Integrating Meta-modelling Aspects with Graph Transformation for Efficient Visual Language Definition and Model Manipulation

Roswitha Bardohl[1], Hartmut Ehrig[1], Juan de Lara[2], and Gabriele Taentzer[1]

[1] Computer Science Department
Technische Universitat Berlin
Berlin, Germany
{rosi,ehrig,gabi}@cs.tu-berlin.de
[2] Escuela Politécnica Superior
Ingeniería Informática
Universidad Autónoma de Madrid
Juan.Lara@ii.uam.es

Abstract. Visual languages (VLs) play a central role in modelling various system aspects. Besides standard languages like UML, a variety of domain-specific languages exist which are the more used the more tool support is available for them. Different kinds of generators have been developed which produce visual modelling environments based on VL specifications. To define a VL, declarative as well as constructive approaches are used. The meta modelling approach is a declarative one where classes of symbols and relations are defined and associated to each other. Constraints describe additional language properties. Defining a VL by a graph grammar, the constructive way is followed where graphs describe the abstract syntax of models and graph rules formulate the language grammar.
In this paper, we extend algebraic graph grammars by a node type inheritance concept which opens up the possibility to integrate both approaches by identifying symbol classes with node types and associations with edge types of some graph class. In this way, declarative as well as constructive elements may be used for language definition and model manipulation. Two concrete approaches, the GenGED and the AToM^3 approach, illustrate how VLs can be defined and models can be manipulated by the techniques described above.

1 Introduction

Visual languages (VLs) play a central role in modelling various system aspects. One, if not the main visual modelling language is the UML [19] which integrates a number of different diagram techniques, useful to describe structural as well as behavioural aspects of object-oriented software systems. Although the UML defines a standard in visual modelling, there are of course various further visual modelling techniques, often domain-specific and often for specific aspects. Especially for those domain-specific solutions which are not widely known, a generator for visual modelling environments is useful. After specifying the VL in mind, a supporting modelling environment consisting of visual editors, simulators, compilers and animators is generated automatically and does not have to be coded by hand. Thus, rapid prototyping is supported.

There are mainly two different lines to define a VL: the declarative way and the constructive way. UML is defined by the Meta Object Facilities (MOF) approach [19] which uses classes and associations to define symbols and relations of a VL. Within this meta modelling approach, multiplicities and OCL constraints [23] are additionally used to formulate desired language properties. While constraint-based formalisms provide a declarative approach to VL definition, grammars are more constructive, i.e. closer to the implementation. In [18] for example, textual as well as graph grammar approaches are considered for VL definition. Due to its appealing visual form, graph grammars can directly be used as high-level visual specification mechanism for VLs [4]. Defining the abstract syntax of visual forms as graphs, a graph grammar defines directly the language grammar. The induced graph language determines the corresponding VL. Visual language parsers can be immediately deduced from such a graph grammar. Furthermore, abstract syntax graphs are also the starting point for model simulation and transformation, i.e., model manipulation [5, 10, 22, 13]. Also here, it is very natural to use graph transformation to come up with a high-level and constructive specification.

In this paper, we consider the integration of meta modelling with graph transformation. As common basis we take into account the types of visual symbols and relations within a VL, i.e. the visual alphabet. While constraints describe additional requirements on this alphabet, transformation rules formulate a constructive procedure. In the MOF approach, classes of symbols can be inherited, meaning that their attribute lists and their associations are also present at all their descendants. Considering graph transformation on the other hand, an additional type graph [8] is used to ensure a certain type safety on nodes and edges. Supporting node type inheritance in addition, leads to a more dense form of a graph transformation system, since similar transformation rules can be abstracted into one. We believe this work can be very valuable for the Model Driven Arquitecture [19] (MDA), where model transformation plays a central role. In Section 2, we present algebraic graph transformation with node type inheritance facilities and show how this kind of graph transformation can be flattened to simply typed graph transformation.

The MOF and the graph transformation approach can be integrated by identifying symbol classes with node types and associations with edge types. In this way, declarative as well as constructive elements may be used for language definition, but it is still open how single parts of a VL specification are defined. In Section 3, we discuss two possible approaches, the AToM^3 and the GenGED approach, which are quite different to each other.

All new concepts are illustrated at a running example which is a variant of UML Statecharts. We focus on the abstract syntax definition of the language as well as the simulation of concrete state models. Last but not least, we compare our approaches to further ones in the literature.

2 Typed Graph Transformation with Node Type Inheritance

In this section we present our new concepts of typed graph transformation with node type inheritance. Due to the space limitations, we omitted all proofs and details. The interested reader is asked to consult [2].

2.1 Type and Instance Graphs

The basic idea for specifying node type hierarchies is to introduce a special kind of (directed) edges, hierarchy edges, into type graphs. The source node of a hierarchy edge is said to be a sub-type of the target node, which is called the former one's super-type. Nodes are marked either as *concrete* or *abstract*. In host graphs only nodes of concrete types shall occur, while graphs in rules may contain nodes of both types.

Definition 1. *(Type Graph with Inheritance) A type graph with inheritance is a triple* (TG, I, A) *consisting of a type graph* $TG = (N, E, s, t)$ *(with a set* N *of nodes, a set* E *of edges, a source and a target function* $s, t : E \to N$*), an inheritance graph* I *sharing the same set of nodes* N*, and a set* $A \subseteq N$*, called abstract nodes.*
For each node n *in* I *the* inheritance clan *is defined by* $clan_I(n) = \{n' \in N \mid \exists \text{ path } n' \xrightarrow{*} n \text{ in } I\}$ *where path of length* 0 *is included, i.e.* $n \in clan_I(n)$*.*
The sub-graph spanned by the hierarchy edges must be acyclic.

To benefit from the well-founded theory of graph transformation [8], type graphs with inheritance can be flattened to ordinary ones.

