Electronic Communications of the EASST
Volume 47 (2012)

Proceedings of the 11th International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2012)

Instance Generation from Type Graphs with Arbitrary Multiplicities

Gabriele Taentzer

15 pages

Guest Editors: Andrew Fish, Leen Lambers
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Instance Generation from Type Graphs with Arbitrary Multiplicities

Gabriele Taentzer

Philipps-Universität Marburg, Germany

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 a major disadvantage: It does not offer a direct means for generating its language elements. This disadvantage poses a severe limitation on 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 formalize a restricted form of meta-models by type graphs with multiplicities and introduce instance-generating graph grammars for creating instance graphs representing the abstract syntax structures of models. Thereby, a further step is taken to overcome the main deficit of the meta-modeling approach.

Keywords: Meta modeling, graph transformation, model transformation

1 Introduction

Model-driven engineering (MDE) is a software engineering discipline that uses models as main artifacts throughout the software development process, adopting (automatic) model transformations for software development steps, model optimization and code generation. Models are written in domain-specific languages (DSLs), usually defined by meta-models. More concretely, a meta model defines the abstract syntax of a modeling language. The way a meta model defines a modeling language is a declarative one, i.e., it enumerates all the model elements with its attributes and its inter-relations. Moreover, well-formedness rules, usually written in OCL, can be used to specify invariants. 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. Furthermore, the meta-modeling approach is also beneficial when it comes to defining complex modeling languages, consisting of several sub-languages. Nevertheless, there is also a major 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, due to the declarative form of a meta model.

In common applications of DSLs, this does not pose a problem because the process of instantiation is performed by the software engineer when constructing models. However, there are a number of emerging applications where firstly an approach is needed for generating a large set of models automatically and secondly, the generation process must also be described explicitly.

In other words, instead of the declarative meta model an equivalent operational description of the language defined by the meta model is required.

Important applications of an operational definition approach for modeling languages include automated testing of model transformations [Küs06, MBT06]. Model-driven architecture favors a widespread usage of model transformations. The quality of model transformations is thus crucial for their successful and widespread usage. For automated testing of model transformations, the generation of instance models is required. However, not all elements of the input language are usually interesting test cases. Further work has to be done to restrict the test case generation to representative cases only. These restrictions could be formulated by additional OCL constraints to be taken into account for instance generation. Furthermore, result models have to be checked wrt. test assertions. At least, result models have to be elements of the result language.

Another kind of applications are editor generations for DSLs. Automatically generated editors for domain-specific languages being defined by meta-models only offer simple editor operations only. An operational language description may help also to automatically generate advanced editor operations: First of all, the application of simple operations does not always yield models of the specified DSL. Grammar rules shall be used to deduce quick fixes for indicated syntax errors. Moreover, there are certainly larger editing steps where several closely related model elements are edited together. Grammar rules show which model elements are closely related in the sense that editing them all or none of them form yields consistent models again. Even further, larger editing operations such as refactorings can be defined by composing several previously defined editing operations to new ones.

Graph grammars [Roz97] provide a constructive, well-studied approach to language definition with a formal foundation that allows to prove important properties. First approaches have emerged to generate graph grammars from restricted forms of meta-models. In [HM10] and [HM11], generalized hyperedge replacement grammars are constructed from a restricted form of class diagrams to generate instances. In [EKT09], we construct graph grammars over type graphs with inheritance for instance generation. Moreover, basic OCL constraints can be translated to graph constraints and further to application conditions of rules [WTEK08]. In this paper, a new form of instance generation graph grammar is presented, allowing meta-models to have arbitrary multiplicities. However, other restrictions, especially concerning supported kinds of well-formedness rules, still hold.

This work of translating the declarative specification of the language into an operational one can be seen as a foundational technique within model engineering because it will allow the adoption of techniques well-known for modeling languages and thereby closes an important conceptual gap. To achieve this aim, the instance generating graph grammar derived from a type graph formalizing a restricted form of meta-models has to generate all possible instances of the type graph and should not generate any graph that is not an instance of the type. In terms of graph grammar derivations, one has to ensure that every graph being created by a derivation of the graph grammar, is a valid instance of the type graph and furthermore that for every instance of the type graph there exists a derivation in the graph grammar. This completeness of the instance generating graph grammar is important for their applications.

This paper is organized as follows: Type graphs with multiplicities are recalled in Section 2 and function as formalization of a restricted form of meta-models. Graph transformation concepts based on type graph with inheritance are recalled in Section 3. Thereafter, we are ready to

present instance generating graph grammars for the generalized case of arbitrary multiplicities in Section 4. Finally, we conclude with a short consideration of related work, summary, and outlook.

2 Type graphs with multiplicities and their languages

In this section, we recall type graphs with inheritance as introduced in [BELT04] and extend them by multiplicities as in [TR05]. This setting is well suited to formally define a restricted form of meta-models with arbitrary multiplicities, but without attributes, special forms of associations, and any further kinds of constraints. Compared to the work in [EKT09], type graphs are allowed to have arbitrary multiplicities (a major extension) and loop edges (a minor extension).

Definition 1 (type graph with inheritance) A *type graph with inheritance* is a triple $TGI = (TG, I, Abs)$ consisting of a graph $TG = (TG_V, TG_E, src_{TG}, tgt_{TG})$ (with a set TG_V of vertices, 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 a type graph with inheritance by a pair of functions, from nodes to vertex 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. Between clan-typed graphs we use type-refining morphisms (see also Def. 5 in [TR05]) where a vertex with type t can be mapped to a vertex with a type in the clan of t.

