Generating Instance Models from Meta Models

Karsten Ehrig[1], Jochen M. Küster[2], Gabriele Taentzer[3], and Jessica Winkelmann[3]

[1] Department of Computer Science, University of Leicester, UK,
karsten@mcs.le.ac.uk
[2] IBM Zurich Research Laboratory, CH-8803 Rüschlikon, Switzerland,
jku@zurich.ibm.com
[3] Department of Computer Science, Technical University of Berlin, Germany,
{gabi,danye}@cs.tu-berlin.de

Abstract. Meta modeling is a wide-spread technique to define visual languages, with the UML being the most prominent one. Despite several advantages of meta modeling such as ease of use, the meta modeling approach has one disadvantage: It is not constructive i. e. it does not offer a direct means of generating instances of the language. This disadvantage poses a severe limitation for certain applications. For example, when developing model transformations, it is desirable to have enough valid instance models available for large-scale testing. Producing such a large set by hand is tedious. In the related problem of compiler testing, a string grammar together with a simple generation algorithm is typically used to produce words of the language automatically. In this paper, we introduce instance-generating graph grammars for creating instances of meta models, thereby overcoming the main deficit of the meta modeling approach for defining languages.

1 Introduction

With models expressed in the Unified Modeling Language (UML) [14] becoming widely used in software engineering, also the meta modeling approach to define the syntax of modeling languages has gained a wide acceptance: Commonly, a meta model is designed which defines the abstract syntax of the language in a declarative way. Instantiation of the meta model then yields a concrete model.

The meta modeling approach has several advantages, one of them being that a visual meta model allows a quick grasp of the concepts being defined. Further, the meta modeling approach is also beneficial when it comes to defining complex modeling languages, consisting of several individual models. Nevertheless, there exists also one disadvantage: Whereas constructing words of a language defined by a string grammar can easily be done by applying grammar derivations, meta model instantiation is hard to operationalize.

In common applications of the UML, this does not pose a problem because the process of instantiation is performed by the software engineer when constructing models. However, there are certain applications when an automatic approach is needed: In compiler testing [4], the generation of a large amount

of models from a context-free grammar is common practice and a key issue in being able to test compilers automatically. Whereas until now such a problem could be neglected in model engineering based on the meta modeling approach, this situation drastically changes with the idea of model driven architecture [13] and the more widespread usage of model transformations. For testing model transformations, a large set of automatically generated instance models must be available in order to ensure the quality of the model transformation developed. Another area requiring an operational description of a language defined by a meta model is automatic editor generation for domain specific languages.

Graph grammars [5] provide a constructive, well-studied approach to language definition with a formal foundation that allows to prove important properties. However, the relationship between meta models and graph grammars has not been studied in depth so far, but started in [3]. Deriving an *instance-generating* graph grammar from an existing meta model is complicated. Here, one has to ensure that every model that is created by a derivation of the graph grammar is a valid instance of the meta model and further it is desirable that for every instance of the meta model there exists a derivation in the graph grammar. This completeness of the instance-generating graph grammar is important for model transformation testing because it allows a complete coverage of all possible inputs. For editor generation, it ensures that the language defined by the meta model is indeed the one supported by the editor.

In this paper, we present our approach for automatic derivation of instance-generating graph grammars from meta models. We first introduce meta models in Section 2 and graph transformation in Section 3. In Section 4, we explain how an instance-generating graph grammar can be derived for a meta model containing all main features. OCL constraints are not yet considered during the generation process, but have to be checked afterwards. Section 5 contains the proof that the derived graph grammar generates exactly those instances induced by the given meta model. As a consequence, the concept of the instance-generating graph grammar allows to formally show the completeness of the generated instances. We conclude by a discussion of related and future work.

2 Metamodels with OCL-Constraints

Visual languages such as the UML [14] are commonly defined using a meta modeling approach. In this approach, a visual language is defined using a meta model to describe the abstract syntax of the language. A meta model can be considered as a class diagram on the metalevel, i. e. it contains meta classes, meta associations and cardinality constraints. Further features include special kinds of associations such as aggregation, composition and inheritance as well as abstract meta classes which cannot be instantiated.

The instance of the meta model must conform to the cardinality constraints. In addition, instances of meta models may be further restricted by the use of additional constraints specified in the Object Constraint Language (OCL) [15].

Figure 1 shows a slightly simplified statechart meta model (based on [14]) which will be used as running example. A state machine has one top CompositeState. A CompositeState contains a set of StateVertices where such a StateVertex can be either an InitialState or a State. Note that StateVertex and State are modeled as abstract classes. A State can be a SimpleState, a CompositeState or a FinalState. A Transition connects a source and a target state. Furthermore, an Event and an Action may be associated to a transition. Aggregations and compositions have been simplified to an association in our approach but they could be treated separately as well. For clarity, we hide association names, but show only role names in Figure 1. The association names between classes StateVertex and Transition are called source and target as corresponding role names. The names of all other associations are equal to their corresponding role names. Since we want to concentrate on the main concepts of meta models here, we do not consider attributes in our example. Having an instance at hand, it is straight forward to generate attribute values in a post processing step.