Definition 2. *(Closure of Type Graph with Inheritance) Given* (TG, I, A) *with* $TG = (N, E, s, t)$*, the* abstract closure *of* (TG, I, A) *is the graph* $\overline{TG} = (N, \overline{E}, \overline{s}, \overline{t})$ *with*

- $\overline{E} = \{(n_1, e, n_2) \mid n_1 \in clan_I(s(e)), n_2 \in clan_I(t(e)), e \in E\}$,
- $\overline{s}((n_1, e, n_2)) = n_1$,
- $\overline{t}((n_1, e, n_2)) = n_2$, *and*
- $proj_E((n_1, e, n_2)) = e$ *for* $e \in E$.

The graph $\widehat{TG} = \overline{TG}|_{N-A}$ *is called* concrete closure *of* (TG, I, A)*.*
Given a graph $G = (N, E, s, t)$ *and a set* $X \subseteq N$*, we denote by* $G|_X$ *the sub-graph* $(X, E_X = \{e \in E \mid s(e), t(e) \in X\}, s|_{E_X}, t|_{E_X})$*.*

The discrimination between the abstract and the concrete closure of a type graph is necessary, since there are instance graphs with respect to either one. The left-hand side (LHS) and right-hand side (RHS) of abstract rules are typed over the abstract closure, while ordinary host graphs and concrete rules (see section 2.2 for rules) are typed over the the concrete closure. Due to the existence of the canonical inclusion $inc_{TG} : \widehat{TG} \hookrightarrow \overline{TG}$ all graphs typed over $\widehat{TG}$ are also typed over $\overline{TG}$.

Definition 3. *(Instance Graph of Type Graph with Inheritance) An abstract instance graph* $(G, type)$ *of* (TG, I, A) *is an instance graph of* $\overline{TG}$*, i.e.* $(G, type \colon G \to \overline{TG})$*. Analogously, a concrete instance graph of* (TG, I, A) *is typed over* $\widehat{TG}$*.*

The choice of triples for the edges of a type graph's closure allows expressing a typing property with respect to the type graph with inheritance. The instance graph can be typed over the type graph with inheritance (for convenience) by a pair of functions, one assigning a node type to each node and the other one assigning an edge type to each edge. Both are defined canonically. A graph morphism is not obtained this way, but some mapping that will be introduced as *clan morphism*, uniquely characterizing the type morphism into the flattened type graph.

Definition 4. *(Clan Morphism) Given a type graph with inheritance* (TG, I, A)*,*
$type' \colon G \rightarrow TG$ *is a* clan-morphism*, if for all* $e \in G_E$ *holds*

- $type'_N \circ s_G(e) \in clan_I(s_{TG} \circ type'_E(e))$ *and*
- $type'_N \circ t_G(e) \in clan_I(t_{TG} \circ type'_E(e))$.

$type'$ *is called* concrete*, if* $type'_N(n) \notin A$ *for all* $n \in G_N$.

The notion of a *type refinement* is used in order to formalize the relationship between abstract and concrete rules as they are proposed in Section 2.2. It defines an order over possible typing morphisms for a given instance graph. A typing morphism is said to be *finer* than another one, if it assigns more concrete node types to the nodes of the instance graph.

Definition 5. *(Type Refinement)*
$(G, type' \colon G \rightarrow TG)$ *is a* type refinement *of* $(G, type \colon G \rightarrow TG)$*, if*

- $type'_N(n) \in clan(type_N(n))$ *for all* $n \in G_N$ *and*
- $type'_E = type_E$.

$type'$ *is respectively called* finer *than type, denoted* $type' \leq type$.

Applying graph transformation with node type inheritance to visual language definition, usually needs attributed nodes. Thus, we have to clarify how the concept of node type inheritance can be extended to node attributes. Assuming node type A has attributes, a descendant node type B inherits not only all adjacent edge types but also its attribute list. Of course, it should be possible to enlargen the inherited list by new attributes.

If we use attributes only as labels, i.e. they are not changed during a transformation, this kind of typed attributed graphs can be defined by ordinary typed graphs. (Potentially infinite) sets of data values are considered as nodes. They are called data nodes in contrast to object nodes denoting all other nodes of an attributed graph. Data nodes and object nodes are linked by attributes, i.e. edges with an object node as source and a data node as target. We assume that there are no edges starting at some data node. If this property is satisfied within the type graph, it also holds for the instance graphs due to the typing morphisms.

Summarising, graphs and graph transformation with node attributes which are not changed are already captured by our formalisation. If we need a more general attribution concept where computations can take place on attributes, future work is needed to extent the formal approach.

Example: Type Graph for a Statechart Variant. Fig. 1 shows a type graph with inheritance for a slightly modified sub-set of the Statecharts meta model proposed in the UML specification [19]. For space limitations, the following simplifications have been performed. Only PseudoStates of the *initial* kind (attribute *ind*) are considered, i.e., we eliminated classes *SynchState*, *StubState* and concurrent states and concentrate on *CallEvent* and *SignalEvent* classes. Events are associated to the transitions they trigger (and not to states). For simulation, objects need to receive events, so we modelled an event queue (by relationships *receives* and *next*); the last event is a special one depicting its end. Additionally, we added a relationship *current* to depict the state a particular object is in. Note how the triple (TG, I, A) has been expressed in a single graph, where the nodes of TG and I are the same, regular edges represent edges in TG, hollow arrow-head edges represent edges in I and the elements of A are represented in italics.

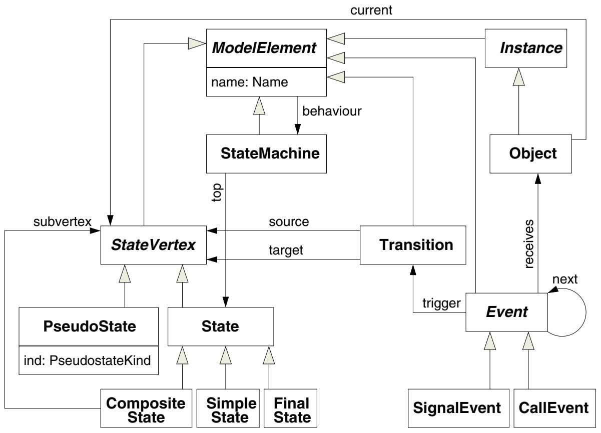

Fig. 1. Type graph with inheritance for a part of UML Statecharts.