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

- $type_V \circ src_G(e) \in clan_I(src_{TG} \circ type_E(e))$ and
- $type_V \circ tgt_G(e) \in clan_I(tgt_{TG} \circ type_E(e))$.

$(G, type)$ is called a *typed graph* or even *clan-typed* graph. Given two typed graphs $(G, type_G)$ and $(H, type_H)$, a graph morphism $f : G \to H$ is a type-refining morphism if for all $v \in G_V$: $type_{HV}(f_N(v)) \in clan(type_{GV}(v))$ and $type_{HE}(f_E(n)) = type_{GE}$. f is called type-preserving if $type_{HV}(f_N(v)) = type_{GV}(v)$ holds. In the following, we call a type-refining morphism just morphism.

Definition 3 (multiplicities) A *multiplicity* is a pair $[i, j] \in \mathbb{N} \times (\mathbb{N} \cup \{*\})$ with $i \leq j \wedge j > 0$ 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 $|X| \leq j$. The minimum i is also denoted by $min([i, j])$, while the maximum j is also denoted by $max([i, j])$.

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

Example 1 *In Figure 1, a type graph for simple activity models is shown. The minimal activity model consists of a start and an end activity with a transition in between. That's why it has at least 3 model elements. I.e., model elements are activities and transitions. There are three kinds of transitions: transitions running between simple transitions, transitions coming out from decision activities, and transitions running into merge activities. This means that decisions split the control flow, i.e. have one incoming and two outgoing transitions while merge activities merge two control flows.*

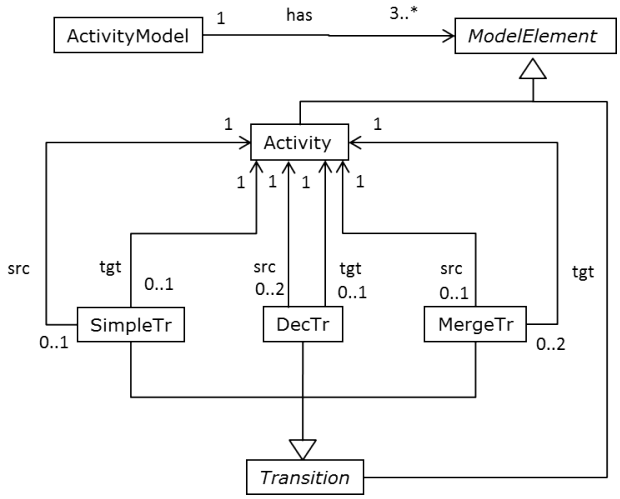


Figure 1: A type graph for simple well-structured activity models

Definition 5 (TGI_{mult}-induced graph language) Given a type graph TGI_{mult} with multiplicities, the induced *graph language* is defined by:
$L(TGI_{mult}) = \{(G = (G_V, G_E, src_G, tgt_G), type_G : G \to TGI) \mid$
$\forall e \in TGI_E \wedge \forall v \in type_G^{-1}(t)$ with $t \in clan_I(src(e)) : |type_G^{-1}(e) \cap src_G^{-1}(v)| \in m_{tgt}(e)$ and
$\forall e \in TGI_E \wedge \forall v \in type_G^{-1}(t)$ with $t \in clan(tgt(e)) : |type_G^{-1}(e) \cap tgt_G^{-1}(v)| \in m_{src}(e)\}$, where $type_G$ is a clan morphism.

To check if all vertex and edge types of a type graph with inheritance and multiplicities can be instantiated, we adapt reasoning methods for UML class diagrams adapting a reasoning method originally formulated for entity-relationship-diagrams [CCM04, BCCG05]. This reasoning method deduces a linear in-equation system.

Definition 6 (Finitely satisfiable type graph with inheritance and multiplicities) A type graph TGI_{mult} with multiplicities is called *finitely satisfiable* if the following condition is satisfied:
Let I be an in-equation system with $TGI'_V \cup TGI'_E$ as variable set containing an integer variable for each element in $TGI_V \cup TGI_E$ correspondingly named. $\forall e \in TGI_E$ with $m_{src}(e) = [k,l]$ and $m_{tgt}(e) = [m,n]$:

- $m \times src(e)' \leq e' \leq n \times src(e)'$ is in I
- $k \times tgt(e)' \leq e' \leq l \times tgt(e)'$ is in I

I has to be solvable such that all variables in $TGI'_V \cup TGI'_E$ are positive.

Example 2 *(Finite satisfiability)*
The type graph shown in Figure 1 is finitely satisfiable since the in-equation system on the right is solvable by positive integers. Actually, a possible solution is $AM' = 1, ME' = has' = 3, A' = 6, SimpleTr' = src'_{Simple} = tgt'_{Simple} = 2, DecTr' = src'_{Dec} = tgt'_{Dec} = 2, MergeTr' = src'_{Merge} = tgt'_{Merge} = 2$ *(Note that ActivityModel is abbreviated by AM, ModelElement by ME, and Activity by A.)*

$$
\begin{aligned}
&ME' = has' \geq 3AM' \\
&SimpleTr' = src'_{Simple} \leq A' \\
&SimpleTr' = tgt'_{Simple} \leq A' \\
&DecTr' = src'_{Dec} \leq 2A' \\
&DecTr' = tgt'_{Dec} \leq A' \\
&MergeTr' = src'_{Merge} \leq A' \\
&MergeTr' = tgt'_{Merge} \leq 2A'
\end{aligned}
$$

3 Graph transformation based on type graphs with inheritance

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 [BELT04]).