The set of instances of the meta model can be restricted by additional OCL constraints. For the simplified statecharts example at least the following OCL constraints are needed:

1. A final state cannot have any outgoing transitions:
   context FinalState inv: self.outgoing->size()=0
2. A final state has at least one incoming transition:
   context FinalState inv:
   self.incoming->size()>=1
3. An initial state cannot have any incoming transitions:
   context InitialState inv: self.incoming->size()=0
4. Transitions outgoing InitialStates must always target a State:
   context Transition inv:
   self.source.oclIsTypeOf(InitialState) implies
   self.target.oclIsKindOf(State)

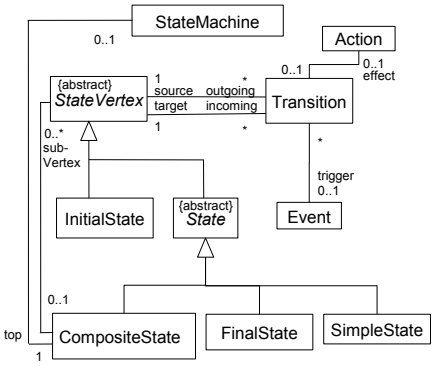

Fig. 1. Meta Model for statecharts

The complexity of generating instances of meta models crucially depends on the language elements used within meta models. For simple meta models without any constraints (not even multiplicity constraints) and inheritance, instantiation is rather straightforward by creating instances of metaclasses and associations. However, meta models as commonly used in language specification documents such as [14] make heavily use of multiplicity and OCL constraints as well as inheritance and abstract classes. For instantiation of such meta models, more sophisticated techniques are needed. In particular, there is a need for a systematic derivation of instances of meta models. In the following, we will describe the concepts of graph transformation which will represent the formal basis of our approach (inspired by the use of context-free grammars for deriving textual languages).

3 Graph Transformation

In this section we present the formal theory of *typed graph transformations with inheritance* (see [3]), which will be the basis for the formal background for *Instance Generating Graph Grammars (IGGG)* in Section 5.

In object-oriented modeling, graphs can be used at two levels: the type level (a class diagram) and the instance level (an instance of the class diagram). This typing concept has been described by *typed graphs* [5], where a fixed *type graph* serves as abstract representation of the class diagram. As in object-oriented modelling, types can be attributed and structured by an inheritance relation. Types should be divided into abstract types which cannot have instances and concrete types. Instances of a *type graph with inheritance (TGI)* are object graphs equipped with a structure-preserving mapping to the type graph. A class diagram can thus be represented by a type graph with inheritance plus a set of constraints over this type graph expressing multiplicities. For examples of the following definitions we refer to Section 4.

Definition 1 (type graph with inheritance). *A type graph with inheritance is a triple* $TGI = (TG, I, Abs)$ *consisting of a type graph* $TG = (TG_V, TG_E, src_{TG}, tgt_{TG})$ *(with a set* TG_V *of nodes, a set* TG_E *of edges, source and target functions* $src_{TG}, tgt_{TG} : TG_E \to TG_V$*), an acyclic inheritance relation* $I \subseteq TG_V \times TG_V$*, and a set* $Abs \subseteq TG_V$*, called abstract nodes. For each* $x \in TG_V$*, the* inheritance clan *is defined by* $clan_I(x) = \{y \in TG_V \mid (y, x) \in I^*\}$*, where* I^* *is the reflexive-transitive closure of* I*.*

A graph can be typed over the type graph with inheritance by a pair of functions, from nodes to node types and from edges to edge types, respectively. This pair of functions does not constitute a graph morphism, but will be called *clan morphism*; it uniquely characterizes the type morphism into the flattened type graph.

Definition 2 (clan morphism). *Let* $TGI = (TG, I, Abs)$ *with* $TG = (TG_V, TG_E, src_{TG}, tgt_{TG})$ *be a type graph with inheritance. A* clan-morphism $ctp : G \to TGI$ *from a graph* $G = (G_V, G_E, src_G, tgt_G)$ *to* TGI *is a pair* $ctp = (ctp_V : G_V \to TG_V, ctp_E : G_E \to TG_E)$ *such that for all* $e \in G_E$ *the following holds:*

- $ctp_V \circ src_G(e) \in clan_I(src_{TG} \circ ctp_E(e))$ *and*
- $ctp_V \circ tgt_G(e) \in clan_I(tgt_{TG} \circ ctp_E(e))$.

(G, ctp) *is called a* clan-typed graph.

The main ingredients of graph grammars are graph rules which will be defined in Definition 4. Between clan-typed graphs we use type-refining morphisms (see also Def. 5 in [16]) where a node with type t can be mapped to a node with a type in $clan(t)$. In the following, we call a type-refining morphism just morphism. If each node is mapped to a node with the same type, the corresponding morphism is called type-preserving.

For controlling a rule application, simple negative application conditions $NAC(x)$ and atomic application conditions $P(x, \wedge_{i\in I} x_i))$ are defined which are needed in Section 4. Although $NAC(x)$ is a special case of $P(x, \wedge_{i\in I} x_i))$ with $I = \emptyset$, we introduce both kinds of application conditions, due to more clear definition of instance generating rules.

