring of a simplified form of statecharts. In Fig. 1 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 $State$. Since $State$ inherits from $Vertex$, it is again contained in $State$. Due to the same reason, $Pseudostate$ is contained in $State$. Moreover, $Transition$ and $FinalState$ are contained in $State$. 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 $State$ 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 $State$ (resp. $FinalState$) may contain $States$ (resp. $FinalStates$) and form containment cycles.

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

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

Please note that the multiplicities shown in Fig. 1 are not formalized by type graphs, but have to be expressed by additional graph constraints [11] (not shown here).

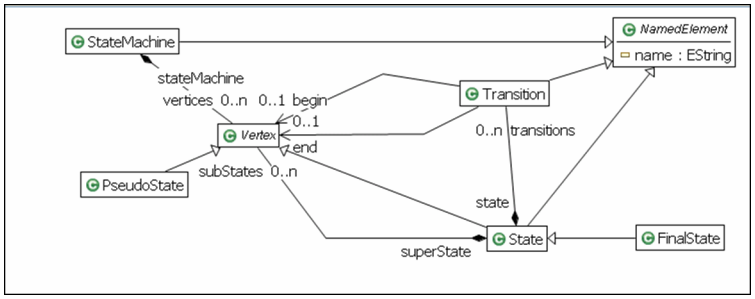

Fig. 1. EMF model for simplified statecharts

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). *Given a type graph $TG = (T, I, A, C)$ and a graph G. Graph G is typed over TG, written $type\colon G \to TG$, if for all $e \in G_E$ holds*

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

$type$ is called a clan morphism. It 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 $type\colon G \to TG$ which is consistent with containment, i.e. $\forall e \in G_C : type_{G_E}(e) \in C$

Please note that type graph TG is no 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. 2 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*.

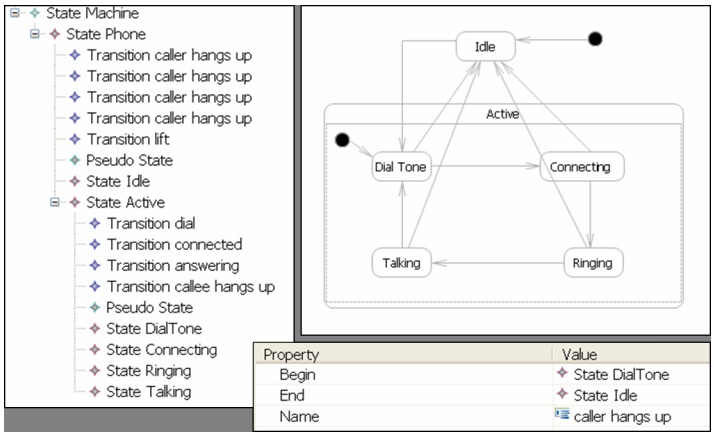

Fig. 2. EMF instance model of a simple phone

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 [5]. For the application of graph transformation theory presented in Section 5, attributes will play a minor role.

3 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. Delete an object node with its containment relation.
2. Create a new object node and connect it immediately to its container.
3. Delete a containment relation together with its contained object node or change the container.

4. Create a containment relation with the contained object node or change the container.
5. Create cycle-capable containment edges only, if the old and the new container 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. 7 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 clan 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* $(N_i, type_{N_i}) \in NAC$

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 $R'_C := R_C - K_C, L'_N := L_N - K_N$ and $L'_C := L_C - K_C$. A graph rule $p = (L \supseteq K \subseteq R,\ type,\ NAC)$ is* consistent *wrt. containment if the following constraints are satisfied:*

1. *(node deletion)* $\forall n \in L'_N : \exists e \in L'_C$ *with* $t_{L'_C}(e) = n$,
2. *(node creation)* $\forall n \in R'_N : \exists e \in R'_C$ *with* $t_R(e) = n$,
3. *(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* $t_R(e') = n)$
4. *(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* $t_L(e') = n)$
5. *(creation of cycle-capable containment edges)*
   $\forall e \in R'_C$ *with* $s(e) = n \wedge t(e) = m : \exists e' \in L'_C$ *with* $s(e') = o \wedge t(e') = m :$
   $((o, n) \in contains_L \wedge (m, n) \notin contains_L) \vee (n, o) \in contains_L$

Example 3 (Refactoring rules). In Figs. 3 to 5, 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. 3 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. 9). This rule is consistent, since the isolated state node is deleted together with its containment edge. Two transitions with

Fig. 3. Refactoring "Remove isolated state"

the same source, the same target, and the same name are considered as redundant and can be removed by the graph rule in Fig. 4. This rule is consistent, since the redundant transition node is deleted together with its containment edge. The third refactoring

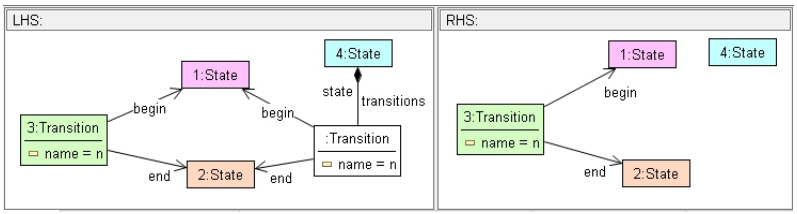

Fig. 4. Refactoring "Delete redundant transition"

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. 5 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 the same name. The application of both rules together allows folding of an arbitrary number of such transitions. The upper rule in Fig. 5 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. 5 is also consistent, since a transition is deleted together with its containment edge.