The main ingredients of graph grammars are graph rules which will be defined in Def. 8. For controlling a rule application, simple negative application conditions and atomic application conditions are defined which are needed in Section 4.

Definition 7 (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 $q_i \circ x_i = p$.

Remarks. Note that $NAC(x)$ is equal to $P(x, \wedge_{i \in I} x_i))$ with $I = \emptyset$, since there does not exist any $i \in I$ to reach $q_i \circ x_i = p$. Hence, there does not exist a p with $p \circ x = m$ which is exactly the satisfaction condition for $NAC(x)$. Nevertheless, we introduce both kinds of application conditions, since they allow a clearer definition of instance generating rules. In examples, morphisms x and x_i are often not shown totally, but can always be made total. A more general kind of application conditions, called nested application condition, is presented in [HP09] but not needed here.

Definition 8 (rule) 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, $type_R^{-1}(Abs) \subseteq r(K_V)$, and A_p is a set of application conditions of the form $NAC(x)$ or $P(x, \wedge_{i\in I} x_i)$.

Definition 9 (rule matching and application) Given a rule p and a clan-typed graph $(G, type_G)$, then m is a match of p in G if

- m is an injective *match* of the rule $p = (L \xleftarrow{l} K \xrightarrow{r} R, A_p)$ in the graph G;
- $t_K(x_1) = t_K(x_2)$ for $t_K = type_G \circ m \circ l$ 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 $(G, type_G) \overset{p,m}{\Longrightarrow} (H, type_H)$ exists if there is a span of graph morphisms $G \longleftarrow D \longrightarrow H$ and a co-match $m^* : R \to H$ of p in H that give rise to a derivation in the double-pushout (DPO) approach of graph transformation as defined in [CMR96, EEPT06] where pushouts are used to model the gluing of graphs. (See the DPO on the right.)

$$\begin{array}{ccccc} L & \xleftarrow{l} & K & \xrightarrow{r} & R \\ {\scriptstyle m}\downarrow & (PO_1) & \downarrow{\scriptstyle k} & (PO_2) & \downarrow{\scriptstyle m'} \\ G & \xleftarrow{g} & D & \xrightarrow{h} & H \end{array}$$

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

Example rules are shown in the next section where the instance generating graph grammar for the meta model in Figure 1 is presented.

4 Instance generating graph grammars

Having formalized a restricted form of meta models by type graphs with inheritance and multiplicities, we are now ready to define a language by 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 given in Figures 2 to 4. They are applied in three layers:

1. Starting from the empty graph, an arbitrary number of vertices is created. This layer does not terminate, i.e., all its rules remain applicable after their (recurring) applications. Thus, rule application has to be stopped by user interaction or specification of application time or number. In any case, the result is a discrete graph.

2. Next, edges are inserted as long as the specified lower bounds are not all reached. There are three kinds of rules in this layer: If there are two vertices which shall be connected (since lower bounds are not yet reached) and can be connected (since upper bounds are not yet reached), they are just connected by a new edge. If a vertex still has to be connected, but there is no suitable "free" vertex available, a new vertex is created with the new edge. Since all given rules are intended to be matched injectively, there are separate rules to insert loop edges. This process of rule application terminates, if the specified multiplicities guarantee finite satisfiability. (Compare Lemma 1). Otherwise, it may happen that new vertices are created again and again to satisfy the lower bounds without terminating.

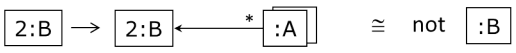
3. Finally, additional edges may be inserted as long as upper bounds are not reached. This applies to loop edges as well.

As the main result of this paper, we present the equivalence 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 10 (instance generating graph grammar and language) Given a type graph TGI_{mult} with multiplicities, an *instance generating graph grammar* is denoted by $IGGG = (TGI, \emptyset, R)$, where $R = R_1 \cup R_2 \cup R_3$ is the union of the following sets of rules. The rules are depicted in Figures 2 - 4 and are formalized in the obvious way according to Def. 8.

- $R_1 = \{\text{createVertex(A')} \mid \forall A' \in TGI_V \wedge A' \notin Abs\}$ with rule createVertex as in Fig. 2
- $R_2 = R_{21} \cup R_{22}$ with
  $R_{21} = \{\text{insertEdge(A,a,B)} \mid \forall A, B \in TGI_V, a \in TGI_E : (m_{src}(a) = [m,n] \wedge m_{tgt}(a) = [k,l])\} \cup$
  $\{\text{insertLoopEdge(A,a)} \mid \forall A \in TGI_V, a \in TGI_E : (m_{src}(a) = [m,n] \wedge m_{tgt}(a) = [k,l])\}$
  $R_{22} = \{\text{insertEdgeWithVertex(A,a,B,B')} \mid \forall A, B, B' \in TGI_V, a \in TGI_E :$ with
  $m_{src}(a) = [m,n] \wedge m_{tgt}(a) = [k,l] \wedge k > 0 \wedge n \neq "*" \wedge B' \in clan(B) \wedge B' \notin Abs\}$
  with rules insertEdge, insertEdgeWithVertex, and insertLoopEdge as in Fig. 3
- $R_3 = \{\text{insertAdditionalEdge(A,a,B)} \mid \forall A, B \in TGI_V, a \in TGI_E$ with
  $m_{src}(a) = [m,n] \wedge m_{tgt}(a) = [k,l]\}$ with rules insertAdditionalEdge as in Fig. 4

If bounds k or m are equal to 0 or bounds l or n are equal to "*", the corresponding negated application conditions are equal to true in rules insertEdge, insertAdditionalEdge, and insertLoopEdge. Atomic application conditions of the form $P(x : L \to X, x_1 : X \to C_1)$ with X consisting of one vertex being mapped to a vertex with infinite many adjacent edges in C_i is semantically equal to a negative application condition containing a vertex of the same type (see below).