Definition 3 (application condition). *A* simple negative application condition *is of the form $NAC(x)$, where $x : L \to X$ is an injective morphism. A morphism $m : L \to G$* satisfies *$NAC(x)$ if there does not exist an injective morphism $p : X \to G$ with $p \circ x = m$. An* atomic application condition *is of the form $P(x, \wedge_{i\in I} x_i)$ where $x : L \to X$ and $x_i : X \to C_i$ with $i \in I$ are injective morphisms. A morphism $m : L \to G$* satisfies *$P(x, \wedge_{i\in I} x_i)$ if for all injective morphisms $p : X \to G$ with $p \circ x = m$ there does exist an $i \in I$ and an injective morphism $q_i : C_i \to G$ with $q_i \circ x_i = p$.*

Definition 4 (rules). *A rule typed over a type graph $TGI = (TG, I, Abs)$ with inheritance is given by $p = (L \xleftarrow{l} K \xrightarrow{r} R, A_p)$, where L, K, R are clan-typed graphs, l and r are type-preserving injective graph morphisms, $ctp_R^{-1}(Abs) \subseteq r(K_V)$, and A_p is a set of application conditions of the form $NAC(x)$ or $P(x, \wedge_{i\in I} x_i)$ as defined in Def. 3.*

Definition 5 (rule matching and application). *Given a rule p as in Def. 4 and a clan-typed graph (G, ctp_G), then m is a match of p in G if*

- *m is an injective match of the rule $p = (L \xleftarrow{l} K \xrightarrow{r} R, A_p)$ as defined in Def. 4 in the graph G;*
- *$t_K(x_1) = t_K(x_2)$ for $t_K = ctp_G \circ m \circ l$ and $x_1, x_2 \in K_V$ with $r(x_1) = r(x_2)$;*
- *m satisfies all simple negative application conditions and all atomic applications in A_p.*

Given a match m, a direct derivation $(G, ctp_G) \overset{p,m}{\Longrightarrow} (H, ctp_H)$ exists if there is a span of graph morphisms $G \leftarrow D \rightarrow H$ and a co-match $m^* : R \to H$ of p in H that give rise to a derivation in the double-pushout approach of untyped graph transformation as defined in [5] where pushouts are used to model the gluing of graphs.

Given a rule set R, $(G, ctp_G) \overset{*}{\Rightarrow}_R (H, ctp_H)$ is a finite sequence of an arbitrary number of direct derivations by rules of R. A derivation $(G, ctp_G) \overset{*}{\Rightarrow}_R (H, ctp_H)$ terminates, if $\nexists r \in R : (H, ctp_H) \Rightarrow_r (H', ctp_{H'})$.

4 Generating Instances by Graph Grammars

In this section, we introduce the idea of an instance-generating graph grammar that allows one to derive instances of an arbitrary meta model in a systematic way. The corresponding graph grammar requires (1) a start graph that will be the empty graph, (2) a type graph that is obtained by converting the meta model class diagram to a type graph and (3) graph grammar rules which are described below.

We use the concept of layered graph grammars [6] to order rule applications. Layer 1 rules create instances of each class. To generate all possible instances we have to allow an arbitrary number of applications of these rules, meaning that Layer 1 does not terminate and has to be interrupted by user interaction or after a random time period. Layer 2 rules deal with generating links corresponding to associations with at least one *1*-multiplicity. Those rules have to be applied as long as possible to ensure the multiplicity constraints, requiring that rule application in this layer has to terminate. Layer 3 creates links corresponding to associations with *0..n*-multiplicities. The rules in this layer can be applied arbitrarily often because these links are optional.

We use abstract node types (corresponding to abstract classes) leading to the concept of abstract rules. An abstract rule contains at least one node of abstract type. For each concrete subtype of the abstract type this induces a corresponding rule.

Given a concrete meta model, assembling the rules derived, the type graph created and the empty start graph leads to an instance-generating graph grammar for this meta model. The rules of the instance-generating graph grammar are determined by the occurrence of specific meta model patterns: The idea is to associate to a specific meta model pattern a graph grammar rule that creates an instance of the meta model pattern under certain conditions. In the following, we describe the rules that we derive for common meta model patterns.

Instance-generating rules: Layer 1 of any instance-generating graph grammar (see pattern p_0 in Figure 2) contains rules of the form createE' where E' is replaced by the name of any non-abstract class. The meta model pattern for this rule is simply a class. For a concrete meta model, we will get such a create rule for each non-abstract class within the meta model, allowing us to create an arbitrary number of instances of all non-abstract classes.

We have three meta model patterns for the rules in Layer 2 (corresponding to the three possible multiplicity constraints) (see Fig. 3 and 4). The first rule for each pattern creates a link between existing instances. The NACs ensure, that the created link does not violate the multiplicity constraints (e.g. the two instances are not already connected by such a link, or the instance of A is not already connected to an instance of E).

To ensure the *to one* multiplicity on the specified association ends insertE'_a_ANewObj resp. insertE'_a_ANewObj2 creates a new instance of any concrete E' $\in$ *clan*(E) resp. A' $\in$ *clan*(A) if no application condition holds. In case of a 1 to * relation (see pattern p_1) a new instance of E' $\in$ *clan*(E) is created if no concrete instance of E is present, which is ensured by NAC_1. In case of a 1 to 0..1 or 1 to 1 relation (see pattern p_2 and p_3) the rule can only be applied if any match of an instance of E is already connected to an instance of A, which is ensured by the application condition. NAC_2 of the rules insertE'_a_ANewObj resp. insertE'_a_ANewObj2 requires that the instance of A is not connected to an instance of E yet.

