Available online at www.sciencedirect.com
ScienceDirect
Theoretical Computer Science 376 (2007) 139–163
Theoretical Computer Science
www.elsevier.com/locate/tcs

Attributed graph transformation with node type inheritance

Juan de Lara[a,*], Roswitha Bardohl[b], Hartmut Ehrig[c], Karsten Ehrig[d], Ulrike Prange[c], Gabriele Taentzer[c]

[a] *Escuela Politécnica Superior, Ingeniería Informática, Universidad Autónoma de Madrid, Madrid, Spain*
[b] *International Conference and Research Centre for Computer Science, Schloss Dagstuhl, Dagstuhl, Germany*
[c] *Computer Science Department, Technische Universitat Berlin, Berlin, Germany*
[d] *Computer Science Department, University of Leicester, Leicester, United Kingdom*

Abstract

The aim of this paper is to integrate typed attributed graph transformation with node type inheritance. Borrowing concepts from object oriented systems, the main idea is to enrich the attributed type graph with an inheritance relation and a set of abstract nodes. In this way, a node type inherits the attributes and edges of all its ancestors. Based on these concepts, it is possible to define *abstract* productions, containing abstract nodes. These productions are equivalent to a number of concrete productions, resulting from the substitution of the abstract node types by the node types in their inheritance clan. Therefore, productions become more compact and suitable for their use in combination with meta-modelling. The main results of this paper show that attributed graph transformation with node type inheritance is fully compatible with the existing concept of typed attributed graph transformation.

Keywords: Graph transformation; Meta-modelling; Double pushout approach; Visual languages

1. Introduction

Graphs and visual notations play a central role in modelling and meta-modelling of software systems. Examples range from simply formatted, graph-like notations such as class diagrams, Petri nets, activity diagrams, etc. to more elaborated diagram kinds such as message sequence charts and state charts as well as to more domain-specific notations for modelling, e.g. for industrial production processes.

In graph-based modelling and meta-modelling, graphs are used to define the static structure, such as class and object structures, data base schemes, as well as visual symbols and interrelations, i.e. visual alphabets and sentences. Graph manipulations describe the dynamic changes of these structures. Graph transformation [14] is a formal, graphical and natural means to express graph manipulation based on rules. There are many areas in software engineering where graph transformation has been applied. Considering especially meta-modelling, there is work done e.g. to define visual

* Corresponding author.
E-mail addresses: jdelara@uam.es (J. de Lara), rosi@dagstuhl.de (R. Bardohl), ehrig@cs.tu-berlin.de (H. Ehrig), karsten@mcs.le.ac.uk (K. Ehrig), uprange@cs.tu-berlin.de (U. Prange), gabi@cs.tu-berlin.de (G. Taentzer).

0304-3975/$ - see front matter
doi:10.1016/j.tcs.2007.02.001

languages [3–5,7,24], visual simulation [21,34], model transformation [11,29,32] and refactoring [25]. The rich theory developed in the last 30 years [14] allows the analysis of the computations expressed as graph transformations.

For the description of model domains, a classification of possible entities and relations in system structures or visual alphabets has emerged as a valuable principle. In the object-oriented approach [6], *class diagrams* are the basic means to specify classification structures, for instance in UML (Unified Modeling Language) [26] for software systems and MOF (Meta Object Facility) [26] for visual language specification. When applying graph transformation for modelling or meta-modelling, *type graphs* are used to classify graph nodes and edges [8].

One of the main principles to handle complex classification structures comes from the object-orientation paradigm: class inheritance enhances the typing principle by adding more abstract types on top of the ones concretely used in the (meta)models. Thus, inheritance allows much more compact representations by reducing redundancy. Moreover, it can improve flexibility, reusability and extensibility of the specified systems. Although there are slightly different semantic interpretations of inheritance depending on the approach, the main idea is that in object-oriented systems, the source element of an inheritance relation, receives features of all the reachable elements through the inheritance relation. Usually, the inherited features are attributes and relations. For example, in a UML class diagram [26], classes inherit attributes, methods and associations of all their ancestor classes. Classes may be *abstract* which means that they cannot be instantiated at run-time. In UML, it is also possible to define object diagrams, which are run-time system configurations being consistent with the defined class diagram. An object in an object diagram has actual values for the attributes, and contains all the relations and attributes defined in its corresponding class, plus the inherited ones.

