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

Generating Instance Models from Meta Models

Karsten Ehrig[1], **Jochen M. Küster**[2], **Gabriele Taentzer**[3]

[1] Department of Computer Science, University of Leicester, United Kingdom, e-mail: : karsten@mcs.le.ac.uk
[2] IBM Zurich Research Laboratory, CH-8803 Rüschlikon, Switzerland, e-mail: : jku@zurich.ibm.com
[3] Department of Computer Science, Philipps-University Marburg, Germany, e-mail: : taentzer@mathematik.uni-marburg.de

Received: date / Revised version: date

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.

Key words meta model, UML, graph grammar, instance generation

1 Introduction

With models expressed in the Unified Modeling Language (UML) [35] 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, due to the declarative form of a meta model.

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 a number of emerging applications where firstly an approach is needed for generating a large set of instances 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 description of the language defined by a meta model include automated testing of model transformations and code generators as well as testing of model debuggers [27]. Model driven architecture [34] favors a widespread usage of model transformations. The quality of model transformations is thus crucial for their successful and widespread usage and needs to be validated by automated testing [29, 32]. In the related problem of compiler testing [12], 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. For testing model transformations and code generators, a large set of automatically generated instance models is required but currently it is unclear how this large set can be obtained. A large set of automatically generated instance models is also beneficial as input for model debuggers for ensuring the quality of the model debugger itself, i.e., for testing that the model debugger can handle all valid models. Further, automatic editor generation for domain specific languages may also depend on an operational description of the language.

Graph grammars [14] provide a constructive, well-studied approach to language definition with a formal foundation that allows to prove important properties. In model-driven engineering, graph transformation has been used to provide a formal foundation for models which then enables to prove termination and confluence of model transformations [28, 39], to formalize model refactoring operations [31] or for formally specifying model interpreters [26]. One key characteristic of

graph grammars is that they provide an operational description of a language.

Up until now, the relationship between meta models and graph grammars has not been studied in depth, but started in [10]. In this paper, we propose to derive an *instance-generating* graph grammar from a meta model in order to obtain an operational description of the language defined by the meta model. 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 languages defined by grammars also to languages defined by a meta model and thereby closes an important technology gap.

In order to achieve this, the *instance-generating* graph grammar derived from a meta model has to generate all possible instances of the meta model and should not generate any model that is not an instance of the meta model. In terms of graph grammar derivation, 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 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 the applications of the instance-generating graph grammar: 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. The paper is based on our previous work [20] but elaborates a new section for tool support. Furthermore, it extensively discusses further extensions of this approach.

The paper is organized as follows: 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 this generation process, but have to be checked afterwards until now. 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 show formally the completeness of the generated instances. The construction of an instance generating graph grammar from the meta model is automated, the tool support is described in Section 6. In Section 7 we consider extensions of our approach. We conclude by a discussion of related and future work.

2 Metamodels with OCL-Constraints

Visual languages such as the UML [35] 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

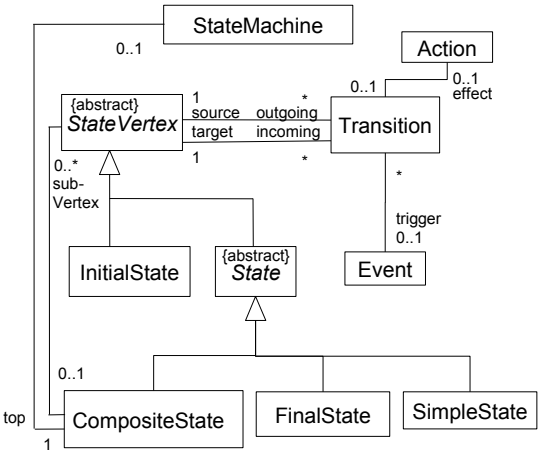


Figure 1 Meta model for statecharts

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 further be restricted by the use of additional constraints specified in the Object Constraint Language (OCL) [36].

Figure 1 shows a slightly simplified statechart meta model (based on [35]) 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 (this would lead to additional rules in the instance-generating graph grammar, see below). 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 random 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)
5. Well-formedness rule for acyclic subvertex relations:
   context CompositeState inv:
   not self.allSubVertices()->includes(self)
   with additional operation:
   CompositeState::allSubVertices():Set(StateVertex)
   allSubVertices = subvertex->
   union(subvertex->collect(v | v.allSubVertices()))

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 (generalization/specialization), instantiation is rather straightforward by creating instances of metaclasses and associations. However, meta models as commonly used in language specification documents such as [35] heavily make 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 *typed graph transformations with inheritance* (see [10]), which will be the basis for the formal background for *Instance Generating Graph Grammars (IGGG)* in Section 5.

In object-oriented modelling, graphs can be used at two levels: the type level and the instance level. This typing concept has been described by *typed graphs* [14], where a fixed *type graph* serves as abstract representation of the meta model. 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 meta model can thus be represented by a type graph with inheritance plus a set of constraints over this type graph expressing multiplicities.

A graph has nodes, and edges, where each edge links two nodes. We consider directed graphs, i.e. every edge has a distinguished start node (its source) and end node (its target). A type graph defines a set of types, which is used to assign a type to the nodes and edges of a graph. A type graph with inheritance is a type graph with a distinguished set of abstract nodes and inheritance relations between the nodes. The inheritance clan of a node represents all its sub nodes.

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*.*

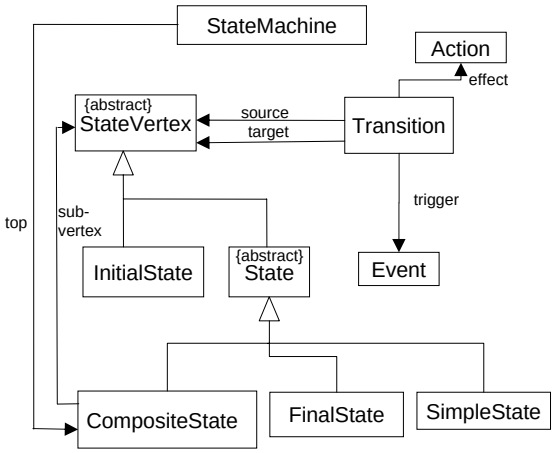


Figure 2 Sample typegraph derived from Figure 1

Figure 2 shows a sample type graph derived from Figure 1. Please note that associations are replaced by labeled directed edges where one role name is taken over as edge label. The arrow direction depends on which role name is taken over, i.e. points to the end of that role name.

Graphs are related by graph morphisms, which map the nodes and edges to those of another graph, compatible with source and target mappings.

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 in general, but takes the inheritance relation into account. Typing is realized as in object-oriented modelling. That means for example that all instance nodes have to be typed by non-abstract type nodes. The resulting morphism is 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.

Figure 3 shows a sample instance graph typed over the graph in Figure 2 where the typing relations are indicated by dashed arrows.

The main idea of graph grammars and graph transformation is the rule-based modification of graphs where each

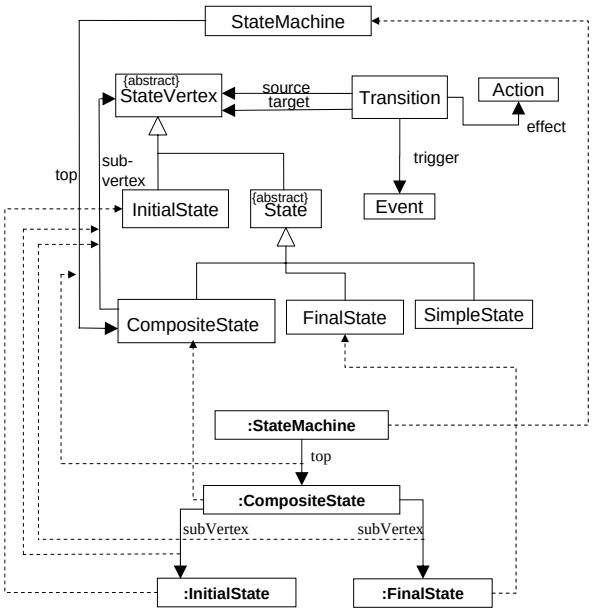
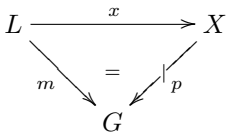


Figure 3 Sample instance graph typed over Figure 2 (typing relations are indicated by dashed arrows).

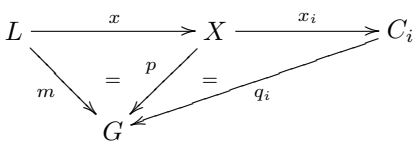
application of a graph transformation rule leads to a graph transformation step. The core of a graph transformation rule is a pair of graphs, called left-hand side and right-hand side. Roughly spoken, applying a rule means to find a match of the left-hand side in the source graph and to replace the image of the left-hand side by a copy of the right-hand side leading to the target graph of the graph transformation.

For controlling a rule application, negative application conditions $NAC(x)$ and atomic application conditions $P(x, \vee_{i \in I} x_i))$ can be defined which are needed in Section 4. Although $NAC(x)$ is a special case of $P(x, \vee_{i \in I} x_i))$ with $I = \emptyset$, we introduce both kinds of application conditions, due to a clearer definition of instance generating rules. Roughly spoken, a negative application condition can be considered as a graph structure that must not be present in the source graph. Formally, a *match* is defined by a matching morphism $m : L \to G$ that embeds the left-hand side L of a rule into the source graph G.

Definition 3 (application condition) *A* 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 $P(x, \vee_{i \in I} x_i)$ 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 *$P(x, \vee_{i \in I} x_i)$ 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 $q_i \circ x_i = p$:*



Remark 1 (partial morphism) For abbreviation proposes the morphism $x : L \to X$ is sometimes noted as a *partial morphism* only. Without loss of generality x could be extended to a total morphism by adding missing nodes and edges of L to X.

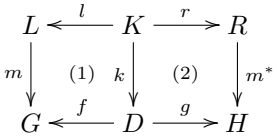
Definition 4 (rules) *A rule typed over a type graph $TGI = (TG, I, Abs)$ with inheritance is given by $p = (L \xleftarrow{l} K \xrightarrow{r} R, A_p)$, where L, K, R are clan-typed graphs, l and r are type-preserving injective graph morphisms, $ctp_R^{-1}(Abs) \subseteq r(K_V)$, and A_p is a set of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

Remark 2 (type-refining morphism) Between clan-typed graphs we use type-refining morphisms (see also Def. 5 in [37]) 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. Intuitively, type-refining morphisms are needed to apply the well developed theory for typed graphs also for typed graphs with inheritance.

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 morphism of the left-hand side L 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 application conditions 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 where (1) and (2) are pushouts in the category of $\mathbf{Graphs_{TG}}$ as defined in [18]:



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 $\nexists r \in R : (H, ctp_H) \Rightarrow_r (H', ctp_{H'})$.