We also have three meta model patterns for the rules of Layer 3 (corresponding to the three possible multiplicity constraints) (see Fig. 5). The rules for these

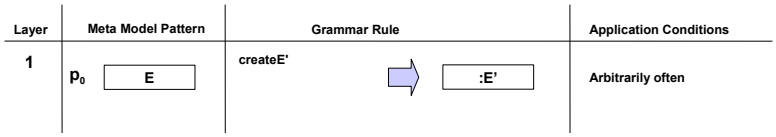

Fig. 2. Rules for graph grammar derivation: Layer 1

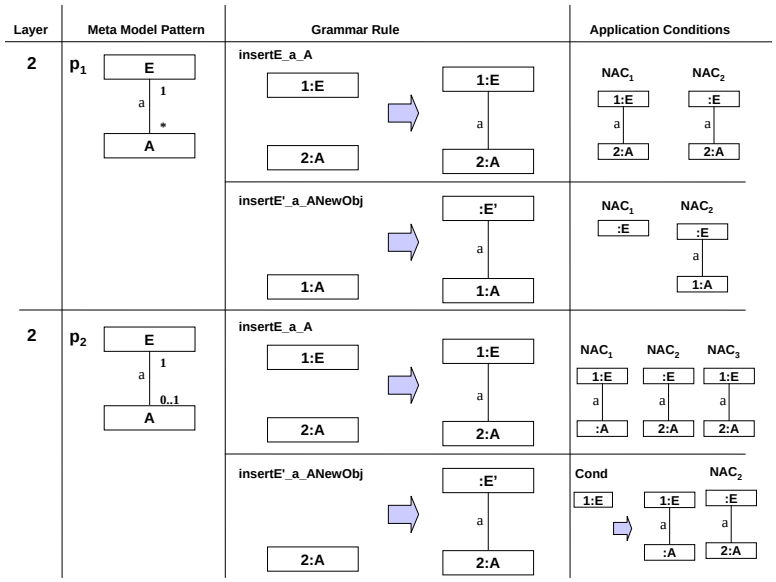

Fig. 3. Rules for graph grammar derivation: Layer 2

patterns create links between existing instances. The NACs ensure, that the created link does not violate the upper multiplicity constraints as in the first rules of the corresponding pattern in Layer 2. The graph grammar derivation rules in layer 3 can be applied *arbitrarily often*, they are terminating as described above.

Generating Statechart Instances: We now discuss an instance-generating graph grammar for the meta model of statecharts (see Figure 1). Due to space limitation we do not show the details of all rules. The example rules shown in Figure 6 - 8 construct a simple instance graph consisting of a state machine with its top CompositeState containing three state vertices and two transitions between them. In the application conditions shown in Figures 6 - 8 the node types are abbreviated (CS for CompositeState etc.).
First, we get Layer 1 rules for all concrete classes occurring in the class diagram. These are createStateMachine, createCompositeState, createSimpleState, createFinalState, createInitialState, createTransition, createEvent, and createAction.
For association source between StateVertex and Transition (corresponding to an instance of pattern p_1), we derive four rules: one rule creates a link source between an existing StateVertex and an existing Transition. Further, for each concrete class that inherits from class StateVertex one rule is derived that creates the StateV-

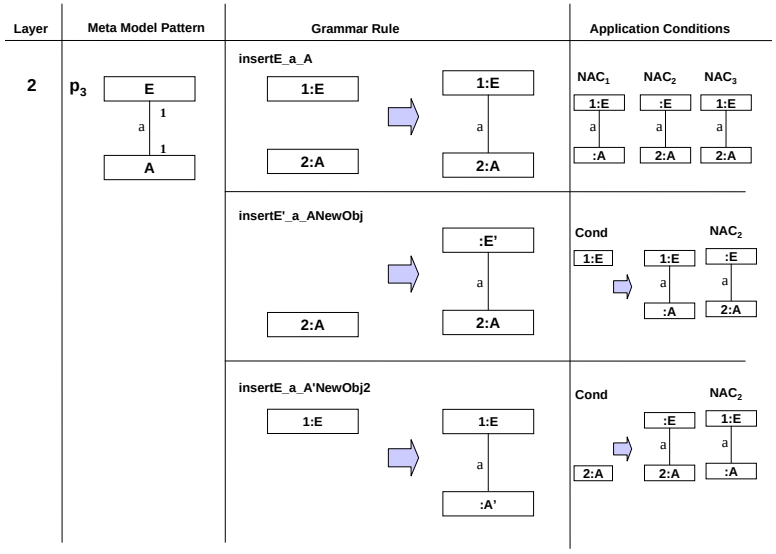

Fig. 4. Rules for graph grammar derivation: Layer 2

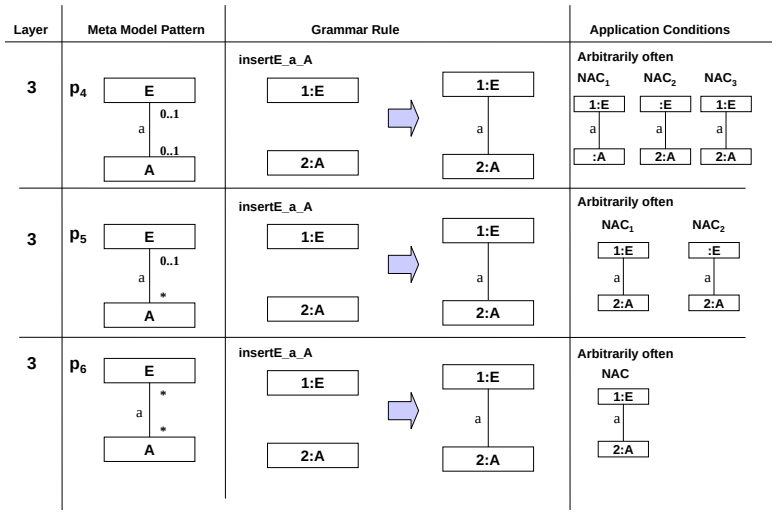

Fig. 5. Rules for graph grammar derivation: Layer 3

ertex, an InitialState, a CompositeState, SimpleState or a FinalState, and the link source. Note that the abstract class StateVertex could be matched to any of its concrete subclasses InitialState, CompositeState, FinalState, and SimpleState. For association target between StateVertex and Transition, similar rules are derived. For association top between StateMachine and CompositeState, an instance of