R is layered, i.e. there is a function $rl : R \to \mathbb{N}$ with $rl(r) = i$ for all $r \in R_i$ for $i = \{1,2,3\}$. Function rl is called *rule layer function*.
The *graph language* generated by $IGGG$ consists of typed graphs $(G, type_G)$ such that $L(IGGG) = \{(G, type_G) \mid \emptyset \overset{*}{\Rightarrow}_{R_1} (H, type_H) \overset{*}{\Rightarrow}_{R_2} (K, type_K) \overset{*}{\Rightarrow}_{R_3} (G, type_G))\}$. $(K, type_K)$ is in normal form, i.e., there does not exist a rule $r \in R_2$ applicable to $(K, type_K)$.

Layered graph grammars are presented in [EEL+05]. The order of rule applications described above can also be considered as a special graph program [HP01]: R_1; as long as possible R_2; R_3
Note that:

- It is obvious that all graphs in $L(IGGG)$ are clan-typed over TGI_{mult}.
- R_1 contains rules createVertex(A') for all concrete types in TGI_V.

- If the type graph contains edge loops, A may become equal to B. E.g. given an edge loop a on A, there are also rules insertEdge(A,a,A), insertEdgeWithNode(A,a,A,A'), and insertAdditionalEdge(A,a,A).
- It is straight forward to see that instance generating graph grammars as defined in [EKT09] for special multiplicities are covered by instance generating graph grammar as defined here for arbitrary multiplicities.

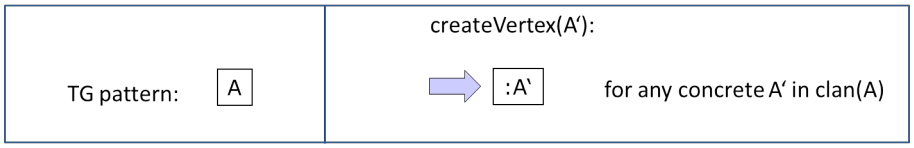


Figure 2: Instance generation: creation of nodes in layer 1

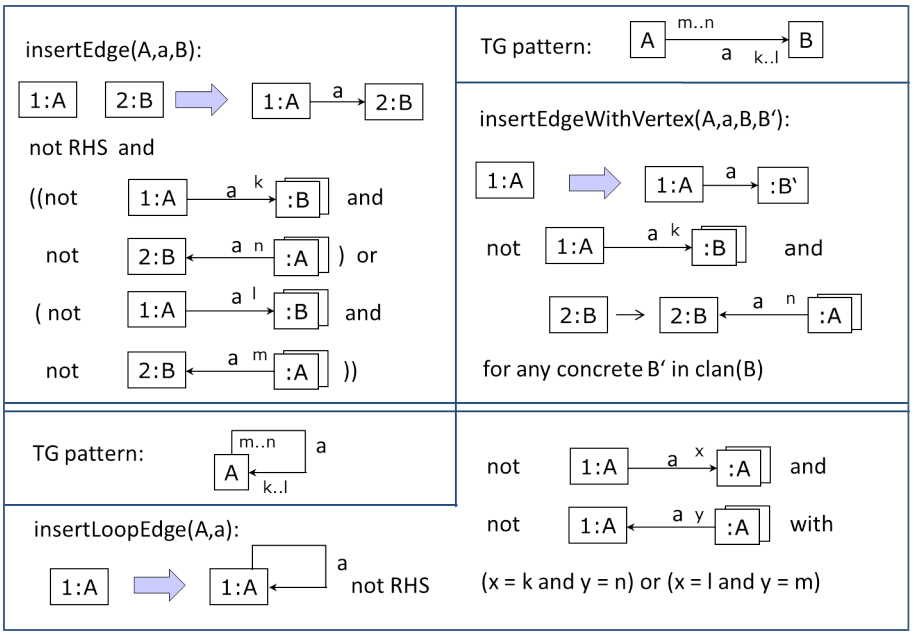


Figure 3: Instance generation: creation of required edges in layer 2

Example 3 (instance generation) *In Figures 5 to 7, example rules for the instance generation of simple activity models are shown. They are built strictly according to the general algorithm. Create vertex rules are generated for all concrete types as indicated in Figure 5. Layer 2 and 3 rules have to be generated for 7 edges. In Figures 6 and 7 we consider the insertion rules needed for the src-edge between* DecTr *and* Activity. *Note that the layer 2 rules in Figure 6 does not contain insertion rules in case that new* DecTr *instances have to be created.*

Example 4 (A non-terminating IGGG) *In this example, we want to discuss what is happening if we take a type graph that is not finitely satisfiable. Considering the sample type graph in the*