Rules can be distributed over different layers. In each layer, the rules are applied for as long as possible before going to the next layer.

Example 1 (rule with application conditions) Figure 4 shows the application of the rule InsertStateVertex_source_Transition taken from the graph grammar derivation rule set in Figure 10 (which will be described in Section 4).

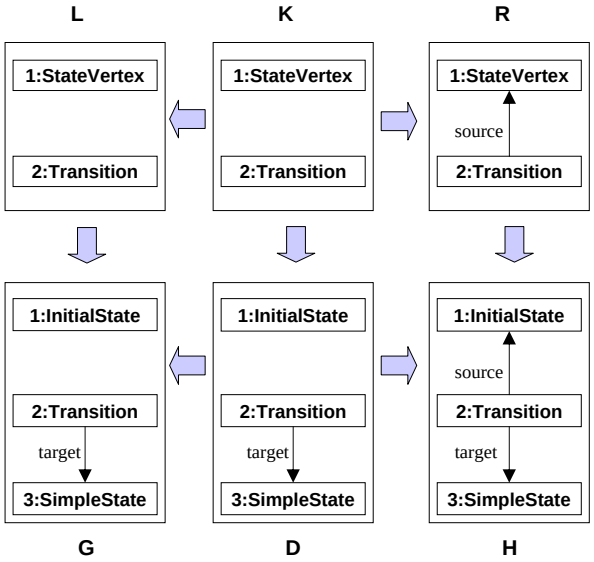


Figure 4 Application of rule InsertStateVertex_source_Transition to connect :InitialState via the source edge with :Transition

The abstract node 1:StateVertex in the left-hand side L is mapped to the concrete node 1:InitialState in the graph G, indicated by the same number in front of the node names. 2:Transition is mapped respectively. All remaining parts of G, i.e. 3:SimpleState and the target edge, are preserved during rule application. Moreover 1:InitialState and 2:Transition are preserved since they are contained in the intermediate graphs K and D. In the right-hand side R the source edge is inserted between both nodes resulting in the target graph H. The advantage of the abstract node 1:StateVertex is that the rule has to be defined only once to be applicable to all concrete nodes typed over StateVertex (see Figure 3 for the typing relations).

Figure 5 shows two application conditions for the sample rule. A rule can be applied only if the part of the rules left-hand side L, identified with the NAC graph, is not contained in the source graph G. The first negative application condition NAC_1 ensures that the rule can only be applied if 1:StateVertex and 2:Transition are not connected via a source edge, so the rule could only be applied once. The second negative application condition NAC_2 ensures that 2:Transition is not connected to another :StateVertex in the current graph. 1:StateVertex has been added to NAC_2 to fulfill the condition that the morphism $x : L \rightarrow NAC_2$ has to be total. Please note, that for abbreviation purposes x may be noted as a partial morphism, i.e. 1:StateVertex may be omitted.

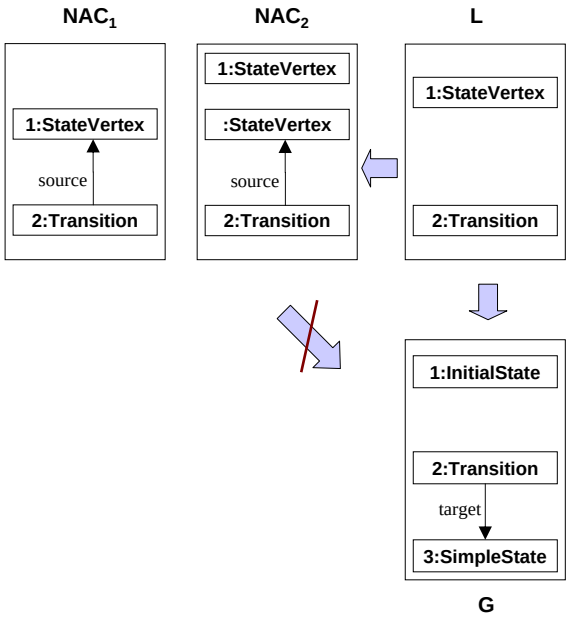


Figure 5 Application conditions NAC_1 and NAC_2 of rule InsertStateVertex_source_Transition

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 [16] 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. Alternatively, we could specify in advance how many instances of each class we allow and terminate automatically once these bounds have been reached. 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 deter-

mined 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 6) contains rules of the form createE' where E' is replaced by the name of any non-abstract class, so $E' \in clan(E)$. 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 Figure 7 and 8). The first rule for each pattern creates a link between existing instances. The second rule for each pattern creates a link together with an instance of an object. In general, we use NACs to 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' $\in clan(\mathrm{E})$ resp. A' $\in clan(\mathrm{A})$ if no application condition holds. In case of a 1 to * relation (see pattern p_1) a new instance of E' $\in clan(\mathrm{E})$ 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 Figure 9). The rules for these 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.

Not shown in the figures is another rule set where all edges are redirected, i.e., their direction is reversed. These complementary rules are needed for all rules in Figure 7 and the second rule in Figure 9 only, since the associations have different multiplicities at their ends.

Generating Statechart Instances: We now discuss an instance-generating graph grammar for the meta model of statecharts (see Figure 1). For brevity, we do not show the details of all rules. The example rules shown in Figure 10 - 12 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 10 - 12 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 StateVertex, 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 pattern p_2, we derive the corresponding two rules. One of them is shown in Figure 10, 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).

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 [10]). 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 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 [37]) is a tuple $TG_{mult} =$*

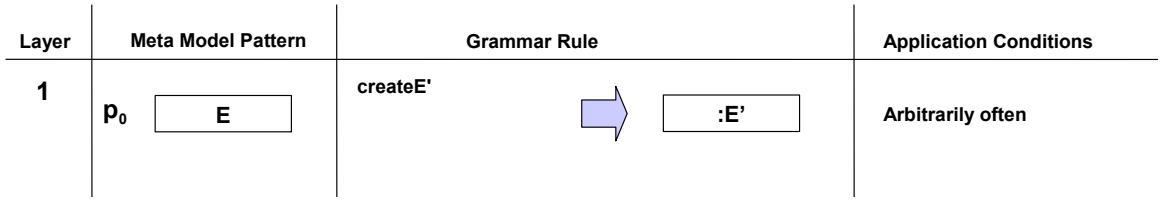


Figure 6 Rules for graph grammar derivation: Layer 1

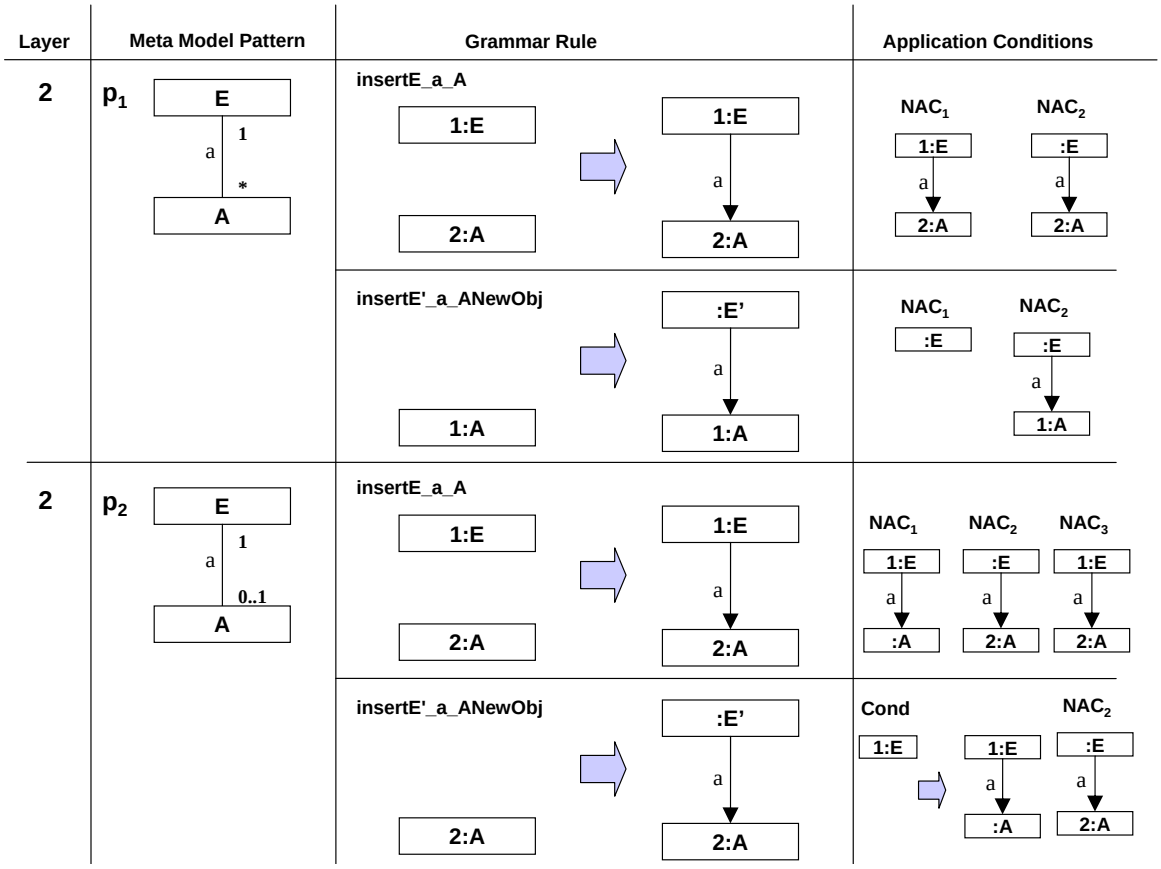


Figure 7 Rules for graph grammar derivation: Layer 2

(TGI, m_{src}, m_{tgt}) *consisting of a type graph with inheritance* TGI *and additional functions* $m_{src}, m_{tgt} : TGI_E \rightarrow Mult$*, called* edge multiplicity functions.

Considering the meta model in Figure 1, it can be formalized to a type graph with multiplicities in a straightforward 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. Figure 13 shows the resulting type graph with multiplicities derived from Figure 1.