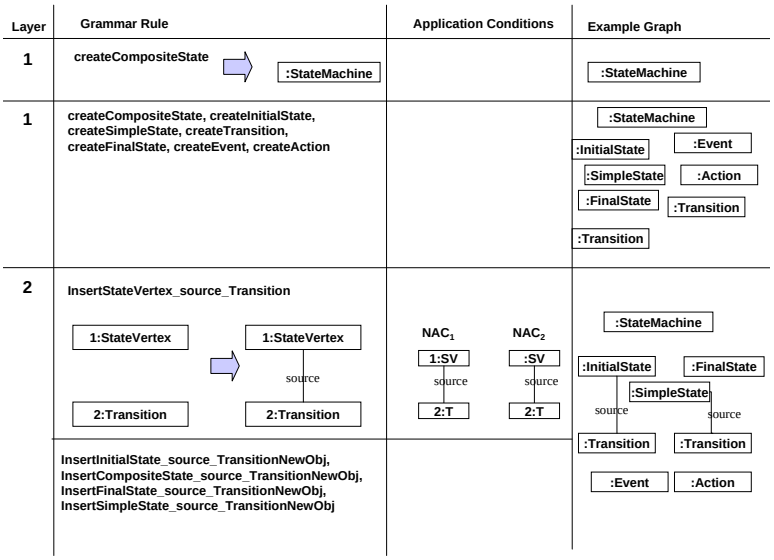

Fig. 6. Example Grammar Rules 1

pattern p_2, we derive the corresponding two rules. One of them is shown in Figure 6, creating a CompositeState to a StateMachine if each other CompositeState is bound and the StateMachine is not already connected to a top CompositeState.

We further get instances of pattern p_4 (association between Transition and Action) and p_5 (association between Transition and Event as well as association between CompositeState and StateVertex).

Extensions: So far, we considered a generation of meta model instances that is somewhat simplified: First of all, we have not explicitly dealt with generating attribute values. There are (at least) two possible solutions for this: One possibility is to perform a postprocessing step which generates arbitrary attribute values. A set of predefined values is specified for each attribute, to be used within attribute assignment. Another approach would be to explicitly include attributes in the graph grammar rules and assign attributes already while deriving the instance of the meta model. Also properties of associations like navigation directions, role names, etc. can be included in certain attributes.

Then, associations being loops as well as associations with arbitrary cardinality constraints (i. e. $m..n$) can be achieved by extending the rule set of the instance generating graph grammar. Moreover if the meta model contains singleton classes, the create rule for the corresponding class has to have an additional application condition that ensures that at most one instance of this class is created.

Ensuring OCL constraints can be done by a constraint checker, once the overall derivation of an instance model has terminated. The instance generation

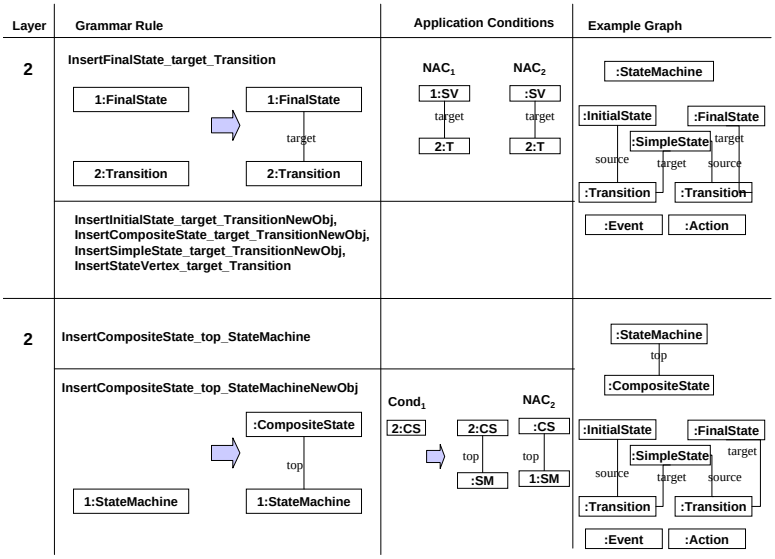

Fig. 7. Example Grammar Rules 2

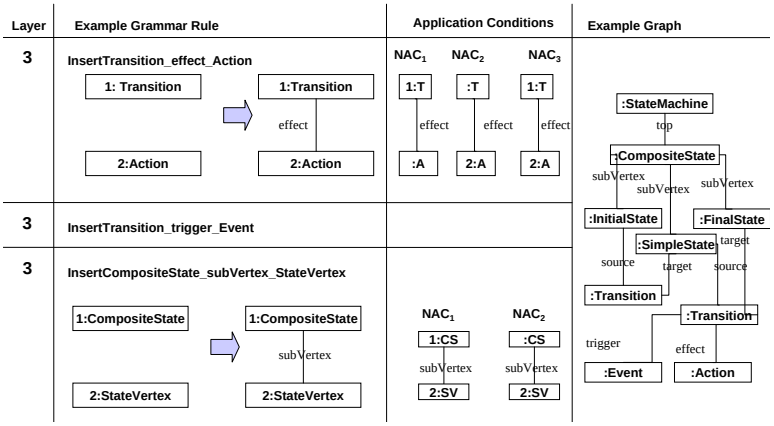

Fig. 8. Example Grammar Rules 3

and the translation of simple OCL constraints are described in [7,8] in more detail.