2.2 Rules and Derivations

Transformations of graphs are described by graph rules. We follow the so-called *Double Pushout* approach to graph transformation [8]. It is desired to allow abstract node instances in rules, such that abstract rules actually represent a set of structurally similar rules, we call *concrete rules*. To get all concrete rules for an abstract rule, any combination of node replacements in the rule's LHS (being of concrete or abstract type) by instances of respective concrete sub-types (reflexive and transitive, i.e. the type's clan) must be considered. The rule morphism's image of an LHS node must always be replaced by an instance of the same type. The other nodes in the RHS remain the same and therefore must be instances of concrete types. Concrete rules are structurally equal to the abstract rule, their typing morphisms are finer (cf. Def. 5) than the ones of the abstract rule and are concrete clan morphisms.

Definition 6. *(Abstract and Concrete Rules)*

An abstract rule typed over a type graph TG with inheritance is given by $r = (L \xleftarrow{l} K \xrightarrow{r} R,\ type,\ NAC)$, where l and r are graph morphisms, type is a triple of typing clan morphisms $type = (type_L\colon L \to TG,\ type_K\colon K \to TG,\ type_R\colon R \to TG)$, and NAC is a set of triples $nac = (N, n, type_N)$ with N being a graph, $n\colon L \to N$ an injective graph morphism, and $type_N\colon N \to TG$ a typing clan morphism, such that the following conditions hold:

- $type_L \circ l = type_K = type_R \circ r$
- $type_{R,N}(R'_N) \cap A = \emptyset$, *where* $R'_N := R_N - r_N(K_N)$
- $type_N \circ n \leq type_L$ *for all* $(N, n, type_N) \in NAC$

A concrete rule r_t with respect to an abstract rule r is given by $r_t = (L \xleftarrow{l} K \xrightarrow{r} R, t, NAC)$, where t is a triple of concrete typing clan morphisms $t = (t_L\colon L \to TG, t_K\colon K \to TG, t_R\colon R \to TG)$ such that the following conditions hold (cf. Fig. 2):

- $t_L \circ l = t_K = t_R \circ r$
- $t_L \leq type_L$, $t_K \leq type_K$, $t_R \leq type_R$, *and*
- $t_{R,N}(x) = type_{R,N}(x) \forall x \in R'_N$.

The set of all concrete rules r_t with respect to an abstract rule r is denoted by $\widehat{r}$.

$$\begin{tikzcd}
N & L \arrow[l, "n"'] \arrow[ddr, shift left, "type_L"] \arrow[ddr, shift right, "t_L"'] & K \arrow[l, "l"'] \arrow[r, "r"] \arrow[dd, shift left, "type_K"] \arrow[dd, shift right, "t_K"'] & R \arrow[ddl, shift left, "type_R"] \arrow[ddl, shift right, "t_R"'] \\
 & & & \\
 & & TG &
\arrow[from=1-1, to=3-3, "type_N", bend right=10] \arrow[from=1-1, to=3-3, "t_{N_i}"', bend right=15]
\end{tikzcd}$$

Fig. 2. Abstract and concrete rules.

The main idea for the application of an abstract rule is to apply one of its concrete rules. Both the host graph and the concrete rule are typed by concrete clan morphisms such that we can define the application of concrete rules. Later we will also define the application of an abstract rule and the equivalence of both (cf. Theorem 1).

Definition 7. *(Matching and Application of Concrete Rules)*
Let $r_t = (L \xleftarrow{l} K \xrightarrow{r} R, t, NAC)$ be a concrete rule, $(G, type_G)$ a typed graph, with $type_G \colon G \to TG$ being a concrete clan morphism, and $m \colon L \to G$ a graph morphism. m is a match *with respect to r_t and $(G, type_G)$, if*

- *m is a match with respect to the untyped rule $L \xleftarrow{l} K \xrightarrow{r} R$ and the graph G,*
- *$type_G \circ m = t_L$, and*
- *m satisfies the negative application conditions NAC, i.e. for each $(N, n, type_N) \in NAC$ it holds, that there does not exist a morphism $o \colon N \to G$, such that $o \circ n = m$ and $type_G \circ o \leq type_N$.*

Given a match m, the concrete rule can be applied to the typed graph $(G, type_G)$ via m. A direct derivation step is denoted by $(G, type_G) \overset{r_t, m}{\Longrightarrow} (H, type_H)$ and can be constructed similar to the classical theory of graph transformations [8].

In [2] we have shown that it is equivalent to apply concrete rules where typing is given by concrete clan morphisms or to apply classical rules with typing morphisms over a given type graph which is the concrete closure over a type graph with inheritance. Nevertheless, it makes sense to examine whether it is possible to find a more direct way to apply an abstract rule, because it is impractical for a tool implementing graph transformation with node type inheritance to hold all concrete rules of an abstract rule in memory or for each of them to find a match morphism into a host graph. Since abstract and concrete rules differ only in typing, but have the same structure, a match morphism from the LHS of the concrete rule into a given instance graph is also a match morphism for the abstract rule, for the latter one not being compatible with typing, though. Using the notion of type refinement, however, we can express a compatibility property.

Definition 8. *(Matching and Application of Abstract Rules)*
Let $r = (L \xleftarrow{l} K \xrightarrow{r} R, type, NAC)$ be an abstract rule typed over TG, $(G, type_G)$ a typed graph with $type_G \colon G \to TG$ being a concrete clan morphism, and $m \colon L \to G$ a graph morphism. Then m is a match with respect to r and $(G, type_G)$, if

- *m is a match with respect to the untyped rule $L \xleftarrow{l} K \xrightarrow{r} R$ and the graph G.*
- *$type_G \circ m \leq type_L$.*
- *$t_{K,N}(x_1) = t_{K,N}(x_2)$ for $t_K = type_G \circ m \circ l$ and all $x_1, x_2 \in K_N$ with $r_N(x_1) = r_N(x_2)$.*
- *m satisfies NAC, i.e. for each $nac = (N, n, type_N) \in NAC$ it holds that it does not exists a morphism $o \colon N \to G$ such that $o \circ n = m$ and $type_G \circ o \leq type_N$.*