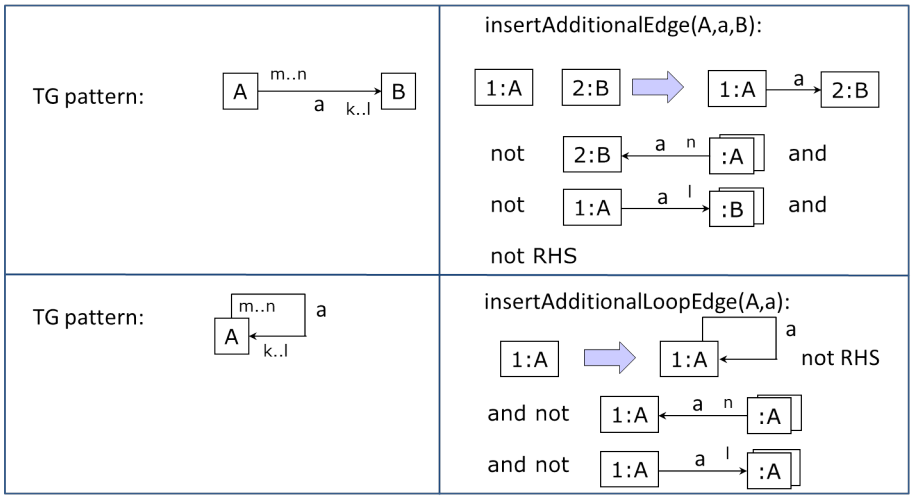


Figure 4: Instance generation: creation of optional edges in layer 3

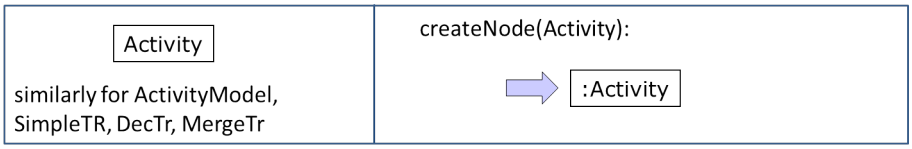


Figure 5: Example rules of layer 1

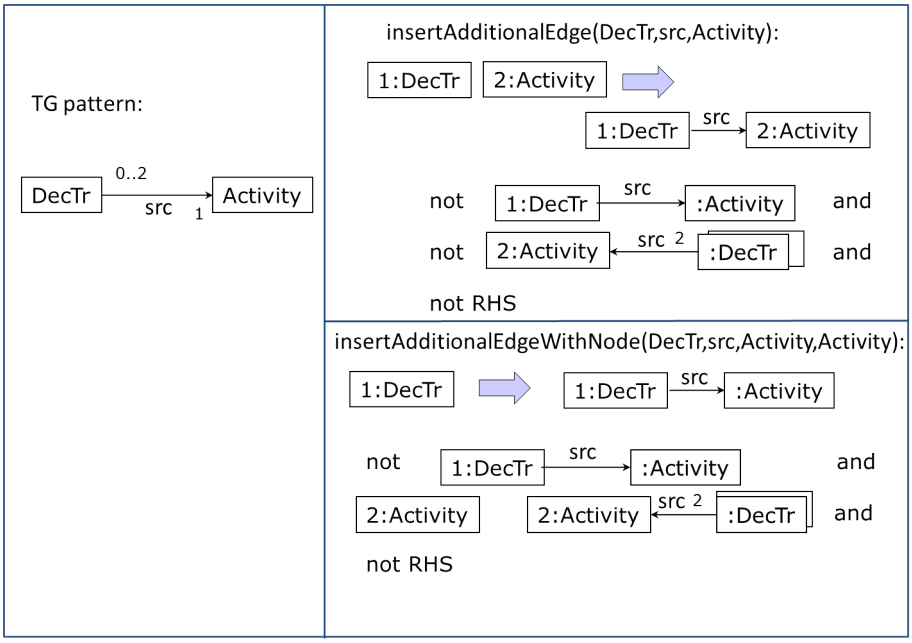


Figure 6: Example rules of layer 2

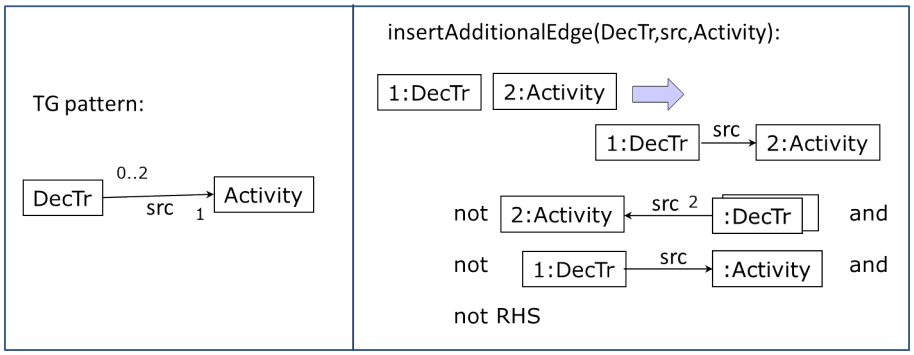


Figure 7: Example rules of layer 3

upper row of Figure 8. Its in-equation system consists the following equation: $A' = a = 2A'$ *not being solvable by finite numbers. Figure 8 also shows rules of Layers 1 and 2. Those of Layer 3 are omitted since Layer 2 is not terminating and therefore, rules of Layer 3 will never be applied. In the lower row of Figure 8, a sample derivation is depicted also using the rule insertLoopEdge(A,a) not being depicted in figure 8. Since only one edge may go out but two have to come in to each vertex, we have to continue creating new vertices which leads to non-termination of Layer 2.*