We have carried over the principle of inheritance to typed attributed graph transformation by extending the type graph with an inheritance relation and a set of abstract node types. Thus, in analogy with object diagrams being consistent with a class diagram, we have attributed graphs typed with respect to an attributed type graph with inheritance. Moreover, we allow graph grammar productions to contain nodes whose (maybe abstract) type is the target of some inheritance relations. These productions are equivalent to a number of concrete productions, resulting from the substitution of this kind of node types by the concrete ones in their inheritance clan. Thus, productions can become more compact and lead to a denser form of a graph grammar or graph transformation system.

The incorporation of the inheritance concept in graph transformation is especially relevant in approaches where graph transformation is combined with meta-modelling concepts, e.g. for visual language definition, simulation and model transformation. Of course, it can also be advantageously used for object-oriented modelling as done e.g. with Fujaba [16].

This paper is an extended version of [2], where we presented the inheritance concept for graph transformation without attributes. We have incorporated further results [10] concerning attribution and show some of the relevant proofs. Moreover, the handling of the negative application conditions has been improved with respect to [10]. In the present paper we present all main concepts for attributed graph transformation with node type inheritance and show how this kind of graph transformation can be translated back to attributed graph transformation without node type inheritance, i.e. typed, attributed graph transformation, which comes along with a comprehensive theory [9]. Having the translation to this kind of graph transformation available, its theory can be used also for graph transformation with inheritance.

The rest of the paper is organized as follows. Section 2 presents a short overview of typed graph transformation, in the Double Pushout (DPO) algebraic approach. Section 3 extends the supporting structure for graphs to consider node and edge attributes. Section 4 shows our approach to consider inheritance in type graphs. In Section 5 we use the inheritance concept in productions by allowing abstract nodes. Section 6 shows the equivalence of abstract and flattened productions. Section 7 presents a case study, with the simulation of Statecharts. Finally, Section 8 ends with the conclusions and prospects for future work. An Appendix shows the details of the proofs of the main theorems. The full proofs of all theorems and lemmas can be found in [20].

2. Introduction to typed graph transformation

This section gives an overview of typed graph transformation (without attributes and inheritance) in the Double Pushout approach [14]. We start defining some basic concepts about graphs and types; then we show how graph transformation works.

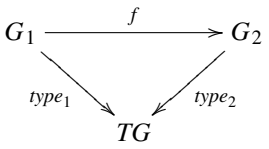


Fig. 1. A typed graph morphism.

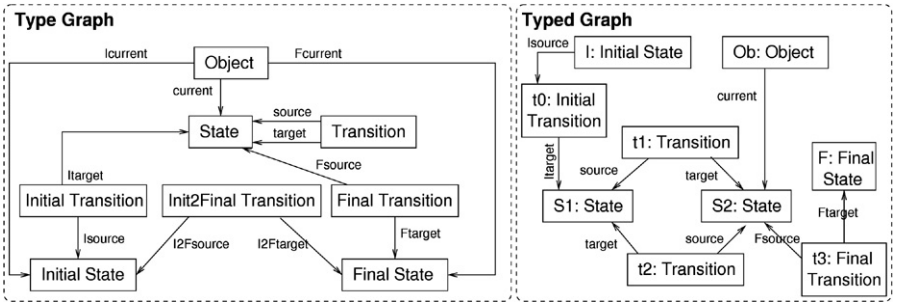


Fig. 2. Example type graph (left). Typed graph (right).

2.1. Graphs and typed graphs

Definition 1 (*Graph*). A *graph* $G = (V, E, s, t)$ consists of a set V of vertices (also called nodes), a set E of edges and the source and target functions $s, t : E \to V$.