Given a match m, the abstract rule can be applied to $(G, type_G)$ yielding an abstract direct derivation $(G, type_G) \overset{r,m}{\Longrightarrow} (H, type_H)$ with concrete type graph $(H, type_H)$ as follows:

1. *Construct the untyped direct derivation $G \overset{r,m}{\Longrightarrow} H$ in the sense of [9].*
2. *Construct $type_D$ and $type_H$ as follows*
   - *$type_D = type_G \circ l'$*
   - *$type_H(x) = \underline{if}\ x = r'(x')\ \underline{then}\ type_D(x')\ \underline{else}\ type_R(x'')$,*
     where $m'(x'') = x$ and $x \in H_E$ or $x \in H_N$

Theorem 1. *(Equivalence of Abstract and Concrete Direct Derivations)*
Given an abstract rule $r = (L \longleftarrow K \longrightarrow R, type, NAC)$ over a type graph TG with inheritance, a concrete typed graph $(G, type_G)$ and a structural match morphism $m \colon L \to G$ (i.e. a match with respect to the untyped rule $L \longleftarrow K \longrightarrow R$). Then the following statements are equivalent, where $(H, type_H)$ is the same concrete typed graph in both cases:

1. *$m \colon L \to G$ is a match with respect to the abstract rule r yielding an abstract direct derivation: $(G, type_G) \overset{r,m}{\Longrightarrow} (H, type_H)$.*
2. *$m \colon L \to G$ is a match with respect to the concrete rule $r_t = L \longleftarrow K \longrightarrow R$ with $r_t \in \widehat{r}$ and $t_L = type_G \circ m$ yielding a concrete direct derivation: $(G, type_G) \overset{r_t,m}{\Longrightarrow} (H, type_H)$.*

Theorem 1 allows us to use the dense form of abstract rules for model manipulation instead of generating and holding all concrete rules, i.e., abstract derivations are much more efficient than concrete derivations. In this sense, Theorem 1 allows us to use on the one hand an efficient procedure and on the other hand we are sure that the result is the same as in the classical theory using concrete rules. Moreover, as a consequence of Theorem 1, graph languages built over abstract rules and mechanisms are equivalent to graph languages that are built over a corresponding set of concrete rules. In general, rules together with a start graph define a graph grammar building up a graph language.

In the case of attribute labels, it might be convenient to add variable nodes of data types to rule graphs which are matched by concrete labels when applying such a rule. Please note that in the following figures for our example, the same variable might occur several times in a rule. It corresponds to one variable node which has to be matched by one data node. (Compare e.g. rule 2 in Fig. 3.)

Example: Generation of Statecharts. The graph grammar for generating valid Statechart instances according to the type graph with inheritance presented in Fig. 1 is shown in Fig. 3, where especially the type *StateVertex* (SV) is abstract. Please note that we omit the gluing graph K for illustrational reasons. The start graph contains a node of type

StateMachine (SM) connected to an *object* (OB). The UML specification establishes that a *StateMachine* has a unique *top* state of type *State*, but the UML well-formedness rules establish that its type should be further refined into a *CompositeState* (CS). For this purpose, rule 1 checks whether the *StateMachine SM* has already a *top* state and if this is not the case, it creates a *top* state together with a *CompositeState* (CS) and a *PseudoState* (PS) of the *initial* kind.

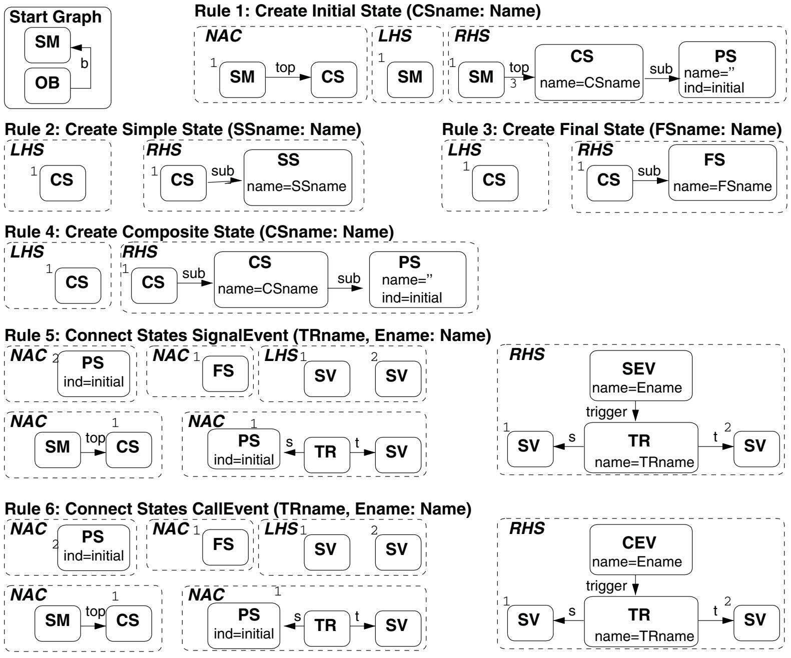
Start Graph
SM
OB
b
Rule 1: Create Initial State (CSname: Name)
NAC
LHS
RHS
top
sub
CS
name=CSname
PS
name=''
ind=initial
Rule 2: Create Simple State (SSname: Name)
SS
name=SSname
Rule 3: Create Final State (FSname: Name)
FS
name=FSname
Rule 4: Create Composite State (CSname: Name)
Rule 5: Connect States SignalEvent (TRname, Ename: Name)
SV
TR
s
t
SEV
name=Ename
trigger
name=TRname
Rule 6: Connect States CallEvent (TRname, Ename: Name)
CEV

Fig. 3. Graph grammar for generating valid Statecharts.