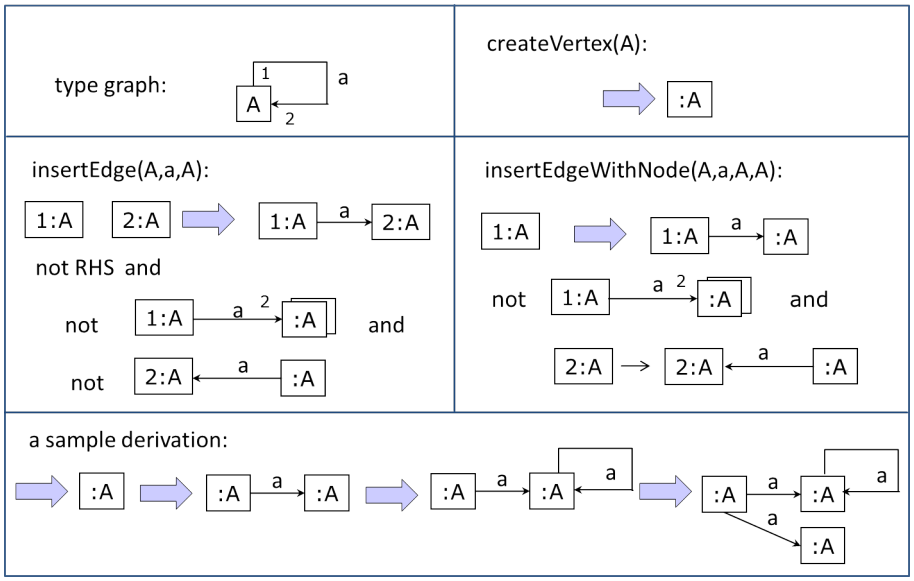


Figure 8: Example of a non-terminating IGGG

In the previous example, we have seen that Layer 2 can become non-terminating if the input type graph is not finitely satisfiable. However, in the following lemma we can show that Layer 2 is terminating if the input type graph is finitely satisfiable.

Lemma 1 (termination of rule layer 2) *Given a type graph with inheritance and multiplic-*

ities TGI_{mult} that is finitely satisfiable and an instance generating graph grammar $IGGG = (TGI, \emptyset, R)$, let $L_1(IGGG) = \{(H, type_H) \mid \emptyset \overset{*}{\Rightarrow}_{R_1} (H, type_H)\}$ constructed as in Def. 10. All derivation sequences $(H, type_H) \overset{*}{\Rightarrow}_{R_2} (G, type_G)$ with $(H, type_H) \in L_1(IGGG)$ terminate.

Proof. We show that if there is a derivation $(H, type_H) \overset{*}{\Rightarrow}_{R_2} (G, type_G)$ with $(H, type_H) \in L_1(IGGG)$ not terminating, then TGI_{mult} is not finitely satisfiable. Such a derivation is non-terminating, if after any derivation step there is still a match for some rule of layer 2. Considering the rules in layer 2, it is obvious that rules insertEdgeWithVertex() are the only ones that insert new matches. The others just consume matches. W.l.o.g. there is rule insertEdgeWithVertex(A,a,B,B') that is applied infinitely often. This means that the number of instances of type B' grows over all limits. Thus, TGI_{mult} is not finitely satisfiable. □

As main result, the following theorem states that the instance sets generated by an *IGGG* and those induced by a type graph with multiplicities (being finitely satisfiable) are equal.

Theorem 1 (equality of languages) *Given a type graph TGI_{mult} with inheritance and multiplicities and an instance generating graph grammar $IGGG = (TGI, \emptyset, R)$ for TGI_{mult}, we have $L(IGGG) = L(TGI_{mult})$.*

Proof. Let $R_i = \{r \mid rl(r) = i\}$ for $i = \{1, 2, 3\}$.

1. $L(IGGG) \subseteq L(TGI_{mult})$: Given any derivation $\emptyset \overset{*}{\Rightarrow}_{R_1} (H, type_H) \overset{*}{\Rightarrow}_{R_2} (K, type_K) \overset{*}{\Rightarrow}_{R_3} (G, type_G)$, we have to show that $(G, type_G) \in L(TGI_{mult})$. Let $\emptyset \overset{*}{\Rightarrow}_{R_1} (H, type_H)$ consist of n steps, i.e. $(H, type_H)$ has n nodes.

   What can be said after $(H, type_H) \overset{*}{\Rightarrow}_{R_2} (K, type_K)$?
   Due to Lemma 1, we know that this derivation sequence does always terminate. Let graph $(K, type_K)$ be the result graph to which none of the rules of layer 2 is applicable. We have to show that $(K, type_K) \in L(TGI_{mult})$:

   - No insertEdgeWithVertex(A,a,B,B') is applicable: Either the lower bounds are reached for all edges and thus, $(K, type_K) \in L(TGI_{mult})$ or there is a vertex of type B' for which the upper bound has not been reached. Then, there is still a pair of A and B-typed vertices not connected by an a-typed edge and thus, there is a match of insertEdge(A,a,B) which contradicts our assumption that no rule of layer 2 is applicable.
   - No insertEdge(A,a,B) is applicable: Either the lower bounds are reached for all edges and thus, $(K, type_K) \in L(TGI_{mult})$ or there is a vertex of type A or type B for which the upper bound has been reached. This situation would lead to a match of rule insertEdgeWithVertex(A,a,B,B') or rule insertEdgeWithVertex(B,a,A,A') for a concrete type $B' \in clan(B)$ or $A' \in clan(A)$ which contradicts our assumption that no rule of layer 2 is applicable.
   - No insertLoopEdge(A,a) is applicable: Either the lower bounds are reached for all edges and thus, $(K, type_K) \in L(TGI_{mult})$ or there is a vertex of type A for which one of the upper bounds has been reached. This situation would lead to a match of rule

insertEdgeWithVertex(A,a,A,A') for a concrete type $A' \in clan(A)$ which contradicts our assumption that no rule of layer 2 is applicable.