Graphs are related by (total) graph morphisms, mapping the nodes and edges of a graph to those of another one, preserving source and target of each edge. Graphs together with graph morphisms form the category **Graph**.

Definition 2 (*Graph Morphism*). Given two graphs $G_i = (V_i, E_i, s_i, t_i)_{i\in\{1,2\}}$, a *graph morphism* $f : G_1 \to G_2$, $f = (f_V, f_E)$ consists of two functions $f_V : V_1 \to V_2$ and $f_E : E_1 \to E_2$ that preserve the source and target functions, i.e. $f_V \circ s_1 = s_2 \circ f_E$ and $f_V \circ t_1 = t_2 \circ f_E$.

As in programming languages, we can assign each element of the graph a type [8]. This can be done by defining a type graph *TG*, which is a distinguished graph containing all the relevant types and their interrelations. The typing itself is depicted by a graph morphism between the graph and the type graph *TG*. Therefore, a tuple $(G, type)$ of a graph G and a graph morphism $type : G \to TG$ is called a *typed graph*.

Given typed graphs $G_i^T = (G_i, type_i)_{i\in\{1,2\}}$, a *typed graph morphism* $f : G_1^T \to G_2^T$ is a graph morphism $f : G_1 \to G_2$ such that $type_2 \circ f = type_1$, as Fig. 1 shows:

Given a type graph *TG*, $\mathbf{Graph_{TG}}$ is the slice category **Graph** \ **TG**, where the category objects and morphisms are the typing morphisms and the typed morphisms.

Fig. 2 shows an example of a typed graph (right) typed over the type graph to its left. In the typed graph, we have depicted the node types inside the nodes in a UML-like notation. For the edges, only their types but no names are given. The type graph example specifies systems made of objects that have a behaviour described by an automaton. The current state of each object is pointed to by the *current* edge. There are three kinds of states: initial, final and regular. There can be transitions between any of them (except transitions whose target is an initial state of whose origin is a final state). However, due to the fact that there are three different kinds of states, we need different kinds of transitions and of *current* edges. This situation will be improved with the inheritance concept to be presented later.

The type graph *TG* defines a set of *valid* graphs, namely those that are typed over *TG*. However, sometimes we need to constrain more this set. For example, we may need to express the fact that each object has a unique initial state and one or more final states. This can be done in several ways. One of them is by means of a *syntax grammar*, which generates the set of all valid models by means of graph transformation.

2.2. Typed graph transformation

Conceptually, a graph transformation production is made of a left-hand side (LHS) and a right-hand side (RHS). Roughly, when a production is applied to a graph G (called *host graph*), a valid matching morphism m has to be found

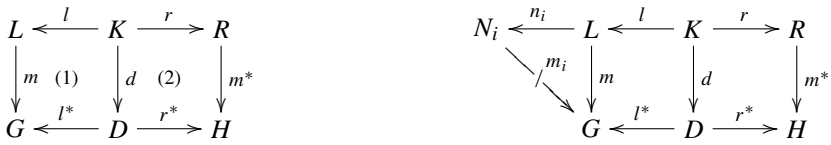


Fig. 3. Direct graph transformation in DPO (left). Direct graph transformation in DPO with negative application condition (right).

between the LHS and G. Then, the image of the LHS in G is substituted by the RHS. A graph grammar consists of a set of productions and a starting graph. The corresponding graph grammar language is made of all possible graphs that can be derived from the starting graph in any number of steps. At each transformation step, any applicable production of the grammar can be executed.

One of the formalizations of graph transformation (the one we use in this paper) is called Double Pushout (DPO) and is based on pushouts in category theory [14]. In the DPO approach, productions are represented by three graphs and two morphisms as shown in the next definition.

Definition 3 (*Graph Production*). A *(typed) graph production* $p = L \xleftarrow{l} K \xrightarrow{r} R$ consists of (typed) graphs L, K and R, called left-hand side, glueing graph and right-hand side respectively, and two injective (typed) graph morphisms l and r.