Rules 2, 3 and 4 create new *SimpleState* (SS), *FinalState* (FS) and *CompositeState* (CS) objects inside a given *CompositeState*. In contrast to rule 1 (where the multiplicity of relationship *top* in the side of the *State* class is "1..1") the multiplicity of the *subvertex* relationship (from *CompositeState* to *StateVertex* in the side of the latter class) is "0..*". This implies that there is no need for a negative application condition checking the multiplicity. Additionally, each *StateVertex* should be connected to at most one *CompositeState* through relationship *subvertex*. This is achieved by the graph grammar as each newly created state is attached to a single *CompositeState*, and this relationship cannot be modified later.

Finally, rules 5 and 6 allow connecting two objects of type *StateVertex* (SV). Rule 5 describes the insertion of a transition with *SignalEvent* (SEV), while rule 6 handles the case of *CallEvent* (CEV). They are *abstract* rules as *StateVertex* is an abstract class. Additionally, the UML specification establishes (by means of constraints expressed in OCL) that a *FinalState* should not have any outgoing connection, that an *PseudoState* of the *initial* kind should not have any incoming connection and at most one outgoing connection, and that the *top* state should not have any outgoing connection. We graphically modelled these constraints by means of negative application conditions (NACs). The advantages of using abstract rules here are clear, as otherwise we would have to model rules for the valid combinations of the states we want to connect. Additionally, the typing in NACs is more concrete than the corresponding typing in the LHS.

Fig. 4 shows a Statechart obtained through the derivation of the previous graph grammar. The concrete syntax of the final Statechart is shown in the lower right corner. In the third step in the derivation, abstract rule 5 is applied. Abstract types of nodes 1 and 2 in the rule instantiate to *PseudoState PS* and *SimpleState* (node called *'SS1'* in the graph), respectively. In the example, abstract rules 5 and 6 have been applied with other instantiations to connect nodes *'SS1'* (type *SS*) and *'CS2'* (type *CS*), *'CS2'* (type *CS*) and *'FS1'* (type *FS*), *'SS2'* (type *SS*) and *'SS1'* (type *SS*), as well as *PseudoState PS* and *'SS2'* (type *SS*). Without the possibility to model abstract rules, we would have had to create concrete rules for these combinations.

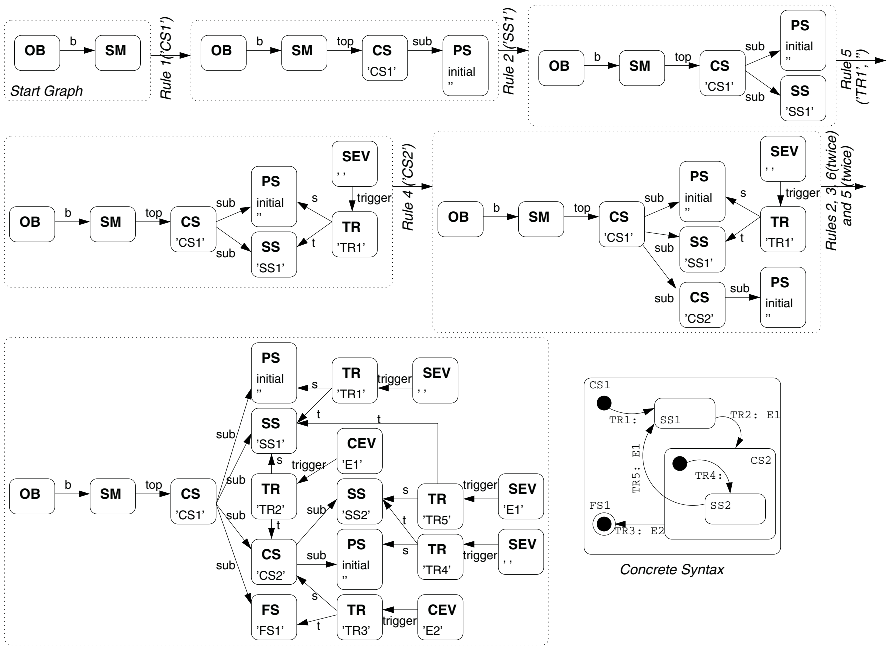

Fig. 4. A derivation of the graph grammar for generating Statecharts.

Example: Simulation of Statecharts. Fig. 5 shows a rule set for simulating our subset of Statecharts. The first rule adds the *current* relationship (*c*) to an object (OB) if it does not already have one. The initial state is the only *InitialState* node which is a *subvertex* (sub) of the *top* state. Rule 2 models a state change due to a transition from the current state. This is an abstract rule, as *StateVertex* nodes are abstract. This feature allows us to condense in a single abstract rule the combinations of all concrete subtypes of *StateVertex* nodes. Rule 3 is similar to the previous one, but models a state change into a composite state. In this case, the current state should be its initial state (that is, the *PseudoState* node is *subvertex* of the *CompositeState*). Rule 4 moves from the initial state to another one without considering events (one does not have to wait for an event to move from this *PseudoState*.) Finally, rule 5 models the fact that we can change the state due to transitions departing from any of the super-states of the *current* state. Thus, this rule allows going up in the *subvertex* hierarchy starting from the *current* state. We cannot apply this rule, if the *current* state is already a *subvertex* of the *top* state, or if the *current* state is indeed a *PseudoState* of the *initial* kind. The latter restriction is modelled by assigning type *State* (*ST*) to the *current* state in rule 5 (PseudoStates are not sub-classes of *State* but of *StateVertex*). The reason for forbidding this is that a transition in a PseudoState is still not finished, we have to end up in a node sub-class of *State*.