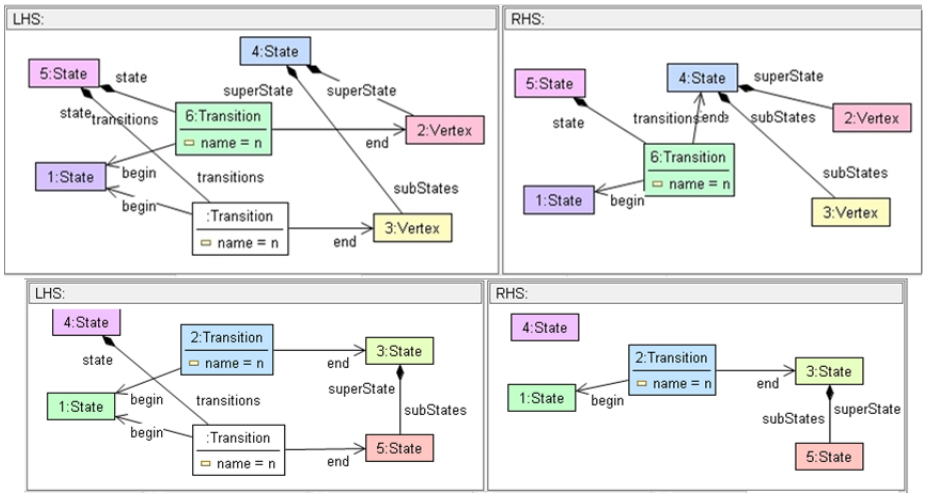

Fig. 5. Refactoring "Fold incoming transitions"

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 the surrounding state. This condition is checked by two similar NACs, one of which is shown together with the rule in Fig. 6. 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

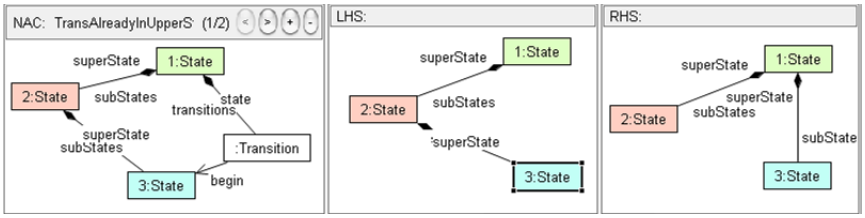

Fig. 6. Refactoring "Move State Up"

containment edge is deleted and a new one is created. Therefore, cond. 3 and 4 of Def. 8 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.

4 EMF Model Transformation as Consistent Graph Transformation Sequence

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 main 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. 9) 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 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 typed over TG, $(G, type_G)$ 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 $(G, type_G)$ yielding a direct transformation $(G, type_G) \overset{p,m}{\Longrightarrow} (H, type_H)$ with concretely typed graph $(H, type_H)$ defined as in [5].

The transformation definition in [5] 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. 7 shows the main parts of the simple statechart in Fig. 2 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*.

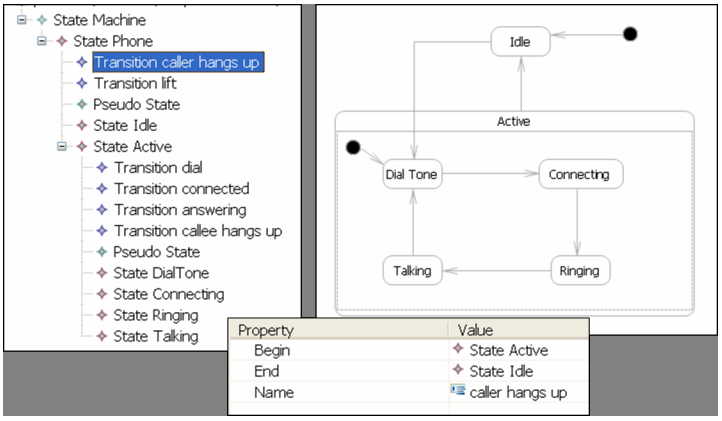

Fig. 7. EMF instance model of Fig. 2 after refactoring

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

Proof. See Section A.1 in the appendix.

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. See Section A.2 in the appendix.