In a production, K contains the preserved elements by the production application. In most examples, l and r are not only injective but inclusions, and therefore $K = L \cap R$. The application of a production to a graph can be modelled through two pushouts (a categorical construction, which, in the case of graphs is the union of two graphs through a common subgraph). The first one eliminates the elements in $L - K$, the second one adds the elements in $R - K$, as the left of Fig. 3 shows. In fact, in the first step, the pushout complement has to be calculated, yielding graph D. A necessary and sufficient condition for the existence of the pushout complement is the well-known gluing condition [14].

Definition 4 (*Graph Transformation*). Given a (typed) graph production $p = L \xleftarrow{l} K \xrightarrow{r} R$ and a (typed) graph G with a (typed) graph morphism $m : L \to G$, called match. A *direct (typed) graph transformation* $G \overset{p,m}{\Longrightarrow} H$ from G to a (typed) graph H is given by the diagram to the left of Fig. 3, where (1) and (2) are pushouts.

A sequence $G_0 \Rightarrow G_1 \Rightarrow \cdots \Rightarrow G_n$ of direct (typed) graph transformations is called a *(typed) graph transformation* and is denoted as $G_0 \overset{*}{\Rightarrow} G_n$. For $n = 0$ we have the identical (typed) graph transformation $G_0 \overset{id}{\Rightarrow} G_0$. Moreover, we allow for $n = 0$ also graph isomorphisms $G_0 \overset{\sim}{=} G'_0$, because pushouts and hence also direct graph transformations are only unique up to isomorphism.

Fig. 4 shows a direct transformation example. The upper part depicts a production typed over the type graph of Fig. 2. The production models an object that changes its current state through a transition. The production is applied to the same typed graph of Fig. 2. Morphisms are depicted with numbers. As the glueing graph K in the production can be deduced given L and R, we usually ommit it in the following.

Productions can be equipped with a set of additional application conditions, the simpler form of them are *negative application conditions* (NACs). These are modelled as additional graphs (N_i to the right of Fig. 3) and morphisms n_i from L to N_i. In order for the production to be applicable, no injective morphism m_i should exist between any N_i and the host graph G such that $m_i \circ n_i = m$. Please note that a NAC is a special case of the more general concept of application condition [13].

Finally, we define graph transformation systems, grammars and languages.

Definition 5 (*GT System, Graph Grammar and Language*). A *graph transformation system* $GTS = (P)$ consists of a set of graph productions P. A *typed graph transformation system* $GTS = (TG, P)$ consists of a type graph TG and a set of typed graph productions P. A *(typed) graph grammar* $GG = (GTS, S)$ consists of a (typed) graph transformation system GTS and a (typed) start graph S. The *(typed) graph language* L of GG is defined by:

$$L = \{G \mid \exists \text{ (typed) graph transformation } S \overset{*}{\Rightarrow} G\}.$$