5 Formal Background for Instance Generating Graph Grammars

In this section we present the formal background for *Instance Generating Graph Grammars (IGGG)* based on the formal theory of typed graph transformations with inheritance (see [3]). As the main result of this paper, we present the equiv-

alence of instance sets generated by an instance-generating graph grammar on the one hand, and induced by a type graph with multiplicities on the other hand.

Definition 6 (multiplicities). *A multiplicity is a pair* $[i,j] \in \mathcal{N} \times (\mathcal{N} \cup \{*\})$ *with* $i \leq j$ *or* $j = *$. *The set of multiplicities is denoted Mult. The special value* $*$ *indicates that the maximum number of nodes or edges is not constrained. For an arbitrary finite set* X *and* $[i,j] \in Mult$, *we write* $|X| \in [i,j]$ *if* $i \leq |X|$ *and either* $j = *$ *or* $|X| \leq j$.

Now we define an induced graph language over a type graph with multiplicities TGI_{mult}. As usual, we use multiplicities to decorate the edges of type graphs. The multiplicities express the number of incoming, respectively outgoing edges for each target, respectively source instance.

Definition 7 (Type graph with multiplicities). *A type graph with multiplicities (see [16]) is a tuple* $TG_{mult} = (TGI, m_{src}, m_{tgt})$ *consisting of a type graph with inheritance TGI and additional functions* $m_{src}, m_{tgt} : TGI_E \to Mult$, *called* edge multiplicity functions.

Considering the meta model in Figure 1, it can be formalized to a type graph with multiplicities in a straight forward way. The node types are given by classes, the edge types by associations. In contrast to the associations, edge types have to be always directed. For each edge type a direction can be arbitrarily chosen.

Definition 8 (TGI_{mult}-induced graph language). *Given a type graph* TGI_{mult} *with multiplicities as defined in Def. 7, the induced* graph language *is defined by:*
$L(TGI_{mult}) = \{(G = (G_V, G_E, src_G, tgt_G), ctp_G : G \to TGI) \mid$
$\forall e \in TGI_E \wedge \forall v \in ctp_G^{-1}(t)$ *with* $t \in clan(src(e)) : |ctp_G^{-1}(e) \cap src^{-1}(v)| \in m_{tgt}(e)$
and
$\forall e \in TGI_E \wedge \forall v \in ctp_G^{-1}(t)$ *with* $t \in clan(tgt(e)) : |ctp_G^{-1}(e) \cap tgt^{-1}(v)| \in m_{src}(e)\}$, *where* ctp_G *is a clan morphism.*

Example 1. Considering e.g. the example graph in Fig. 8, the multiplicities for edge type subvertex are fulfilled: For the only composite state c $|ctp^{-1}(\mathsf{subvertex}) \cap src^{-1}(c)| = 3 \in [0,*]$ and for all state vertices s $|ctp^{-1}(\mathsf{subvertex}) \cap tgt^{-1}(s)| \leq 1 \in [0,1]$. The composite state is not subvertex of any vertex and all other state vertices are subvertex of the composite state.

Having formalized a meta model given by a class diagram through a type graph with multiplicities, we are now ready to define the language of an instance-generating graph grammar. Based on a given type graph with multiplicities, we mainly formalize the set of rules needed for instance generation. The rules are already given in Sec. 4. Please note that rules insertE_a_A and insertE'_a_ANewObj differ dependently on the source and target multiplicities of the corresponding patterns.

Since all given rules are intended to be matched injectively, they do not capture the case of patterns with loops as edge types, which would be translated to loops in the type graph. That's why loops are excluded in the following.

Definition 9 (instance-generating graph grammar and language). *Given a type graph TGI_{mult} with multiplicities as in Def. 7 without loops, an instance generating graph grammar is denoted by $IGGG = (TGI, \emptyset, R)$, where R is the union of the following sets of rules. The rules are depicted in Figures 2 - 5 and are formalized in the obvious way according to Def. 4.*

- $R_1 = \{\mathsf{createE'} \mid \forall E' \in TGI_N \wedge E' \notin Abs\}$ *with rules* createE' *as in Fig. 2*
- $R_2 = R_{21} \cup R_{22} \cup R_{23}$ *with*
  $R_{21} = \{\mathsf{insertE\_a\_A} \mid \forall A, E \in TGI_N, a \in TGI_E :$ *with*
  $(m_{src}(a) = [1,1] \vee m_{tgt}(a) = [1,1])\}$
  $R_{22} = \{\mathsf{insertE'\_a\_ANewObj} \mid \forall A, E \in TGI_N, a \in TGI_E :$ *with*
  $(m_{src}(a) = [1,1] \vee m_{tgt}(a) = [1,1]) \wedge E' \in clan(E) \wedge E' \notin Abs\}$
  $R_{23} = \{\mathsf{insertE\_a\_A'NewObj2} \mid \forall A, E \in TGI_N, a \in TGI_E :$ *with*
  $(m_{src}(a) = [1,1] \vee m_{tgt}(a) = [1,1]) \vee A' \in clan(A) \wedge A' \notin Abs\}$
  with rules insertE_a_A, insertE'_a_ANewObj, *and* insertE'_a_ANewObj2 *as in Fig. 3 - 4*
- $R_3 = \{\mathsf{insertE\_a\_A} \mid \forall A, E \in TGI_N, a \in TGI_E$ *with* $m_{src}(a) \neq [1,1] \wedge m_{tgt}(a) \neq [1,1]\}$ *with rules* insertE_a_A *as in Fig. 5*