5 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 [5] 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.

5.1 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 [1] 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 5 (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 [9], 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. 6 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. 8 shows the critical pair for this conflict[6]. This conflict can be resolved by moving inner states up at first.

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

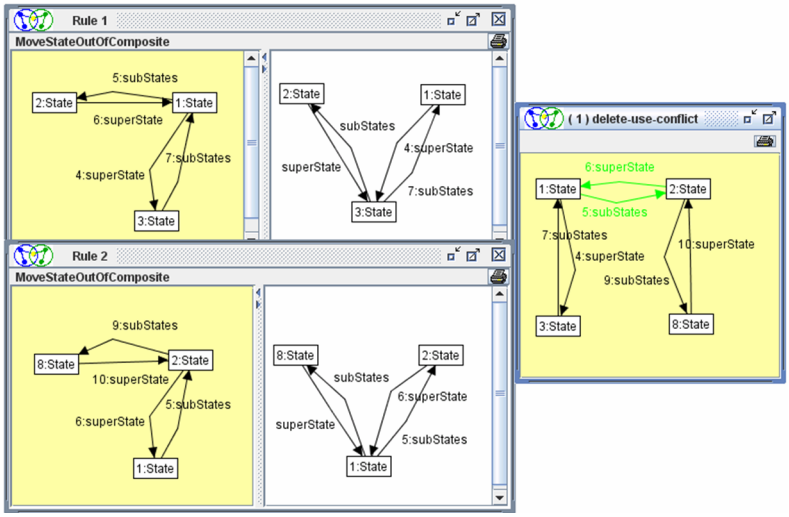

Fig. 8. Conflicting applications of Rule *MoveStateUp* two times

5.2 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 [5] 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 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 3, 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.

6 Related Work and 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 contained 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.

There are a number of model transformation engines which can modify EMF models: ATL [7], Tefkat [8], VIATRA2 [12], MOMEMT [3], etc. Each of these projects can be used to specify model transformations such as the statechart refactorings presented. In contrast to most model transformation engines, MOMEMT has a formal basis given by Maude which might be exploited for validation of EMF model transformations. But to the best of our knowledge, none of the existing transformation approaches supports confluence and termination analysis of EMF model transformations yet. The EMF Transformation Framework [2] currently supports the generation of transformation code in Java and the translation of EMF models and rules to AGG [1], 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.

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. AGG Homepage. `http://tfs.cs.tu-berlin.de/agg`.
2. E. Biermann, K. Ehrig, C. Köhler, G. Kuhns, G. Taentzer, and E. Weiss. Graphical Definition of In-Place Transformations in the Eclipse Modeling Framework. In *Proc. 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS'06)*, Genova, Italy, October 2006.
3. A. Boronat. *MOMENT: A Formal Framework for Model Management*. PhD thesis, Universtat Politècnica de València, 2007.
4. Eclipse Consortium. *Eclipse Modeling Framework (EMF) – Version 2.4*, 2008. `http://www.eclipse.org/emf`.
5. H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. EATCS Monographs in Theoretical Computer Science. Springer Verlag, 2006.
6. Eclipse model development tools. `http://www.eclipse.org/modeling/mdt`, 2007.

7. F. Jouault and I. Kurtev. Transforming Models with ATL. In *Proc. Workshop Model Transformations in Practice (at MoDELS'05)*, Montego Bay, Jamaica, 2005.
8. M. Lawley and J. Steel. Practical Declarative Model Transformation With Tefkat. In *Proc. Workshop Model Transformation in Practice (at MoDELS'05)*, 2005.
9. T. Mens, G. Taentzer, and O. Runge. Analysing refactoring dependencies using graph transformation. *Software and System Modeling*, 6(3):269–285, September 2007.
10. T. Mens and T. Tourwé. A survey of software refactoring. *Transactions on Software Engineering*, 30(2):126–139, February 2004.
11. A. Rensink and G. Taentzer. Ensuring structural constraints in graph-based models with type inheritance. In *Proc. Fundamental Approaches to Software Engineering (FASE)*, volume 3442 of *Lecture Notes in Computer Science*, pages 64–79. Springer Verlag, 2005.
12. D. Varró and A. Balogh. The model transformation language of the viatra2 framework. *Sci. Comput. Program.*, 68(3):214–234, 2007.
13. MOF 2.0 / XMI Mapping Specification. `http://www.omg.org/technology/documents/formal/xmi.htm`, 2008.

A Proofs

A.1 Proof of Theorem 1

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. 8):
$t(e'_2) = t(e_2) = t(e_1) \implies e_1 = e'_2 \in L'_C$ 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. 8, 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. 8, 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 $\lightning$.

A.2 Proof of Theorem 2

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. 8. 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. 8, 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. 8, a new containment path $n \xrightarrow{*} y$ is created. According to Def. 8, 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 $\lightning$.