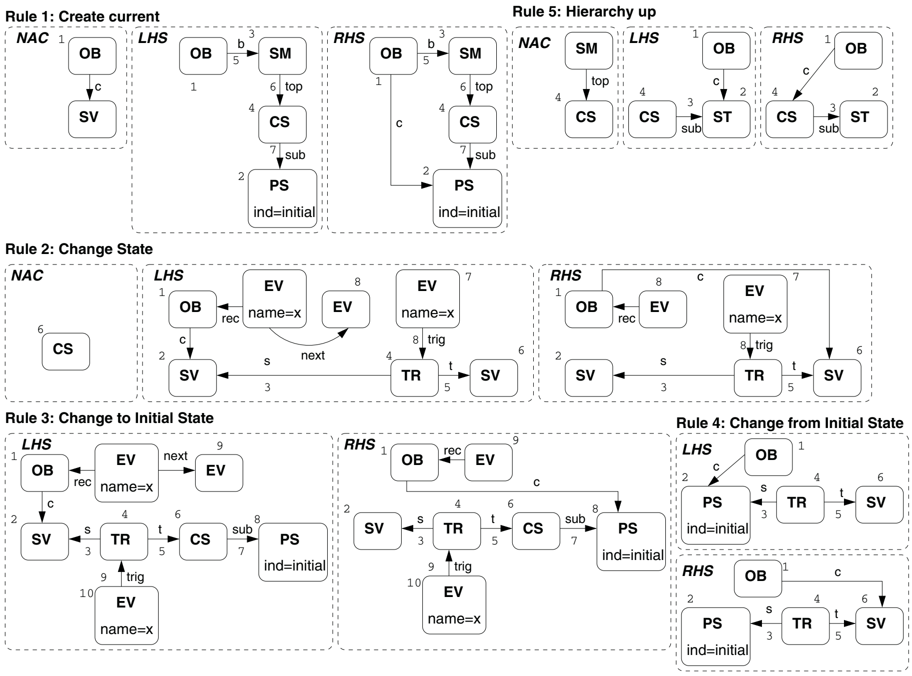

Fig. 5. Graph grammar for simulating Statecharts.

Fig. 6 shows an execution of the previous grammar to the Statechart we built in Fig. 4. In the first step, we apply rule 1, setting the *current* state pointer to the *PseudoState* (*initial* kind) of the *top* state. Then, abstract rule 4 moves the *current* state to node *'SS1'*. Node 6 in the rule (*StateVertex* type) is matched to node *'SS1'* in the graph, typed over *SimpleState*. Next, abstract rule 3 is applied and the pointer is moved to the initial state of composite state *'CS2'*. Node 2 (of type *StateVertex*) in the rule matches node *'SS1'* of type *SimpleState* in the graph; and the *Event* is of type *CallEvent*. Then, abstract rule 4 can be applied, which moves the pointer to node *'SS2'*. The type instantiation is from *StateVertex* in the rule to *SimpleState* in the graph. Now, abstract rule 5 is applied, moving the *current* pointer up in the hierarchy to node *'CS2'*. The type of node 2 (*CompositeState*) in the rule is instantiated to *SimpleState* of node *'SS2'* in the graph. For the following step, abstract rule 2 can be applied, and the pointer is set to node *'FS1'*. The type instantiation is from *StateVertex* and *Event* in the rules to *CompositeState*, *FinalState* and *CallEvent* in the graph. Here, no rule can be applied anymore, and the simulation finishes. Thus, this graph grammar models all possible simulations of the initial model. Some derivations may lead to dead ends. This may happen for example, going up in the hierarchy with rule 5, and finally discovering that none of the super-states have any outgoing transition.

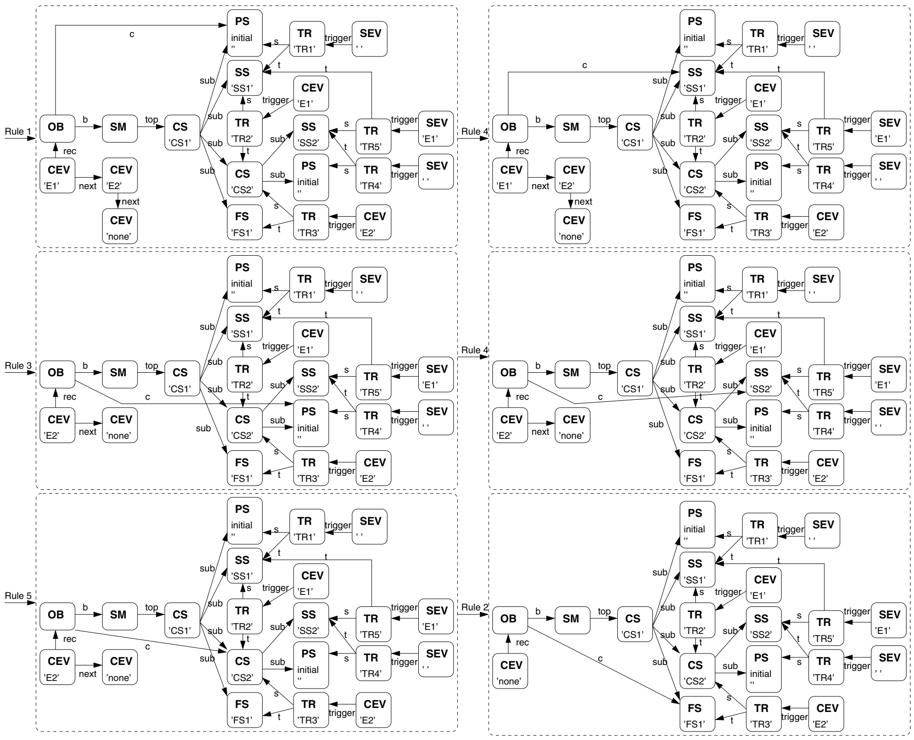

Fig. 6. A derivation of the simulation graph grammar starting from the graph generated in Fig. 4.

3 Integration of Meta-modelling with Graph Transformation

The extension of algebraic graph transformation with node type inheritance facilitates its integration with meta modelling. If we identify model element classes with node types and associations with edge types, a unique basis for the description of symbols and their relations is laid. Model elements can share common attributes and relations to other model elements which is expressed by a generalisation relationship. Similarly, an inheritance relation is supported for node types (see Sec.2). Summarising, the information expressed by class diagrams in the meta modelling approach is formulated by type graphs (with node type inheritance) for graph transformation. On top of this common basis, constraints are used to describe language properties in the meta modelling approach. On the other hand, typed graph grammars describe the modelling language as shown for the sample sub-language of Statecharts in Sec.2. In the following, two approaches for visual language (VL) definition and model manipulation are presented which distinguish in exactly this design decision. We first shortly present these approaches and compare them afterwards.