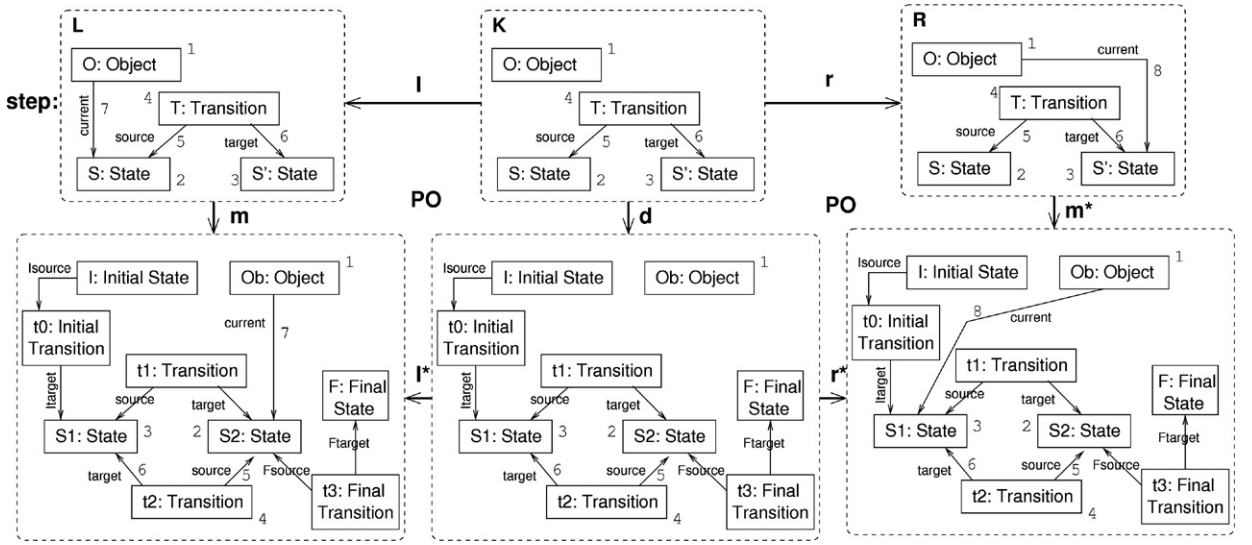


Fig. 4. Direct graph transformation example.

For practical applications, the previous concept of typed graph has to be extended in two ways. In software engineering applications, graphs represent data structures where nodes are associated with attributes. The type of these attributes is defined in the type graph, while at the instance graph level, attributes are assigned values of the proper type. As stated in the introduction, the concept of *inheritance* is quite common in most modelling notations (such as UML) and in object-oriented systems. Inheritance is a special kind of transitive relation, that reflects the fact that a child node (the source of an inheritance relation) receives all the features of the parent node (the target of the relation). The features the child node inherits are the attributes and the associations. Using the concept of inheritance is very useful in large applications as a means to structure the system, reducing its complexity by eliminating redundancy, and improving flexibility and extensibility. In the next sections we formally define a framework which extends the presented typed graph transformation concepts with these two features.

3. Attributed type graphs

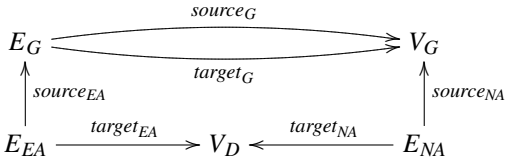
In this section, we provide nodes and edges in graphs with attributes. We follow the approach of [12] by defining a new kind of graph, called E-graph. This kind of graph allows attribution for both nodes and edges. This new kind of attributed graphs combined with the concept of typing leads to a category $\mathbf{AGraphs_{ATG}}$ of attributed graphs typed over an attributed type graph *ATG*.

Definition 6 (*E-graph and E-graph Morphism*). An *E-graph* G with $G = (V_G, V_D, E_G, E_{NA}, E_{EA}, (source_j, target_j)_{j\in\{G,NA,EA\}})$ consists of sets:

- V_G and V_D called graph and data nodes (or vertices) respectively;
- E_G, E_{NA}, E_{EA} called graph, node attribute and edge attribute edges respectively.

and source and target functions:

- $source_G : E_G \to V_G$, $target_G : E_G \to V_G$ for graph edges;
- $source_{NA} : E_{NA} \to V_G$, $target_{NA} : E_{NA} \to V_D$ for node attribute edges;
- $source_{EA} : E_{EA} \to E_G$, $target_{EA} : E_{EA} \to V_D$ for edge attribute edges.



Let $G^k = (V_G^k, V_D^k, E_G^k, E_{NA}^k, E_{EA}^k, (source_j^k, target_j^k)_{j\in\{G,NA,EA\}})$ for $k = 1, 2$ be two E-graphs. An *E-graph morphism* $f : G^1 \to G^2$ is a tuple $(f_{V_G}, f_{V_D}, f_{E_G}, f_{E_{NA}}, f_{E_{EA}})$ with $f_{V_i} : V_i^1 \to V_i^2$ and $f_{E_j} : E_j^1 \to E_j^2$ for $i \in \{G, D\}$, $j \in \{G, NA, EA\}$ such that f commutes with all source and target functions, e.g. $f_{V_G} \circ source_G^1 = source_G^2 \circ f_{E_G}$.