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:*
$L(TGI_{mult}) =$
$\{(G = (G_V, G_E, src_G, tgt_G), ctp_G : G \rightarrow TGI) \mid$
$\forall e \in TGI_E \wedge \forall v \in ctp_G^{-1}(t)$ *with*
$t \in clan(src(e)) : \ |ctp_G^{-1}(e) \cap src^{-1}(v)| \in m_{tgt}(e)$ *and*

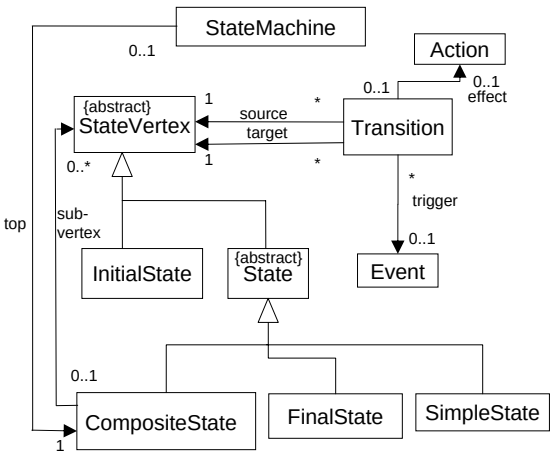


Figure 13 Type graph with multiplicities derived from Figure 1

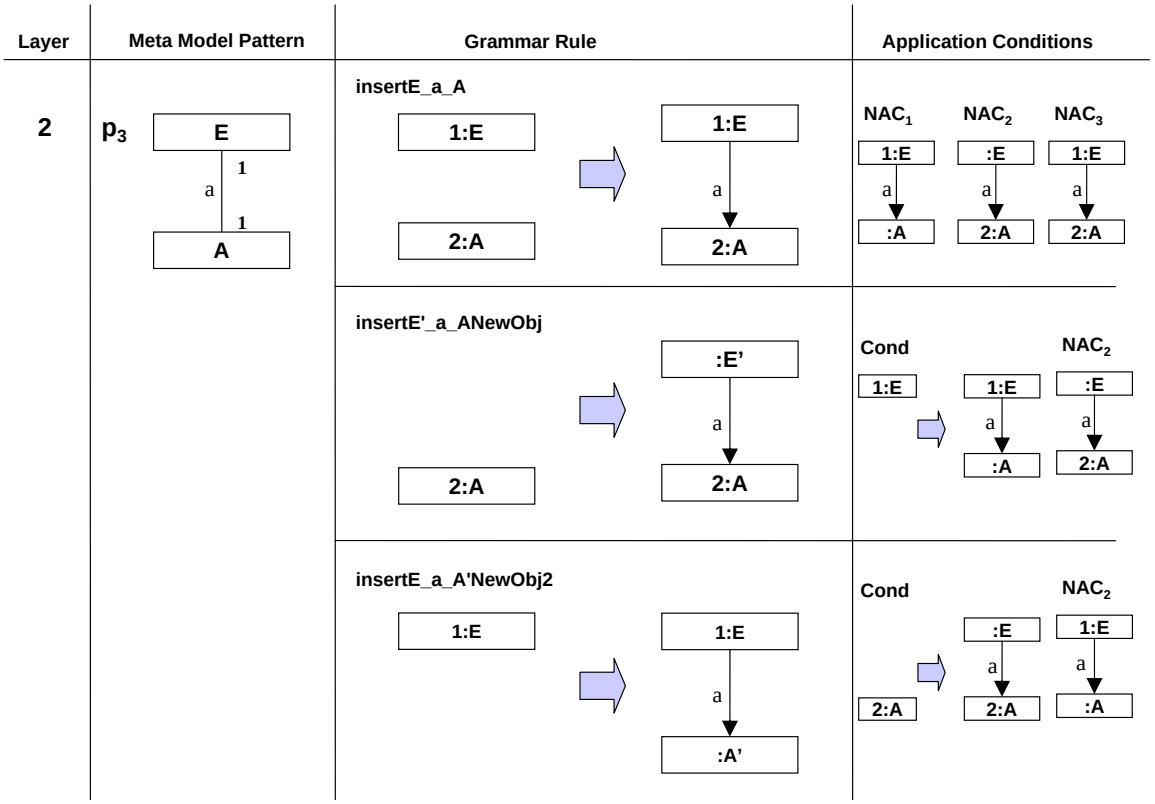


Figure 8 Rules for graph grammar derivation: Layer 2

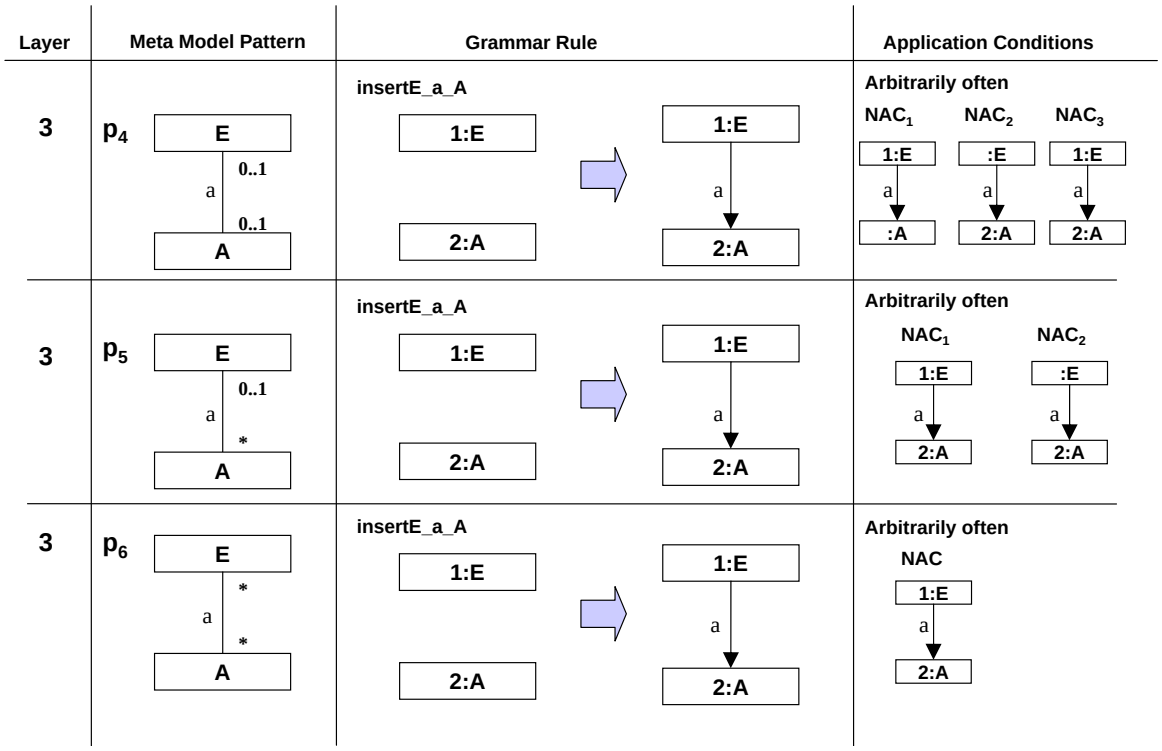


Figure 9 Rules for graph grammar derivation: Layer 3

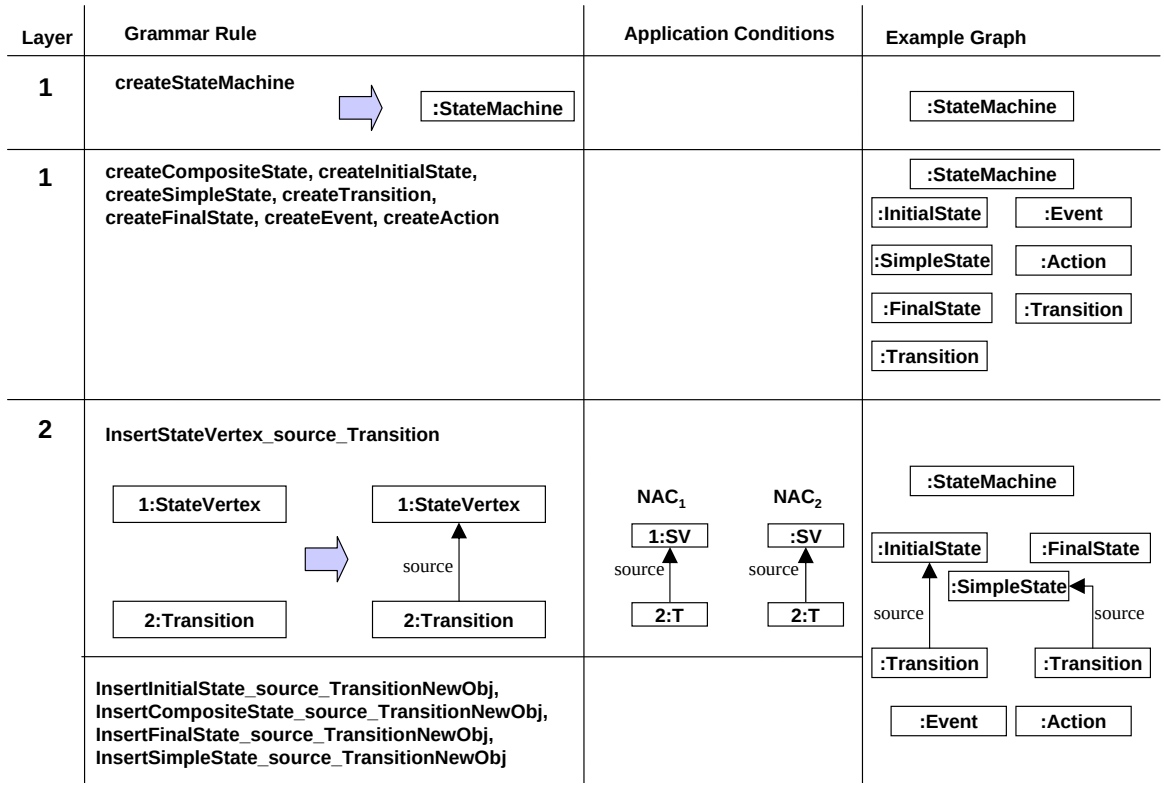


Figure 10 Example grammar rules 1

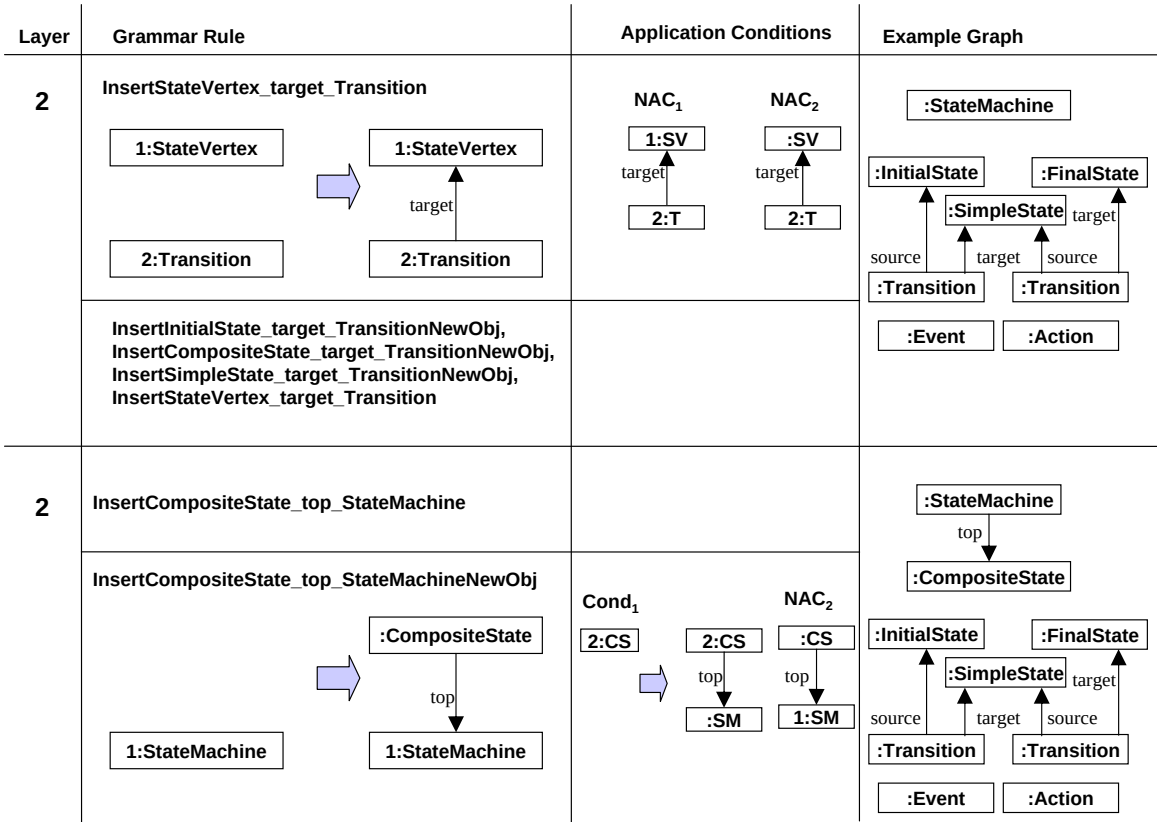


Figure 11 Example grammar rules 2

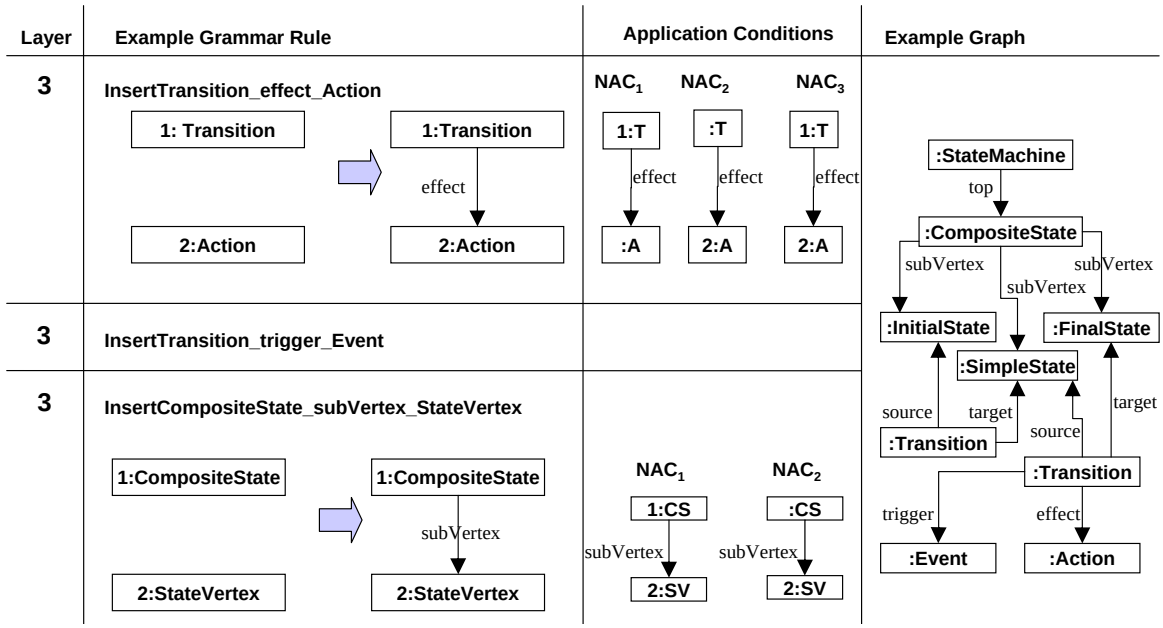


Figure 12 Example grammar rules 3

$\forall e \in TGI_E \wedge \forall v \in ctp_G^{-1}(t)$ *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 2 Considering the example graph in Figure 12, the multiplicities for edge type subvertex are fulfilled: For the only composite state c $|ctp^{-1}(\text{subvertex}) \cap src^{-1}(c)| = 3 \in [0, *]$, since three edges of type subvertex start in c. For all state vertices s $|ctp^{-1}(\text{subvertex}) \cap tgt^{-1}(s)| \leq 1 \in [0, 1]$, since each s has an incoming edge of type subvertex. The composite state is not subvertex of any other vertex and all other state vertices are subvertices of the only 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 depending 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 6 - 9 and are formalized in the obvious way according to Def. 4.*

- $R_1 = \{$*createE'* $| \forall E' \in TGI_N \wedge E' \notin Abs\}$ *with rules* *createE'* *as in Figure 6*
- $R_2 = R_{21} \cup R_{22} \cup R_{23}$ *with*
  $R_{21} = \{$*insertE_a_A* $| \forall A, E \in TGI_N, a \in TGI_E$: *with* $(m_{src}(a) = [1, 1] \vee m_{tgt}(a) = [1, 1])\}$
  $R_{22} = \{$*insertE'_a_ANewObj* $| \forall A, E \in TGI_N, a \in TGI_E$: *with* $(m_{src}(a) = [1, 1] \vee m_{tgt}(a) = [1, 1]) \wedge E' \in clan(E) \wedge E' \notin Abs\}$
  $R_{23} = \{$*insertE_a_A'NewObj2* $| \forall A, E \in TGI_N, a \in TGI_E$: *with* $(m_{src}(a) = [1, 1] \vee m_{tgt}(a) = [1, 1]) \wedge A' \in clan(A) \wedge A' \notin Abs\}$
  with rules *insertE_a_A, insertE'_a_ANewObj, and insertE'_a_ANewObj2* *as in Figure 7 - 8*
- $R_3 = \{$*insertE_a_A* $| \forall A, E \in TGI_N, a \in TGI_E$ *with* $m_{src}(a) \neq [1, 1] \wedge m_{tgt}(a) \neq [1, 1]\}$ *with rules* *insertE_a_A* *as in Figure 9*

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 [19].

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 $L(IGGG) = L(TGI_{mult})$.*

Proof See [19].

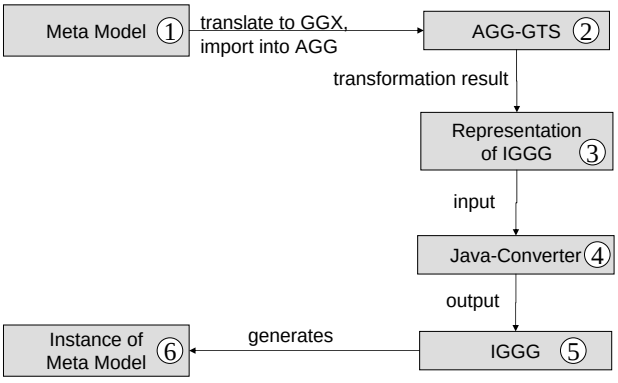


Figure 14 From meta model to instance generating graph grammar

6 Application and Tool Support

In this section, we describe application and tool support for instance-generating graph grammars. We first explain how, given a meta model for a language, an IGGG can be automatically generated, using a model transformation encoded in the graph transformation tool environment AGG [7, 38]. We then show how the instance-generating graph grammar for the statecharts meta model can be used to generate arbitrary statechart instances. Finally, we discuss the application of ICGGs to complete incomplete instance models.

6.1 Generation of the IGGG

Automatic derivation of instances from meta models is a complex task which needs tool support. We have automated the construction of an IGGG by providing a model transformation that derives an IGGG from a meta model.