The GENGED Approach. In GENGED [1], a VL is defined (or generated) by an alphabet and a grammar. An alphabet establishes a type system for model elements (called *symbols*) and their relations (called *links*), i.e. it defines the vocabulary of a VL. The abstract syntax of symbols is represented by graph nodes, whereas graph edges represent the abstract syntax of links. The layout of symbols is given by graphical objects defining node attributes, and for each edge (abstract link) at least one graphical constraint is defined. An alphabet instance is given by an abstract syntax graph which is extended by graphical objects for the layout; the corresponding graphical constraints are solved accordingly. Usually, an abstract syntax graph is built up by VL rules (occurring in a VL grammar) which are modeled as graph rules. The grammar definition as well as the manipulation of models like Statecharts [3] is done purely by graph transformation as GENGED uses the graph transformation engine AGG [11] for this purpose.

Up to now, neither meta modelling nor inheritance concepts are realized. For defining all the features of Statecharts as we did by the type graph in Fig. 1, this type graph must be flattened in order to establish an alphabet. With the flatting, some more links have to be added. Moreover, the set of VL rules would correspond to concrete rules, i.e. the grammar contains many similar rules. Using node type inheritance concepts as proposed in Sec. 2 would reduce the set of rules in a sense that the proposed abstract rules have to be defined only. Such concise rule sets can be used to define concise abstract grammar rules for different purposes then, like syntax-directed editing, parsing, and simulation as it is supported by GENGED.

The AToM^3 Approach. AToM^3 [10] is a *multi-paradigm* modelling tool, which includes meta modelling, multi-formalism and modelling at different abstraction levels. Its main component is the *kernel*, responsible for loading, saving, creating and manipulating models, as well as for generating code for the meta modelled formalisms. The generated code must be loaded on top of the *kernel* again to allow the user building models in the defined formalism. The tool uses a pure meta modelling approach for VL definition, i.e. a VL is completely defined by a meta model, which is a type graph with inheritance with additional constraints. Some of them are assigned (pre- or post-conditions) to events (editing, connecting, etc.), the evaluation of which prohibits or enables the execution of the events and guarantees model correctness by construction.

In AToM^3, models can be manipulated by means of Python or with graph grammars. Typical manipulations are simulation, optimization and formalism transformation (which produces an instance model of a different meta model). When defining graph grammar rules, one may choose either an *"exact type matching"* or a *"sub-type matching"*. In the latter case, rules are considered *abstract* and any node can be matched with any of its sub-types. There is no distinction between *abstract* and *concrete* nodes and *sub-typing* relationships are found at runtime (by comparing nodes attributes and connections). This is due to the fact that some of the formalisms for meta modelling do not provide for inheritance. This feature also allows applying transformation rules to instances of meta models that are not explicitly related through inheritance relationships. In this way, the inheritance concept can be mapped to the semantics defined in this work, as AToM^3 can be configured to work in the *Double Pushout* approach.

Comparison of Both Approaches. After having defined the classes or types of model elements and their relations, AToM^3 supports the meta modelling approach which yields in a free editor where the model is checked according to given language constraints at specific events. Instead, GENGED can generate two kinds of editors: Either editing is done in a syntax-directed way where graph rules define the editor operations or free editing is supported where a parser has to check, if the edited model is syntactically correct. While the definition of a language by corresponding language constraints is usually easier, a parser is normally more efficient than a constraint checker. Syntax-directed editing assumes a language understanding which knows well about the structure and dependencies of its elements. Pure syntax-directed editors can be directly deduced from a language grammar. Combining both kinds of editing, the corresponding specification can be purely rule-based or mixed in the sense that rules define complex editing operations while language constraints define syntactic correctness.

Both approaches use graph transformation for model manipulations such as simulation. Due to the availability of node type inheritance, graph transformation concepts can build up directly on meta modelling concepts as in AToM^3. In GENGED, several kinds of graph transformation systems are used for different purposes as editing, parsing and simulation. Node type inheritance can condense each of them.

4 Related Work

Considering the node type inheritance concept for graph transformation, there are already tools like [21, 20] which support this concept in the same or nearly the same way. However, node type inheritance has been rarely considered in formal graph transformation approaches. The graph transformation-based language PROGRES is formalised by programmed structure rewriting systems [21] where so-called schema consistent structures are transformed. A schema corresponds to a type graph with node type inheritance, while a schema consistent structure corresponds to a well-typed instance graph. Thus, a formalisation of node type inheritance is available for PROGRES, but there is no theory building up on that. GME [16] e.g., is a meta modelling tool (for model integrated computing) which has lately incorporated graph transformation techniques for model manipulation, although its approach is not founded on the theory of graph transformation and its formalization has not been shown.

At the “Symposium on Visual Languages and Formal Methods” in 2001 there was a so-called “statechart modeling contest” where declarative as well as constructive methods have been used to define Statecharts and their behaviour. No winner was selected, but the specific strengths of the different methods have been discussed. There was not any approach integrating meta modelling with graph transformation, thus combining declarative with constructive methods. A number of graph transformation-based approaches were presented where most of the approaches could have been simplified using the hierarchy concept proposed in the present work. In addition, there is the work in [22] where Statecharts modelling is based on a meta model for extended hierarchical automata and graph transformation rules for its simulation. A similar approach is taken into account in [10] where a graph grammar is used to transform Statecharts to Petri nets which can be simulated, but there is no connection to formal graph transformation approaches.

The approach of [15] uses transformation units for generating and simulating statecharts, and is a clear example where our approach could have simplified the graph grammars. They encode the type hierarchy in graph grammar rules in such a way that they define rules for replacing each super-type for each one of its sub-types. Nonetheless, embedding conditions are needed for these rules and are not directly applicable in the standard *Double Pushout* approach.

5 Conclusions

In the literature, the main approaches to visual language definition are meta modelling and grammar-based approaches. We discussed how to integrate meta modelling with graph grammar concepts in order to support an efficient language definition and model manipulation. We presented two concrete approaches which differ in the way how meta modelling and graph transformation concepts are used and compared them.