R is layered, i.e. there is a function $rl : R \to \mathcal{N}$ with $rl(r) = i$ for all $r \in R_i$ for $i = \{1,2,3\}$. Function rl is called rule layer function.
The generated graph language *is defined by the following set of concrete typed graphs: $L(IGGG) = \{(G, ctp_G) \mid \emptyset \overset{*}{\Rightarrow}_{R_1} (H, ctp_H) \overset{*}{\Rightarrow}_{R_2} (K, ctp_K) \overset{*}{\Rightarrow}_{R_3} (G, ctp_G) \wedge \nexists r \in R_2 : (K, ctp_K) \Rightarrow_r (K', ctp_{K'})\}$.*

The following lemma states that the rule application of rules in R_2 to any graph created by rules of R_1 always terminates. This property is needed in the following theorem.

Lemma 1 (termination of rule layer 2). *Given an instance generating graph grammar $IGGG(TGI, \emptyset, R)$ where TGI does not contain any loop as edge type, let $L_1(IGGG) = \{(H, ctp_H) \mid \emptyset \overset{*}{\Rightarrow}_{R_1} (H, ctp_H)\}$. All derivation sequences $(H, ctp_H) \overset{*}{\Rightarrow}_{R_2} (G, ctp_G)$ with $(H, ctp_H) \in L_1(IGGG)$ terminate.*

Proof. See [7].

As one main result the following theorem states that the instance sets generated by an $IGGG$ and those induced by a type graph with multiplicities are equal.

Theorem 1 (equality of languages). *Given a type graph TGI_{mult} with multiplicities and without loops and an instance generating graph grammar $IGGG = (TGI, \emptyset, R)$ for TGI_{mult}, we have $L(IGGG) = L(TGI_{mult})$.*

Proofidea. We have to proof that
(1) $(G, ctp_G) \in L(TGI_{mult})$ holds for any derivation $\emptyset \overset{*}{\Rightarrow}_{R_1} (H, ctp_H) \overset{*}{\Rightarrow}_{R_2} (K, ctp_K) \overset{*}{\Rightarrow}_{R_3} (G, ctp_G)$. This is true, since Layer 1 creates nodes of valid types only, the NACs prohibit the exceeding of the upper bound, and the rules in Layer

2 are applied until the lower bounds are fulfilled.
(2) For a given graph $(G, ctp_G) \in L(TGI_{mult})$ there exists a derivation sequence $\emptyset \overset{*}{\Rightarrow} (G, ctp_G)$ over $IGGG$. We create the sequence by first creating all nodes by rules of Layer 1, and then creating the edges for each pattern. For the complete proof see [7].

6 Related Work

One closely related approach is the one by Alanen and Porres [2]: They describe two algorithms, one to derive a context-free grammar from a meta model and another one for deriving a meta model from a context-free grammar. However, their algorithm for grammar derivation can only deal with composite associations between metaclasses, restricting it to tree-like meta models which is a severe limitation for practical usage. Further, the algorithm does not support ordinary associations with arbitrary cardinalities. This limitation is not surprising given the properties of context-free grammars and represents one reason for the approach to use graph grammars instead of context-free grammars.

Another related problem is the one of automated snapshot generation for class diagrams for validation and testing purposes, tackled by Gogolla et al. [10]. In their approach, properties that the snapshot has to fulfill are specified in OCL. For each class and association, object and link generation procedures are specified using the language ASSL. In order to fulfill constraints and invariants, ASSL offers try and select commands which allow the search for an appropriate object and backtracking if constraints are not fulfilled. The overall approach allows snapshot generation taking into account invariants but also requires the explicit encoding of constraints in generation commands. As such, the problem tackled by automatic snapshot generation is different from the meta model to graph grammar translation.

Formal methods such as Alloy [1] can also be used for instance generation: After translating a class diagram to Alloy one can use the instance generation within Alloy to generate an instance or to show that no instances exist. This instance generation relies on the use of SAT solvers and can also enumerate all possible instances. In contrast to such an approach, our approach aims at the construction of a grammar for the metamodel and thus establishes a bridge between metamodel-based and grammar-based definition of visual languages.

In the area of pattern recognition, there have been several approaches to grammatical inference: Given a finite set of sample patterns, a grammar should be deduced such that the language generated by the grammar contains the sample patterns. Originally, this problem has been tackled where patterns are encoded as strings and regular grammars are generated [9]. In the context of graph grammars, Jeltsch and Kreowski [12] describe how a hyperedge replacement grammar can be derived from a finite set of graph samples. Our problem setting is slightly different because we are given a meta model to describe all instances and not only a finite set of samples.