The general procedure is shown in Figure 14. First the existing meta model (for example the statecharts meta model) has to be modeled using some CASE tool such as Rational Software Architect [1], Poseidon [11], OMONDO Eclipse-UML [2], etc. Nowadays most of these CASE tools support XMI [6] as export format. Unfortunately, each CASE tool supports its own variant of XMI such that an exchange of meta models between different CASE tools is not that easy. The meta model stored in XMI has to be first translated to GXL [5] (the standard exchange format for graphs), e.g. by stylesheet format transformations [4].

The GXL representation of the meta model is imported into the graph transformation system AGG. Inside the AGG tool, the model transformation for deriving from an existing meta model an instance-generating graph grammar is applied. The result of this translation (3) is a graph representation of a graph transformation system that has to be converted into an executable AGG graph grammar. The converter (4) which translates such a graph into an AGG graph grammar in GGX, the storing format used by AGG, is written in Java. The Java-Converter produces the instance generating graph grammar (5) that creates instance models (6) for the given meta model. Generated instance models could be exported to GXL and translated to various XMI formats to be used in other modelling tools.

This tooling support is considered for the example state machine meta model in the following.

The abstract syntax graph of the state machine meta model in GXL format shown in Figure 15 is the start graph for the AGG graph transformation system. The type graph of the meta graph transformation system shown in Figure 16 contains the source and target type graphs. This meta model representation is close to the XMI representation, but heavily simplified.

On the left of Figure 16, the meta model soure type graph is shown. In our case, a meta model consists of Classes, binary Associations where each association end (AssEnd) holds its multiplicity constraints, and inheritance relations between Classes. An Association is connected to its ends by s and t-edges correspondingly to source and target and just keep the reading information. An inheritance relation is represented by a parent edge from the child class to its parent class. In addition type Visited and edge type typeGraphParent are included. Type Visited is needed to keep information which classes and associations have already been processed, edge type typeGraphParent is needed to store the inheritance relations in the TypeGraph node of the target type graph. On the right-hand side of Figure 16 the target type graph is shown. It describes a graph representation of a graph transformation system. The root types are RuleSet and TypeGraph which corresponds to the structural configuration of graph transformation systems. A RuleSet contains a set of Rules which each have one left-hand side (LHS), one right-hand side (RHS), and can have negative application conditions (NAC). LHS, RHS, and NAC are graphs which are described by Nodes, possibly with Attributes, and Edges. The different parts of a rule are presented in an integrated way, i.e. a node occurring in LHS and RHS is presented only once but with two edges pointing to their containers. The TypeGraph contains a set of Nodes (possibly with Attributes), their inheritance relations (parent edge type), and Edges.