The integration of meta modelling with graph transformation is based on a node type inheritance concept for algebraic graph transformation. This concept allows the definition of abstract rules, in which abstract nodes can appear. These can be matched with nodes of any of its sub-types. The concept is extremely useful in practice as graph grammars can be notably simplified. This has been demonstrated by showing a generation and a simulation grammar for a sub-set of UML Statecharts. The formalism presented is restricted to attributes being labels. It is up to future work to extend this work to attributed graph transformation where computations on attributes can take place and also edges may be attributed.

Moreover, analysis techniques available for attributed graph transformation such as constraint checking [14, 17] and critical pair analysis [13], should be lifted to graph transformation with node type inheritance. Having e.g. constraint checking available, language requirements could be expressed by syntactic consistency constraints in the meta modelling approach first. If parsing rules are developed thereafter, their correctness with respect to requirements could be checked. In this way we ensure that the language defined by the parser is at least a sub-language of that defined by constraints. Critical pair analysis can be useful to optimise the visual language parser (see [7]).

References

1. Bardohl, R., 2002 *A Visual Environment for Visual Languages* Science of Computer Programming 44, pages 181-203. The GenGED home page: http://tfs.cs.tu-berlin.de/genged
2. Bardohl,R., Ehrig, H., de Lara, J., Runge, O., Taentzer, G., Weinhold, I. 2003. *Node Type Inheritance Concept for Typed Graph Transformation* Technical Report 2003–19, TU Berlin.
3. Bardohl, R., and Ermel, C. 2001. *Visual Specification and Parsing of a Statechart Variant using* GenGED. In Statechart Modeling Contest, part of VLFM 2001.

4. Bardohl, R., Taentzer, G., Minas, M., Schürr, A. 1999. *Application of Graph Transformation to Visual Languages*. In Handbook of Graph Grammars and Computing by Graph Transformation, Volume 2, H.Ehrig, G.Engels, H.-J.Kreowski, and G.Rozenberg (eds.), pages 105–181. World Scientific.
5. Baresi, L., Pezze, M. 2002. *A Toolbox for Automating Visual Software Engineering*. In FASE 2002, R. Kutsche and H. Weber (eds.), pages 189 – 202. Springer LNCS 2306.
6. Bottoni, P., Koch, M., Parisi-Presicce, F., Taentzer, G. 2001. *A Visualization of OCL using Collaborations*. In UML 2001, M.Gogolla and C.Kobryn (eds.), Springer LNCS 2185.
7. Bottoni, P., Schürr, A., Taentzer, G. 2000. *Efficient Parsing of Visual Languages based on Critical Pair Analysis and Contextual Layered Graph Transformation*. Technical Report no. si-2000-06, University of Rome.
8. Corradini, A., Montanari, H., Rossi, F. 1996. *Graph Processes*. Special Issue of Fundamenta Informaticae, Vol 26(3-4), pages 241–266.
9. Corradini, A., Montanari, U., Rossi, F., Ehrig, H., Heckel, R., Löwe, M. 1997 *Algebraic Approaches to Graph Transformation I: Basic Concepts and Double Pushout Approach*. In Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1, G.Rozenberg (ed.), pages 163–245. World Scientific.
10. de Lara, J., Vangheluwe, H., Alfonseca, M. 2003. *Meta-Modelling and Graph Grammars for Multi-Paradigm Modelling in AToM^3*. To appear in Software and Systems Modelling. Springer. See also the AToM^3 home page at: http://atom3.cs.mcgill.ca
11. Ermel, C., Rudolf, M., Taentzer, G. 1999 *The AGG Approach: Language and Tool Environment*. In Handbook of Graph Grammars and Computing by Graph Transformation, Volume 2, H.Ehrig, G.Engels, H.-J.Kreowski, and G.Rozenberg (eds.), pages 551 – 603. World Scientific. See also the AGG Home Page: http://tfs.cs.tu-berlin.de/agg
12. Harel, D. 1987. *Statecharts: A Visual Formalism for Complex Systems*. Science of Computer Programming, 8:231-274.
13. Heckel, R., Küster, J., Taentzer, G. 2002. *Towards Automatic Translation of UML Models into Semantic Domains*. In Proc. AGT 2002, H.-J. Kreowski (ed.), pages 11 – 22.
14. Heckel, R., Wagner, A., 1995. *Ensuring Consistency of Conditional Graph Grammars – A constructive Approach*. In ENTCS no. 2, Elsevier.
15. Kuske, S., 2001. *A Formal Semantics of UML State Machines Based on Structured Graph Transformation*. In UML 2001, M.Gogolla and C.Kobryn (eds.), Springer LNCS 2185.
16. Lédczi, A., Bakay, A., Marói, M., Vögyesi, P., Nordstrom, G., Sprinkle, J., Karsai, G. 2001. *Composing Domain-Specific Design Environments*. IEEE Computer, pages 44-51. See also the GME home page: http://www.isis.vanderbilt.edu/Projects/gme/default.html
17. Matz, M., 2002. *Konzeption und Implementierung eines Konsistenznachweisverfahrens für attributierte Graphtransformation*. Master's thesis, TU Berlin, Fak. IV.
18. Marriot, K., Meyer, B. 1998. *Visual Language Theory*. Springer.
19. MDA, MOF and UML specifications at the OMG web page: http://www.omg.org/
20. Nickel, U., Niere, J., Zündorf, A. 2000. *The Fujaba Environment*. In ICSE 2000, pages 742–745. See also the Fujaba Home Page: http://www.fujaba.de/
21. Schürr, A. 1996. *Programmed Graph Replacement Systems*. In Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1, G.Rozenberg (ed.), pages 479–546. World Scientific. See also the PROGRES home page: http://www-i3.informatik.rwth-aachen.de/research/projects/progres/
22. Varro, D. 2002. *A Formal Semantics of UML Statecharts by Model Transition Systems*. In ICGT 2002, A. Corradini, H. Ehrig, H.-J. Kreowski, and G. Rozenberg (eds.), pages 378–392, Springer LNCS 2505.
23. Warmer, J. B., Kleppe, A. 1998. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley Object Technology Services.