Applications of rules in R_3 always lead to graphs in $L(TGI_{mult})$, since they just add edges as long as upper bounds have not reached.

2. $L(TGI_{mult}) \subseteq L(IGGG)$: Given a graph $(G, type_G) \in L(TGI_{mult})$, we have to construct a derivation sequence $\emptyset \stackrel{*}{\Rightarrow} (G, type_G)$ over $IGGG$. Since TGI_{mult} is finitely satisfiable, for all (A,a,B) there are $i(A)$ vertices of type A, $i(a)$ edges of type a, and $i(B)$ vertices of type B in G_V such that all multiplicities are satisfied.

   Since R_1 contains a creation rule for every vertex type, the vertices in G_V are created in $|G_V|$ steps by applying rules of R_1 to graph $\emptyset$, i.e. we construct a derivation $\emptyset \stackrel{|G_V|}{\Rightarrow}_{R_1} (H, type_H)$ with $H = (H_V, \emptyset, \emptyset, \emptyset)$, $H_V = G_V$, and $type_H = (type_{G_V}, \emptyset)$.

   Since all nodes needed have already been created in layer 1, rules of set R_{21} are used only to create all required edges in layer 2. Let a be an edge in TGI_E with $src(a) = A$ and $tgt(a) = B$:

   - $A \neq B$: Since $m \times i(A) \leq i(a) \leq n \times i(A)$ and $k \times i(B) \leq i(a) \leq l \times i(B)$, rule insertEdge(A,a,B) is applied as long as $i(a) \leq m \times i(A)$ or $i(a) \leq k \times i(B)$. When they are not applicable anymore, both lower bounds, k and m, have been reached.
   - $A = B$: Since $m \times i(A) \leq i(a) \leq n \times i(A)$ and $k \times i(A) \leq i(a) \leq l \times i(A)$, rules insertEdge(A,a,A) and insertLoopEdge(A,a) are applied as long as $i(a) \leq m \times i(A)$ or $i(a) \leq k \times i(A)$. When it is not applicable anymore, both lower bounds, k and m, have been reached.

   Thus, if no rule of R_{21} is applicable anymore, we have reached graph $(K, type_K) \in L(TGI_{mult})$ with $K_V = G_V$ and $K_E \subseteq G_E$.

   Additional edges are created by the application of rules in R_3. Since $i(a) \leq l \times i(B)$ and $i(a) \leq n \times i(A)$, the upper bounds, l and n, are not reached and the rules in R_3 can be applied as long as all edges in G_E have been created. Thus, the resulting graph $(G, type_G)$ is in $L(IGGG)$.

□

5 Related Work

There is work on the relation of different kinds of grammars to meta-models. In all approaches, meta-models come without additional well-formedness rules:

One closely related approach is the one by Alanen and Porres [AP03]: They describe two algorithms, one to derive a context-free string grammar from a meta model and another one for deriving a meta model from a context-free string grammar. The aim of their work is to bridge the gap between artifacts defined by a context-free string grammar and software models for which the syntax is specified by a meta model. Their algorithm for grammar derivation can only deal

with composite associations between meta classes, restricting it to tree-like meta models which is a severe limitation for practical usage. Furthermore, the algorithm does not support ordinary associations with arbitrary multiplicities. This limitation is not surprising given the properties of context-free string grammars. It represents one reason for the approach to use graph grammars instead of context-free grammars.

The approach by Hoffmann and Minas [HM10, HM11] translates a restricted form of class diagrams into contextual star grammars being a kind of hyperedge replacement grammars extended by contextual rules including path expressions and variable parts. Hence the instance generation process uses non-terminals during instance construction, similarly to context-free string grammars, and constructs instances along spanning trees. In contrast, our instance generation is purely graph-oriented. It restricts rule application by using rules with application conditions and layering their application. While Hoffmann and Minas focus on parsing of meta model instances, the approach in this paper shall be used to generate test cases for model transformations and to provide users of graphical editors with advanced editing operations deduced from our grammar rules. Especially for the second application case, rules without non-terminals are better suited.

Instance generation for meta-models has also been considered in further approaches not using grammars: A related problem is the automated snapshot generation for class diagrams for validation and testing purposes, tackled by Gogolla et al. [GBR05]. In their approach, properties that the snapshot has to fulfill are specified in OCL. Formal methods such as Alloy [Jac02, ABGR07] 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. While these approaches are pretty promising for automatic test case generation, they seem to fit less for the automatic deduction of advanced editing operations, since incremental structure modifications are needed in this case. Instead, grammar rules seem to be well suited to deduce such incremental modifications.

6 Conclusion

In this paper, we have presented how a restricted form of meta-models can be formalized by type graphs with inheritance and arbitrary multiplicities. This kind of type graphs can be translated to instance generating graph grammars. We have shown that the languages defined by each of these concepts are equal if the type graph has multiplicities that are finitely satisfiable, i.e., that can be satisfied by a finite instance. This work is a direct extension of the translation shown in [EKT09]. The main new contribution of this paper is the extension from restricted multiplicities such as 0..1, 1, 1..*, and 0..* to arbitrary multiplicities. Thus, another step is taken to close the conceptual gap in modeling language definition by allowing the adoption of grammar-based definition techniques also to languages defined by meta models.