The rules of the meta graph transformation system transform a source meta model to a target graph grammar. First the

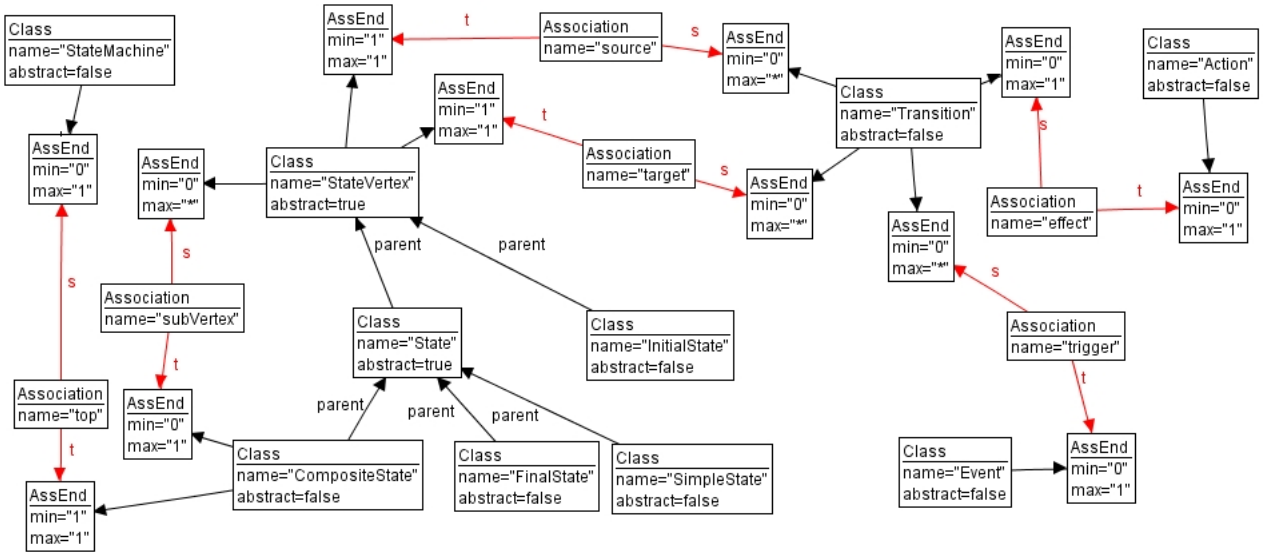


Figure 15 Abstract syntax graph of state machine meta model in Figure 1

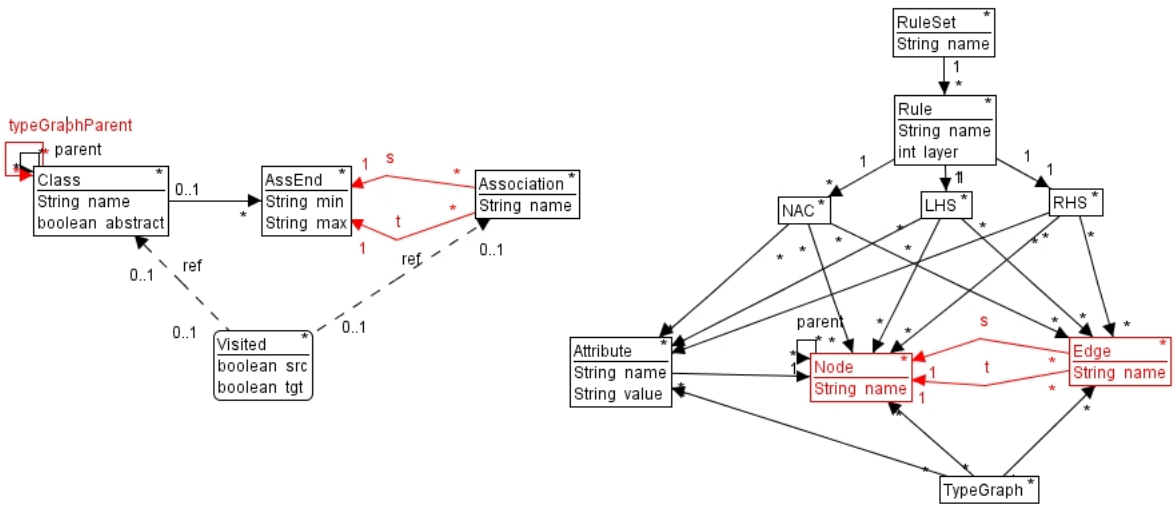


Figure 16 Type graph of the meta graph transformation system

TypeGraph of the target type graph is generated. The rules createNodeTypeWithoutBound and createEdgeType in Figure 17 and 18 create Node nodes in the target type graph for Class nodes in the source type graph, Edge nodes for Association nodes and parent edges for parent edges. Please note that both rules have an additional NAC each (not shown), checking if the right-hand side pattern can already be found in the graph. In this case, the corresponding rule has already been applied at the considered class node (Association node). A Class is unbound if it has no edge to an association end with multiplicity *1..1*. If it is bound we have to store this information by an additional edge to the helper node Bound, these node types are created by similar rules.

Then the rules of the instance generating graph grammar are built. For each Class node in the source instance a Rule node with name createObject, an empty left-hand side and a Node node with the given class name in the right-hand side is generated, see Figure 19. Again note that this rule has an additional NAC (which is not shown), checking if the right-hand side pattern can already be found in the graph. These rules belong to layer 1.

For each Association node the corresponding rules are generated as described in Section 4.

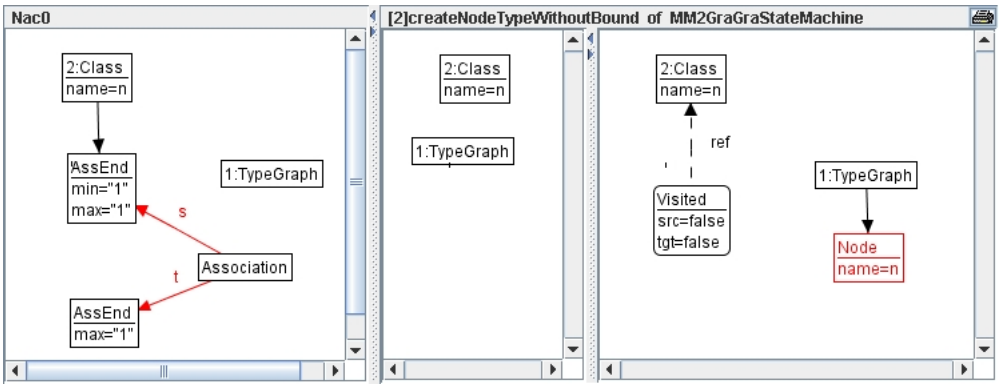


Figure 17 Creation of Nodes

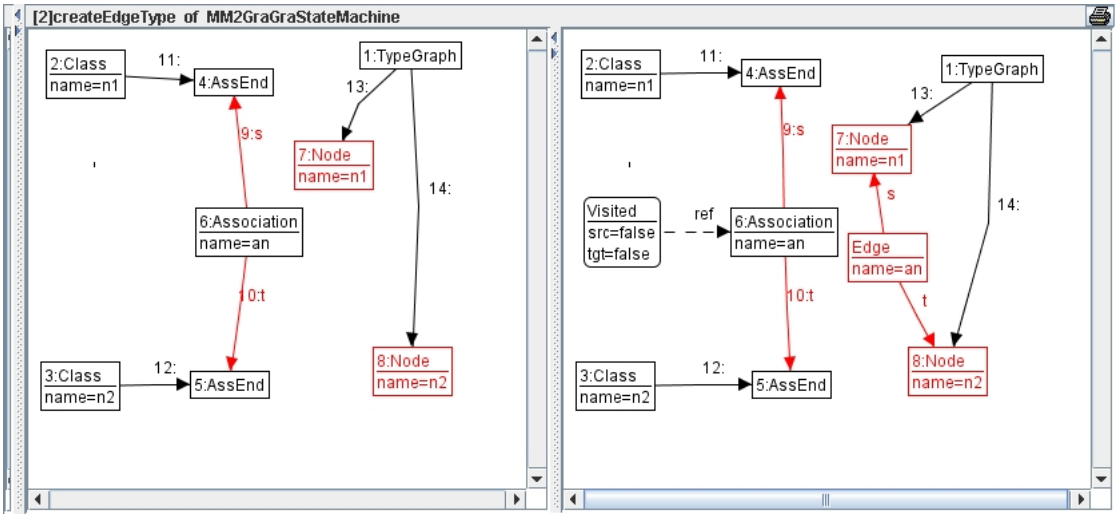


Figure 18 Creation of Edges to represent associations

The result of this graph transformation is a graph representation of a graph transformation system that has to be converted into an executable AGG graph grammar. The Java converter translates this graph into an AGG graph grammar using the AGG application programming interface methods to read the AGG graph representing the graph transformation system and to create corresponding rules and types in AGG accordingly. The result is the instance generating graph grammar that creates instance models for the given meta model. The converting tool as well as the complete description and implementation of two examples are available at `http://tfs.cs.tu-berlin.de/agg/MM2GraGra`.

6.2 Example generation of a statechart instance

In the following, we will discuss the practical results that we obtained when generating statechart instances using the generated IGGG.

Figure 20 shows an instance graph as abstract syntax graph that has been generated by the IGGG for the statechart meta model in Fig. 1. It shows an instance of the class StateMachine, connected to a top CompositeState which contains a CompositeState. This CompositeState contains two FinalStates and one InitialNode. In addition, the instance graph contains a number of Transitions, Events and Actions. When comparing the instance graph with the metamodel for statecharts shown in Figure 1 we realize that the instance graph is indeed an instance of this metamodel. As such it could be converted to the concrete syntax of statecharts automatically to contain a concrete statechart.

However, there are two problems. The first problem arises because the generation has not ensured OCL constraints formulated over the metamodel in Figure 1. For example, in the UML specification, there is an OCL constraint that forbids cyclic subvertex dependency. As we do not currently take OCL constraints into account during the generation, the instance might violate such OCL constraints which results into an invalid statechart instance. How to deal with OCL constraints in the generation is discussed in Section 7.