Further (complementary) related work can be seen in the area of model-driven testing [11] where the aim is to use a model of the system to produce suitable test data. The problem of generating those instances from the grammar that provide a suitable coverage for testing can possibly benefit from existing research in this area.

7 Conclusion and Future Work

Currently, the widespread approach of defining visual languages has one main disadvantage: The systematic generation of instances of meta models is difficult to automate which poses limitations for e. g. automated testing of model transformations. In this paper, we have introduced the idea of instance-generating graph grammars which is basically the equivalent to a Chomsky grammar for textual languages.

On the basis of meta model patterns and corresponding derivation rules, our approach allows the construction of an instance-generating graph grammar for meta models without OCL constraints. This construction is based on a type graph with inheritance. As running example, we have constructed an IGGG for a simplified statechart meta model. Using the theory of typed graph transformation with inheritance, we have shown that the instance sets generated by an IGGG and those induced by the corresponding type graph with multiplicities are equal.

Automatic derivation of instances from meta models is a complex task which needs tool support. So far, we have automated the construction of an IGGG by providing a model transformation that automatically derives an IGGG from a meta model. For a complete description of this implementation we refer to the URL http://tfs.cs.tu-berlin.de/agg/MM2GraGra. Although the current model transformation does not support all features of meta models yet, it nevertheless shows the feasibility of our approach.

Future work should extend the automatic instance generation by meta models with OCL constraints. Ensuring OCL constraints can be done in two ways: One is to check constraints once the overall derivation of an instance model has terminated. However, this leads to the generation of a large number of non-valid instances. An approach avoiding the generation of invalid instances is presented in [7,8].

Further work is needed to apply our approach to testing model transformations: For that, techniques are needed that allow the generation of selected instance models that represent a suitable diversity of all possible models. Furthermore a syntax graph grammar could be generated from a meta model providing the basis for automatically generated visual editing rules.

References

1. *The Alloy Analyzer - 3.0 Beta http://alloy.mit.edu/*, 2000.

2. M. Alanen and I. Porres. A Relation Between Context-Free Grammars and Meta Object Facility Metamodels. Technical Report TUCS No 606, TUCS Turku Center for Computer Science, March 2003.
3. R. Bardohl, H. Ehrig, J. de Lara, and G. Taentzer. Integrating Meta Modelling with Graph Transformation for Efficient Visual Language Definition and Model Manipulation. In M. Wermelinger and T. Margaria-Steffens, editors, *Proc. Fundamental Aspects of Software Engineering 2004*, volume 2984. Springer LNCS, 2004.
4. A. S. Boujarwah and K. Saleh. Compiler test case generation methods: a survey and assessment. *Information and Software Technology*, 39(9):617–625, 1997.
5. A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Löwe. Algebraic Approaches to Graph Transformation Part I: Basic Concepts and Double Pushout Approach. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph transformation, Volume 1: Foundations*, pages 163–246. World Scientific, 1997.
6. H. Ehrig, K. Ehrig, J. de Lara, G. Taentzer, D. Varró, and S. Varró-Gyapay. Termination Criteria for Model Transformation. In M. Wermelinger and T. Margaria-Steffen, editors, *Proc. Fundamental Approaches to Software Engineering (FASE)*, volume 2984 of *Lecture Notes in Computer Science*, pages 214–228. Springer Verlag, 2005.
7. K. Ehrig, J. Küster, G. Taentzer, and J. Winkelmann. Automatically Generating Instances of Meta Models. Technical Report 2005–09, Technical University of Berlin, Dept. of Computer Science, November 2005.
8. K. Ehrig, J. Küster, G. Taentzer, and J. Winkelmann. Translation of Restricted OCL Constraints into Graph Constraints for Generating Meta Model Instances by Graph Grammars. In *Proc. (GT-VMT)*, 2006. To appear. A preliminary version of the proceedings is available at http://hobbit.inf.mit.bme.hu/GT-VMT2006/ProceedingsGTVMT2006.pdf.
9. K. S. Fu and T. L. Booth. Grammatical Inference: Introduction and Survey. *IEEE Transcations on Systems, Man, and Cybernetics*, SMC-5:95–111, 409–423, 1975.
10. M. Gogolla, J. Bohling, and M. Richters. Validating UML and OCL Models in USE by Automatic Snapshot Generation. *Software and Systems Modeling*, 2005. To appear.
11. A. Hartman and K. Nagin. Model Driven Testing - AGEDIS Architecture, Interfaces, and Tools. In *Proceedings 1st European Conference on Model-Driven Software Engineering*, 2003.
12. E. Jeltsch and H.-J. Kreowski. Grammatical Inference Based on Hyperedge Replacement. In Hartmut Ehrig, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *Proc. 4th. Int. Workshop on Graph Grammars and their Application to Computer Science*, volume 532 of *Lecture Notes in Computer Science*, pages 461–474. Springer-Verlag, 1991.
13. Object Management Group. *MDA Guide Version 1.0.1*, June 2003.
14. Object Management Group (OMG). *UML 2.0 Superstructure Final Adopted Specification. OMG document pts/03-08-02*, August 2003.
15. Object Management Group (OMG). *OCL 2.0 Specification. OMG document ptc/2005-06-06*, June 2005.
16. A. Rensink and G. Taentzer. Ensuring Structural Constraints in Graph-Based Models with Type Inheritance. In *Proc. Fundamental Approaches to Software Engineering (FASE)*, pages 64–79. LNCS 3442, Springer, 2005.