Further extensions have to be made in future to cover all essential concepts of meta-modeling by the translation to instance generating graph grammars. A major step is to include well-formedness rules formulated in OCL into this translation. In [EKT09] we sketch already how a restricted form of OCL constraints can be translated into graph constraints and how they can be taken into account during instance generation. In future, the existing restrictions on meta-models

have to be erased step-by-step to close the conceptual gap between meta-models and grammars finally.

Future work is also needed to elaborate on possible applications of our technique: The instance generation shall be applied to further and especially larger type graphs than shown in the running example, proving that this approach can scale. For testing model transformations, techniques are needed that allow the generation of selected instances that are representative test cases. For editor generation, it needs to be explored how the graph grammar generated from a meta model can provide a basis to automatically deduce advanced visual editing rules.

Acknowledgements

Many thanks to Annegret Habel and Thorsten Arendt who gave valuable comments on this paper.

References

[ABGR07] K. Anastasakis, B. Bordbar, G. Georg, I. Ray. UML2Alloy: A Challenging Model Transformation. In Engels et al. (eds.), *Model Driven Engineering Languages and Systems, 10th International Conference, MoDELS 2007, Nashville, USA, September 30 - October 5, 2007, Proceedings*. LNCS 4735, pp. 436–450. Springer, 2007.

[AP03] M. Alanen, I. Porres. A relation between context-free grammars and meta object facility metamodels. Technical report 606, TUCS Turku Center for Computer Science, March 2003.

[BCCG05] D. Berardi, A. Cali, D. Calvanese, G. D. Giacomo. Reasoning on UML Class Diagrams. *Artifical Intelligence* 168:70–118, 2005.

[BELT04] R. Bardohl, H. Ehrig, J. de Lara, G. Taentzer. Integrating Meta-modelling Aspects with Graph Transformation for Efficient Visual Language Definition and Model Manipulation. In Wermelinger and Margaria (eds.), *Fundamental Approaches to Software Engineering (FASE 2004), Proceedings*. LNCS 2984, pp. 214–228. Springer, 2004.

[CCM04] M. Cadoli, D. Calvanese, T. Mancini. Finite satisfiability of UML class diagrams by Constraint Programming. In *Proc. of the 2004 International Workshop on Description Logics (DL 2004)*. Volume 104. CEUR-WS.org, 2004.

[CMR96] A. Corradini, U. Montanari, F. Rossi. Graph Processes. *Fundam. Inform.* 26(3/4):241–265, 1996.

[EEL+05] H. Ehrig, K. Ehrig, J. de Lara, G. Taentzer, D. Varró, S. Varró-Gyapay. Termination Criteria for Model Transformation. In Cerioli (ed.), *Fundamental Approaches to Software Engineering, 8th International Conference (FASE 2005), Proceedings*. LNCS 3442, pp. 49–63. Springer, 2005.

[EEPT06] H. Ehrig, K. Ehrig, U. Prange, G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. EATCS Monographs in Theor. Comp. Science. 2006.

[EKT09] K. Ehrig, J. M. Küster, G. Taentzer. Generating instance models from meta models. *Software and System Modeling* 8(4):479–500, 2009.

[GBR05] M. Gogolla, J. Bohling, M. Richters. Validating UML and OCL Models in USE by Automatic Snapshot Generation. *Software and Systems Modeling* 4(4):386–398, 2005.

[HM10] B. Hoffmann, M. Minas. Defining Models - Meta Models versus Graph Grammars. *ECEASST* 29, 2010.

[HM11] B. Hoffmann, M. Minas. Generating Instance Graphs from Class Diagrams with Adaptive Star Grammars. *ECEASST* 39, 2011.

[HP01] A. Habel, D. Plump. Computational Completeness of Programming Languages Based on Graph Transformation. In *Proc. Foundations of Software Science and Computation Structures (FOSSACS 2001)*. Volume 2030, pp. 230–245. 2001.

[HP09] A. Habel, K.-H. Pennemann. Correctness of high-level transformation systems relative to nested conditions. *Mathematical Structures in Computer Science* 19(2):245–296, 2009.

[Jac02] D. Jackson. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.* 11(2):256–290, 2002.

[Küs06] J. M. Küster. Definition and validation of model transformations. *Software and System Modeling* 5(3):233–259, 2006.

[MBT06] J.-M. Mottu, B. Baudry, Y. L. Traon. Mutation Analysis Testing for Model Transformations. In Rensink and Warmer (eds.), *Model Driven Architecture - Foundations and Applications, Second European Conference, ECMDA-FA 2006, Bilbao, Spain, July 10-13, 2006, Proceedings*. LNCS 4066, pp. 376–390. Springer, 2006.

[Roz97] G. Rozenberg (ed.). *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*. World Scientific, 1997.

[TR05] G. Taentzer, A. Rensink. Ensuring Structural Constraints in Graph-Based Models with Type Inheritance. In *Fundamental Approaches to Software Engineering (FASE 2005), Proceedings*. LNCS 3442, pp. 64–79. Springer, 2005.

[WTEK08] J. Winkelmann, G. Taentzer, K. Ehrig, J. M. Küster. Translation of Restricted OCL Constraints into Graph Constraints for Generating Meta Model Instances by Graph Grammars. *Electr. Notes Theor. Comput. Sci.* 211:159–170, 2008.