A second problem arises because in the generation we do not ensure that certain classes are instantiated more often than others. Typically, a statechart contains many more simple states than final states or initial states. For example, the instance graph generated in Figure 20 does not contain a single SimpleState. This leads to artificial statecharts that might look odd when comparing them to those statecharts that we are used to. To restrict the number of possible instances, fur-

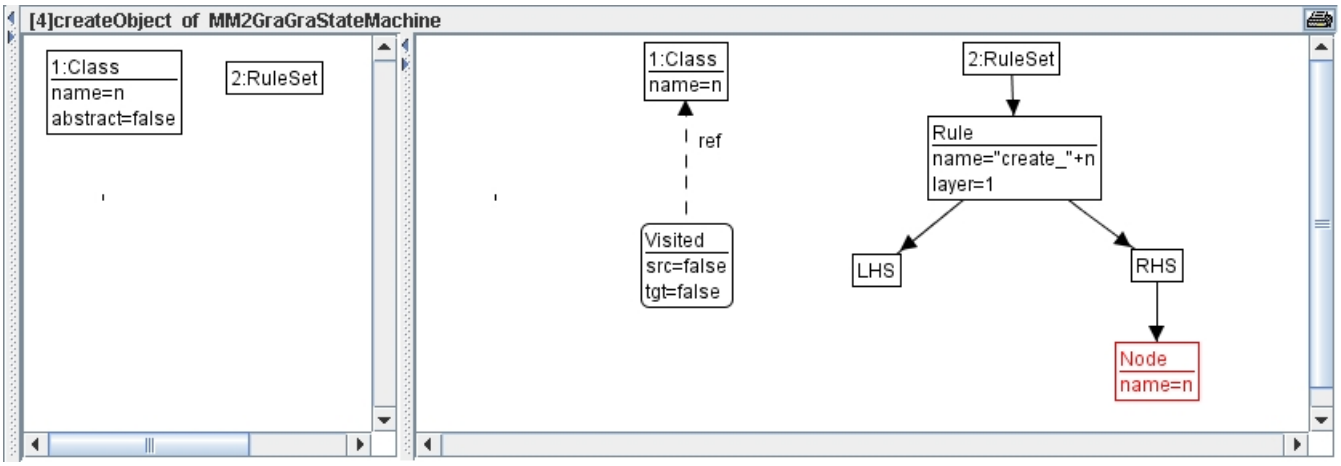


Figure 19 createObject rule for creating Nodes in the instance graph

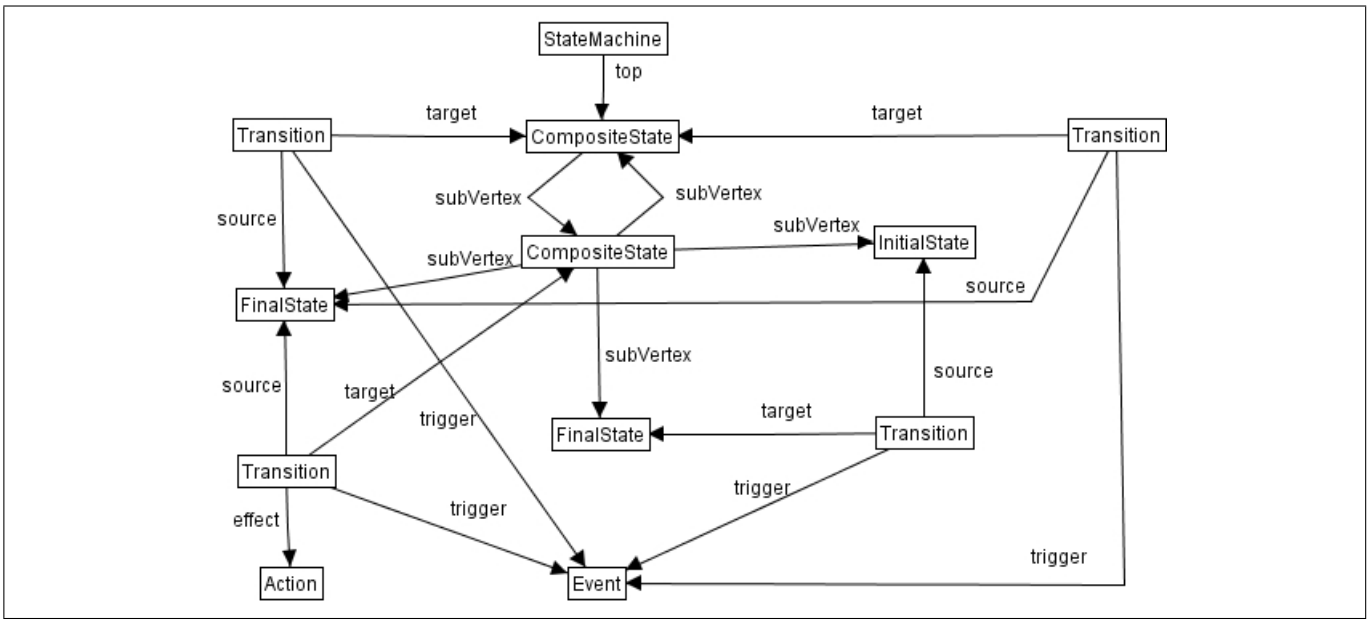


Figure 20 1st sample instance graph generated by the IGGG

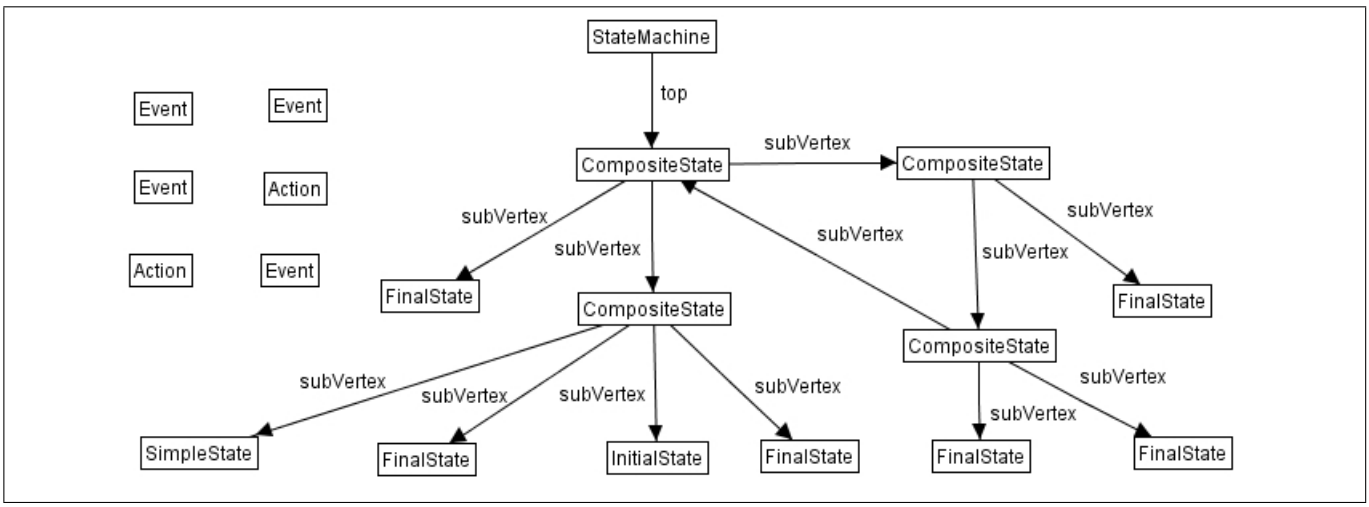


Figure 21 2nd sample instance graph generated by the IGGG

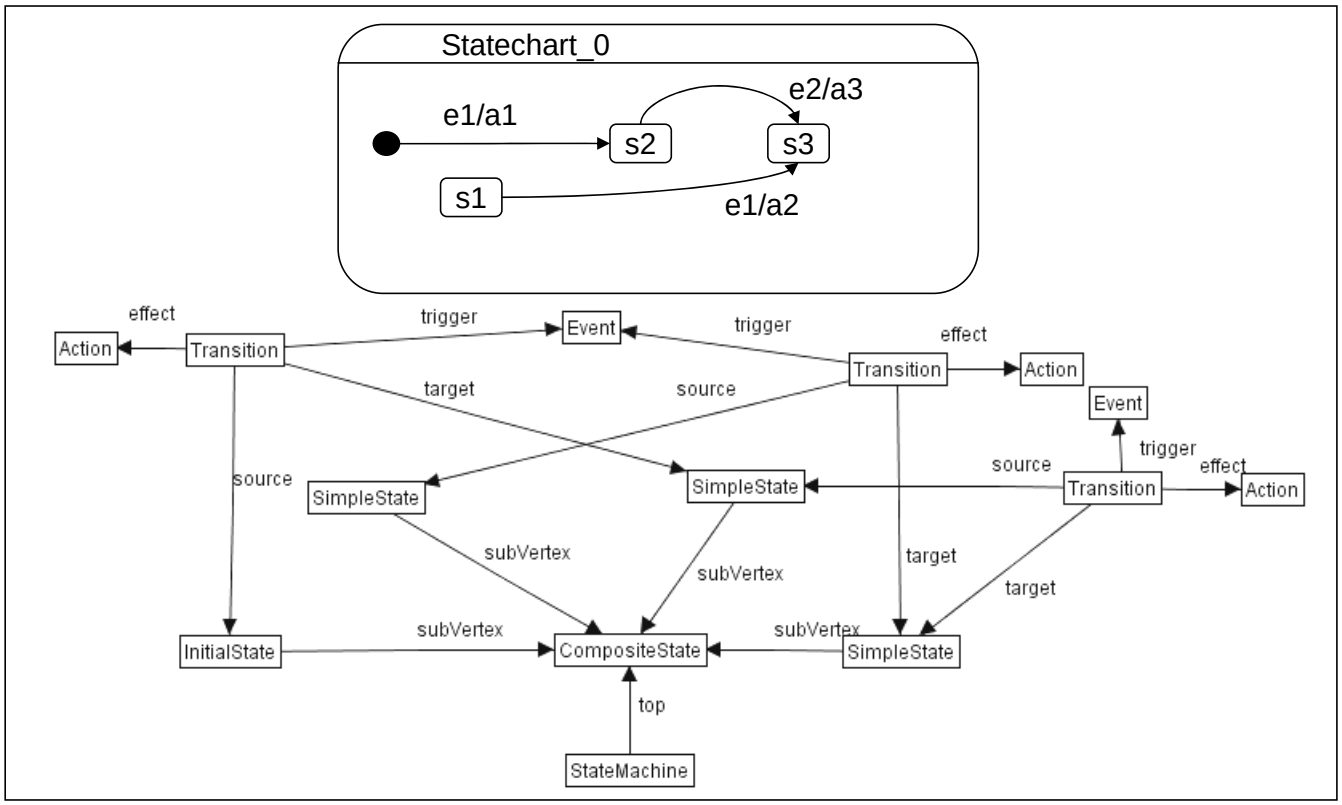


Figure 22 3rd sample instance graph generated by the IGGG in abstract and concrete syntax

ther constraints may be added to the metamodel. A related problem that occurs in this context is that instances of several classes need not be connected to other parts of the generated instance graph. For example, according to the metamodel in Figure 1, an instance of the class Event can be connected to any number of Transition objects, including zero. This leads to valid instance graphs such as the one shown in Figure 21, where Actions and Events stay unconnected. This problem could be resolved by changing the target multiplicities of trigger from 0..1 to 1.

Node multiplicity constraints can be defined to ensure that reasonable statecharts are generated. These multiplicity constraints can be translated to application conditions as shown in [37] and would thus be checked during executing rules of layer 1 and during creation of new objects in layer 2. Multiplicity constraints could also include constraints for requiring certain relations between the numbers of instances to hold. For example, it could be defined that each composite state has to contain at least one simple state. Realistic multiplicity constraints could be obtained by mining or harvesting existing statecharts.

Figure 22 shows a statechart in abstract syntax and in the corresponding concrete syntax that we generated using AGG, with the following maximal node multiplicities: At most one StateMachine, CompositeState, InitialState, at most three SimpleStates, at most four Transitions, and no FinalState. For Events and Actions, we allowed at most 3. Figure 23 shows another statechart that we generated with the following maximal node multiplicities: At most one StateMachine, CompositeState, FinalState and InitialState, at most three SimpleStates, and at most four Transitions. For Events and Actions, we allowed at most five. In both cases, we have imposed additional association constraints in order to minimize unconnected classes such as unconnected events. Note that for readability reasons we have manually named the statecharts, states, events and actions.

From these two examples we can conclude that our generation approach can be adapted to generate typical instances. The equivalence proof in Section 5 shows that all instances can be generated by IGGGs, even the odd ones usually not thought of. In particular the odd ones can prove beneficial when using generated statecharts for model transformation testing purposes.

6.3 Completion of statechart instances

Beside instance generation, instance completion may also play an important role and could be performed with our approach. This is useful for the generation of instances of a given meta model containing a specific pattern, e.g., all instances having five transitions, or all instances with exactly one transition starting at a specific class. So the instances expected to be particular test cases can be generated for a specific test suite.

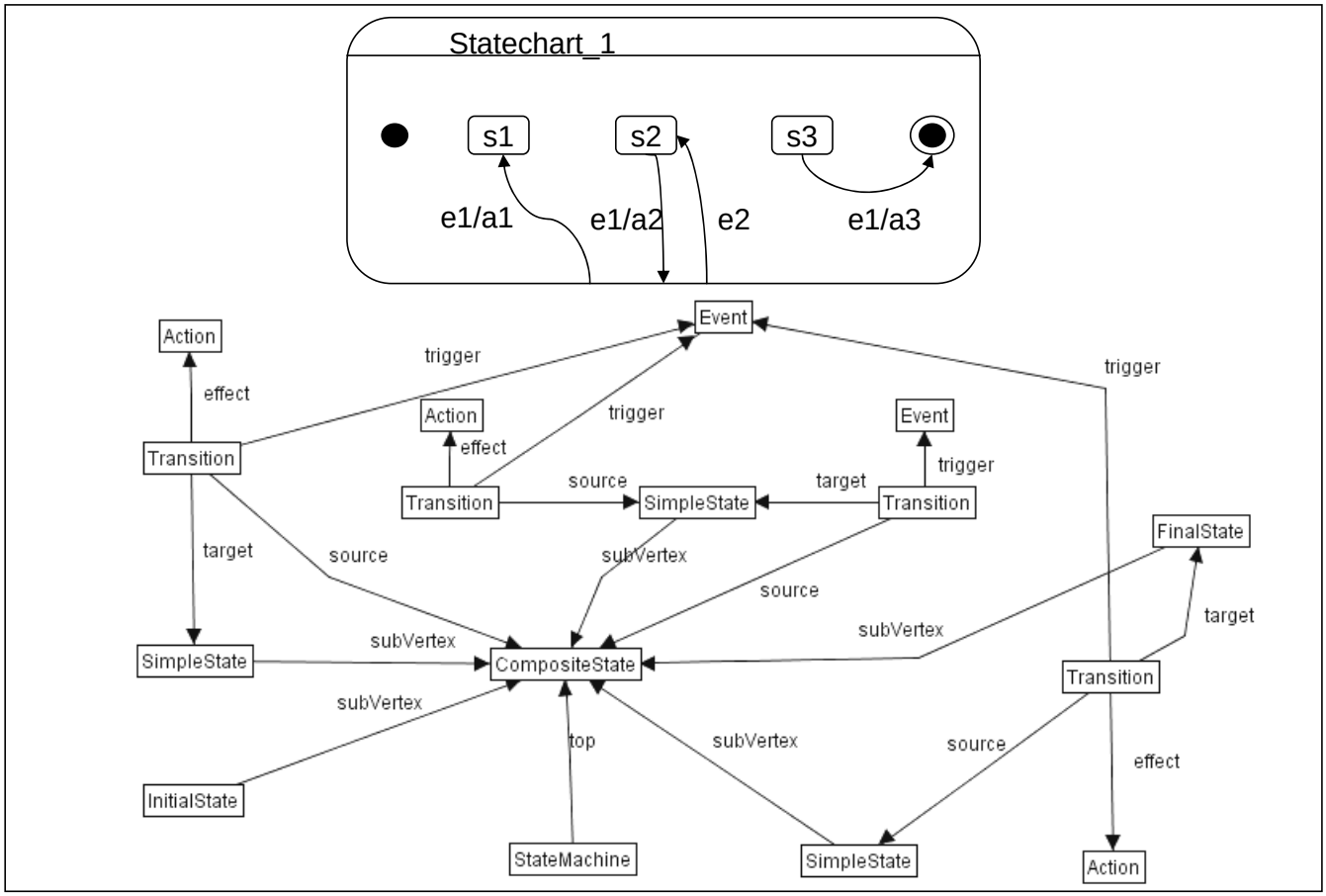


Figure 23 4th sample instance graph generated by the IGGG in abstract and concrete syntax

First it has to be checked whether the given instance is valid, i.e. if it is typed over the type graph of the instance generating graph grammar. This is done by importing the instance as start graph into the instance generating graph grammar. AGG checks the typing if the type graph is enabled.

Additionally, the given instance must not violate the upper multiplicities given by the meta model. They can be expressed by graph constraints, what is shown in [37]. AGG supports graph constraints, each multiplicity can be defined as atomic constraint, they are combined to one constraint which is checked.

If the imported instance fulfills the graph constraint and is typed correctly, it can be completed to a valid meta model instance by applying the rules of layer 2. Thereafter, we ensure that the instance also fulfills the lower multiplicities. The rules in layer 3 ensure that all valid instances are generated.

7 Extensions of the Approach

So far, we considered a kind of meta models that is restricted in a certain sense: We have not considered textual properties such as attributes, and special features such as loop associations and arbitrary multiplicity constraints. Moreover, well-formedness rules in form of OCL constraints have not yet been investigated. In this section, we discuss our approach to instance generation in the context of these extensions.

7.1 Textual properties

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 also be included in certain attributes.

7.2 Special Features

The instance generating graph grammar has not considered meta models that contain loop associations, singleton classes and arbitrary multiplicity constraints. In the following, we briefly describe how our approach can also deal with meta models containing such features.

Loop associations can be generated by just extending the rule set of the instance generating graph grammar. Similarly to the rules in Figure 7 - 9 a new association is added, but class E is equal to class A. Thus, both instances in each rule are of type E, or are even the same instance. The negative application conditions have to be adapted accordingly.

If the meta model contains singleton classes, the create rule for the corresponding instance has to have an additional application condition to ensure that at most one instance of this class is created.

In our approach, we do not allow associations with multiplicity constraints of the form $m..n$ with $m, n \neq 0, 1$ and $*$. If any such multiplicity constraints would be allowed, it might happen that certain combinations of multiplicities do not have legal instances. To find out whether legal instances exist, a system of inequations has to be solved (see e.g. [30]). If legal instances exist, they can be generated by similar rules as presented above. To ensure a minimal number of instances, rules insertE'_a_ANewObj have to adapted such that not only one A has to exist, but as many instances as the lower bound prescribes. Moreover, to ensure the maximal number of instances in application conditions, we have to specify them in a negative application condition which is no problem for small numbers, but can become inconvenient for larger numbers. However, the experience showed that larger numbers in multiplicity constraints are extremely seldom and may be subject to future work.

7.3 Well-formedness rules

The graph grammar introduced in Section 4 ensures multiplicity constraints of the meta model, but well-formedness rules are not considered until now. Ensuring OCL constraints can be done in two ways: In a first solution, constraints are checked once the overall derivation of an instance model has terminated. However, this leads to the generation of a large number of non-valid instances in between. A more promising approach is to take the constraints into consideration during the derivation process: For each meta model class the corresponding OCL constraints can be identified. Following this line, OCL constraints are first translated to graph constraints which are further translated to application condition of instance generating rules in the sense of [17].

We started to work on this second line of constraint checking and showed in [40] how a restricted form of OCL constraints can be translated to graph constraints. We restricted OCL constraints to equality, size, and attribute operations for navigation expressions, called *restricted OCL constraints*. In the following, we introduce this restricted form of OCL constraints for which we have already a clear idea how to translate them to graph constraints as defined above. A precise definition of that translation is still future work.

Restricted OCL constraints The *restricted OCL constraints* that can be translated are divided into *atomic navigation expressions* and *complex navigation expressions*.

Atomic navigation expressions are OCL expressions that

- express equivalent navigations, like self.assoc1=self.assoc2.assoc3,
- end with operation *size()* (if the result is compared with constants),
- end with operations *isEmpty()*, *notEmpty()* or *isUnique()*, or
- end with attribute operations (not considered explicitly in the paper).

The navigation expressions contain navigation along association ends or association classes only. Atomic navigation expressions can be transformed into basic graph constraints of the form $\exists x$ or boolean formulae over basic graph constraints.

Operation *size()* can be translated into a Boolean graph constraint that is composed of two basic graph constraints. The first constraint ensures that there exist the minimum number of association ends, the second prohibits the existence of more than the constant value association ends. If the comparison operation is $\leq$ or $\geq$ the OCL constraint would be translated into just one graph constraint.

Operations *isEmpty()* and *notEmpty()* can be translated back to a *size()* operation: self.assoc1->isEmpty() is translated back to self.assoc1->size()=0, self.assoc1->notEmpty() to self.assoc1->size()>=1.

Collection operation isUnique() can be translated into a *size()* operation, if the body of the collection operation is a navigation expression ending at an instance set: self.assoc1->isUnique(navexp) is translated back to self.assoc1.navexp->size()<=1.

Operations *isTypeOf* and *isKindOf* can also be used inside of navigation expressions, since graph constraints are formulated based on typed graph morphisms and, especially on clan morphisms (see [37]).

Complex navigation expressions are characterized like this: Atomic navigation expressions are complex navigation expressions. Given complex navigation expressions a, b and c, expressions not(a), a and b, a or b, a implies b, and if a then b else c are complex navigation expressions.

Although being restricted, all OCL constraints for the simple statechart meta model example in Figure 1 are of this form and thus can be translated to graph constraints (see [40]). In [37], we showed how graph constraints (which can also contain abstract types) are translated to application conditions. This enables to take into account OCL constraints during the derivation process.

In future work, OCL constraints and graph constraints have to be further compared concerning their expressiveness. It is expectable that not all OCL constraints can be translated to graph constraints. Those OCL constraints are either checked by a constraint checker after the generation process (as indicated above), or they are translated not only to application conditions of rules, but to special rules themselves. To illustrate this idea at an example, consider e.g. the well-formedness rule for acyclic subvertex relations which can be expressed by the following OCL constraint:

```
context CompositeState inv:
not self.allSubVertices()->includes(self)
```

with additional operation:

```
CompositeState::allSubVertices():Set(StateVertex)
allSubVertices = subvertex->
union(subvertex->collect(v | v.allSubVertices()))
```

Unlike the rules in layer 1 (Figure 6), the creation of a new composite state is possible only, if another composite state is already there, or if this is not the case, the composite state is created as top state of the state machine. Creating composite states just by rules where parent nodes are required on the left-hand sides, an acyclic subvertex relation is granted.

8 Related Work

One closely related approach is the one by Alanen and Porres [8]: 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. The aim of their work is to bridge the gap between artifacts defined by a context-free 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 metaclasses, 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 cardinalities. This limitation is not surprising given the properties of context-free grammars. It 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. [22]. 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.

Courcelle [15] considers graph grammars in the context of monadic second order logic, a logical language which is favourite for its decidability properties and suitable for expressing graph properties. It is up to future work to investigate the literature for equivalence results of graph languages defined by graph grammars on the one hand, and by graph properties on the other hand.

Formal methods such as Alloy [3, 24] 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. As the grammar rules are explicitly mentioned, our approach also allows applications which require interaction during the derivation process or only a partial derivation process. One possible application is the completion of a model instance by applying only a partial set of grammar rules to it. One advantage of our instance generation approach over approaches relying on formal methods such as Alloy is that our instance generation process is close to instance generation in practice, while being fully formal, and avoids a translation to a formal method such as Alloy. Although such a translation is already available (see Anastasakis et al. [9]), this gives rise to the problem how results can be visualized in a user-friendly manner. For example, an instance found in the Alloy language needs to be translated back into UML. This additional overhead is avoided by our approach which operates more directly on meta models and their instances.

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 [21]. In the context of graph grammars, Jeltsch and Kreowski [25] 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 [13, 23, 33] 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.

9 Conclusion and Future Work

Currently, the widespread approach of defining visual languages by a meta model and a set of OCL constraints has one main disadvantage: Such a definition is not constructive. This represents a severe disadvantage for applications where an operational description is required i.e. for generating a large set of instances automatically. Possible applications include automated testing of model transformations or automatic editor generation.

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. Being able to

generate an instance-generating graph grammar for an arbitrary meta model closes an important technology gap and allows the adoption of well-known techniques for grammar-based languages also for languages defined by meta models.

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. We have illustrated our approach for a simplified statechart meta model and shown that the automatic generation of instances is possible. With regards to completeness, we have used the theory of typed graph transformation with inheritance to show that the instance sets generated by an IGGG and those induced by the corresponding type graph with multiplicities are equal.

In Section 7, we discussed extensions of our approach which have to be considered in future work. These are especially well-formedness rules in form of OCL constraints. We sketched how a restricted set of OCL constraints can be translated into graph constraints and explained how they can be taken into account during instance generation. The precise translation of metamodels with loops associations, arbitrary multiplicity constraints, and well-formedness rules to an equivalent instance generating graph grammars is left to future work.

Further work is needed in order to elaborate on possible applications of our technique: Besides the presented example for statecharts generation, we like to test instance generation for further and especially larger meta models, last but not least, for showing that this approach can scale. For testing model transformations, techniques are needed that allow the generation of selected instance models that represent a suitable diversity of all possible models. For editor generation, it needs to be explored how the graph grammar generated from a meta model can be used to test the usability of the editor.

Acknowledgements

This work has been partially sponsored by the IST-2005-16004 Integrated Project SENSORIA (Software Engineering for Service-Oriented Overlay Computers), and the German Research Foundation with project "Application of graph transformation to visual modeling languages". The authors would also like to thank their former colleague Jessica Winkelmann for her contributions to the presented work and the referees for their useful comments.

References

1. IBM Rational Software Architect V7.0. Available at `http://www.ibm.com/software/awdtools/architect/swarchitect/index.html`.
2. OMONDO EclipseUML 3.2.0 Studio. Available at `http://www.omondo.com`.
3. *The Alloy Analyzer 4.0* `http://alloy.mit.edu/`.
4. XSL Transformations (XSLT) Version 1.0. Available at `http://www.w3.org/TR/xslt`.
5. *GXL* `http://www.gupro.de/GXL`, 2005.
6. *XMI – XML Metadata Interchange Version 2.0*, 2005.
7. AGG Homepage. `http://tfs.cs.tu-berlin.de/agg`.
8. 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.
9. K. Anastasakis, B. Bordbar, G. Georg, and I. Ray. UML2Alloy: A Challenging Model Transformation. In G. Engels et al., editor, *Proceedings ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems*, volume 4735 of *Lecture Notes in Computer Science*, pages 436–450. Springer, 2007.
10. 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, pages 214–228. Springer LNCS, 2004.
11. M. Boger, T. Sturm, E. Schildhauer, and E. Graham. *Poseidon for UML Users Guide*. Gentleware AG, 2003. Available under http://www.gentleware.com.
12. A. S. Boujarwah and K. Saleh. Compiler test case generation methods: a survey and assessment. *Information and Software Technology*, 39(9):617–625, 1997.
13. M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, editors. *Model-Based Testing of Reactive Systems, Advanced Lectures [The volume is the outcome of a research seminar that was held in Schloss Dagstuhl in January 2004]*, volume 3472 of *Lecture Notes in Computer Science*. Springer, 2005.
14. 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.
15. B. Courcelle. The expression of graph properties and graph transformations in monadic second-order logic. In G. Rozenberg, editor, *Handbook of graph grammars and computing by graph transformations, vol. 1: Foundations*, pages 313–400. World Scientific, New-Jersey, London, 1997.
16. 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.
17. H. Ehrig, K. Ehrig, A. Habel, and K.-H. Pennemann. Constraints and application conditions: From graphs to high-level structures. In F. Parisi-Presicce, P. Bottoni, and G. Engels, editors, *Proc. 2nd Int. Conference on Graph Transformation (ICGT'04)*, LNCS 3256, pages 287–303, Rome, Italy, October 2004. Springer.
18. H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. EATCS Monographs in Theoretical Computer Science. Springer, 2006.
19. 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.

20. K. Ehrig, J. M. Küster, G. Taentzer, and J. Winkelmann. Generating Instance Models from Meta Models. In R. Gorrieri and H. Wehrheim, editors, *Formal Methods for Open Object-Based Distributed Systems, 8th IFIP WG 6.1 International Conference, FMOODS 2006, Bologna, Italy, June 14-16, 2006, Proceedings*, volume 4037 of *LNCS*, pages 156–170. Springer, 2006.
21. 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.
22. M. Gogolla, J. Bohling, and M. Richters. Validating UML and OCL Models in USE by Automatic Snapshot Generation. *Software and Systems Modeling*, 4(4):386–398, 2005.
23. A. Hartman and K. Nagin. The AGEDIS Tools for Model Based Testing. In N. J. Nunes, B. Selic, A. da Silva, and J. Álvarez, editors, *UML Satellite Activities, Revised Selected Papers*, volume 3297 of *Lecture Notes in Computer Science*, pages 277–280. Springer, 2005.
24. D. Jackson. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, 2002.
25. 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.
26. G. Karsai, A. Agrawal, F. Shi, and J. Sprinkle. On the Use of Graph Transformation in the Formal Specification of Model Interpreters. *Journal of Universal Computer Science*, 9(11):1296–1321, 2003.
27. A. Kirshin, D. Dotan, and A. Hartman. A UML Simulator Based on a Generic Model Execution Engine. In T. Kühne, editor, *MoDELS Workshops*, volume 4364 of *Lecture Notes in Computer Science*, pages 324–326. Springer, 2006.
28. J. M. Küster. Definition and validation of model transformations. *Software and Systems Modeling (SoSyM)*, 5(3):233–259, 2006.
29. J. M. Küster and M. Abd-El-Razik. Validation of Model Transformations - First Experiences Using a White Box Approach. In T. Kühne, editor, *MoDELS Workshops*, volume 4364 of *Lecture Notes in Computer Science*, pages 193–204. Springer, 2007.
30. A. Maraee and M. Balaban. Efficient Reasoning About Finite Satisfiability of UML Class Diagrams with Constrained Generalization Sets. In D. H. Akehurst, R. Vogel, and R. F. Paige, editors, *Model Driven Architecture- Foundations and Applications, Third European Conference, ECMDA-FA 2007, Haifa, Israel, June 11-15, 2007, Proceedings*, volume 4530 of *Lecture Notes in Computer Science*, pages 17–31. Springer, 2007.
31. T. Mens. On the Use of Graph Transformations for Model Refactoring. In R. Lämmel, J. Saraiva, and J. Visser, editors, *Generative and Transformational Techniques in Software Engineering*, volume 4143 of *Lecture Notes in Computer Science*, pages 219–257. Springer, 2006.
32. J.-M. Mottu, B. Baudry, and Y. Le Traon. Mutation Analysis Testing for Model Transformation. In A. Rensink and J. Warmer, editors, *Model Driven Architecture - Foundations and Applications, Second European Conference, ECMDA-FA 2006, Bilbao, Spain, July 10-13, 2006, Proceedings*, volume 4066 of *LNCS*, pages 376–390. Springer, 2006.
33. C. Nebut, F. Fleurey, Y. Le Traon, and J.-M. Jézéquel. Automatic Test Generation: A Use Case Driven Approach. *IEEE Trans. Software Eng.*, 32(3):140–155, 2006.
34. Object Management Group. *MDA Guide Version 1.0.1*, June 2003.
35. Object Management Group (OMG). *UML 2.0 Superstructure Final Adopted Specification. OMG document pts/03-08-02*, August 2003.
36. Object Management Group (OMG). *OCL 2.0 Specification. OMG document ptc/2005-06-06*, June 2005.
37. 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.
38. G. Taentzer. AGG: A Graph Transformation Environment for Modeling and Validation of Software. In J. Pfaltz, M. Nagl, and B. Boehlen, editors, *Application of Graph Transformations with Industrial Relevance (AGTIVE'03)*, pages 446 – 456. LNCS 3062, Springer, 2004.
39. D. Varró, S. Varró-Gyapay, H. Ehrig, U. Prange, and G. Taentzer. Termination Analysis of Model Transformations by Petri Nets. In A. Corradini, H. Ehrig, U. Montanari, L. Ribeiro, and G. Rozenberg, editors, *Graph Transformations, Third International Conference*, volume 4178 of *LNCS*, pages 260–274. Springer, 2006.
40. J. Winkelmann, G. Taentzer, K. Ehrig, and J. M. Küster. Translation of Restricted OCL Constraints into Graph Constraints for Generating Meta Model Instances by Graph Grammars. In *Proceedings of the 5th International Workshop on Graph Transformations and Visual Modeling Techniques (GT-VMT'06)*, pages 153–164, 2006.