The sets E_{NA} and E_{EA} are needed as we want to allow nodes and edges to have several attributes. On the contrary, having directly a function from V_G or E_G to V_D would not allow this. Moreover, attribute edges are needed to replace attribute values during a graph transformation. Simple functions would not allow this either. E-graphs and E-graph morphisms form the category **EGraphs**. An attributed graph is an E-graph combined with an algebra over a data signature *DSIG*, in the sense of algebraic signatures (see [15]). In the signature, we distinguish a set of attribute value sorts. The corresponding carrier sets in the algebra are used for the attribution.

Definition 7 (*Attributed Graph and Attributed Graph Morphism*). Let $DSIG = (S_D, OP_D)$ be a data signature with attribute value sorts $S'_D \subseteq S_D$. An *attributed graph* $AG = (G, D)$ consists of an E-graph G together with a *DSIG*-algebra D such that $\uplus_{s\in S'_D} D_s = V_D$.
For two attributed graphs $AG^i = (G^i, D^i)$ with $i = 1, 2$, an *attributed graph morphism* $f : AG^1 \to AG^2$ is a pair $f = (f_G, f_D)$ with an E-graph morphism $f_G : G^1 \to G^2$ and an algebra homomorphism $f_D : D^1 \to D^2$ such that (1) commutes for all $s \in S'_D$.

$$\begin{array}{ccc} D_s^1 & \xrightarrow{f_{D,s}} & D_s^2 \\ \downarrow & (1) & \downarrow \\ V_D^1 & \xrightarrow{f_{G,V_D}} & V_D^2 \end{array}$$

Given a data signature *DSIG*, attributed graphs and morphisms form the category **AGraphs**. For the typing of attributed graphs, we use a distinguished graph, which is attributed over the final *DSIG*-algebra Z, with $Z_s = \{s\}$ $\forall s \in S_D$.

Definition 8 (*Typed Attributed Graph and Morphism*). Given a data signature *DSIG*, an *attributed type graph* is an attributed graph $ATG = (TG, Z)$, where Z is the final *DSIG*-algebra.
A *typed attributed graph* (AG, t) over *ATG* consists of an attributed graph *AG* together with an attributed graph morphism $t : AG \to ATG$.
A *typed attributed graph morphism* $f : (AG^1, t^1) \to (AG^2, t^2)$ is an attributed graph morphism $f : AG^1 \to AG^2$ such that $t^2 \circ f = t^1$.

Typed attributed graphs over an attributed type graph *ATG* and typed attributed graph morphisms form the category $\mathbf{AGraphs_{ATG}}$.

As an example, we have extended the type graph in Fig. 2 with some attributes. The resulting type graph is shown to the left of Fig. 5 using an explicit notation for node and edge attributes. We have provided objects, states and transitions with names. In addition, transitions are also provided with the name of the event that produces a transition change and objects may receive events through the *rec* relation. The edge named *current* has been provided with an attribute that counts the number of state changes that the object has performed. Note also that the data node *String* has been included twice for better readability. In the centre, the figure shows a compact notation (UML-like) for the same type graph, where the attributes are depicted in an additional box with the node name, or below the edge type. Finally, in the right part of the figure, we show an attributed graph typed over the previous type graph.

The fact of using sets of special edges for node and edge attributes (E_{EA} and E_{NA}) implies that a typed graph may have nodes with an arbitrary number of attributes of a certain type (that is, the typing morphism identifies all of them with a certain attribute in the type graph), including zero. Although this allows more flexibility for practical applications, the multiplicity of the attribution edges can be restricted to one by means of constraints [13]. Moreover, the fact of having a set of attribution edges implies that each element is unique. Although this can be interpreted as the fact that it is not possible to have attributes with the same name in the type graph, in practice, it is possible to solve this restriction by naming conventions or considering edges as triples (see Definition 10 and Fig. 7).

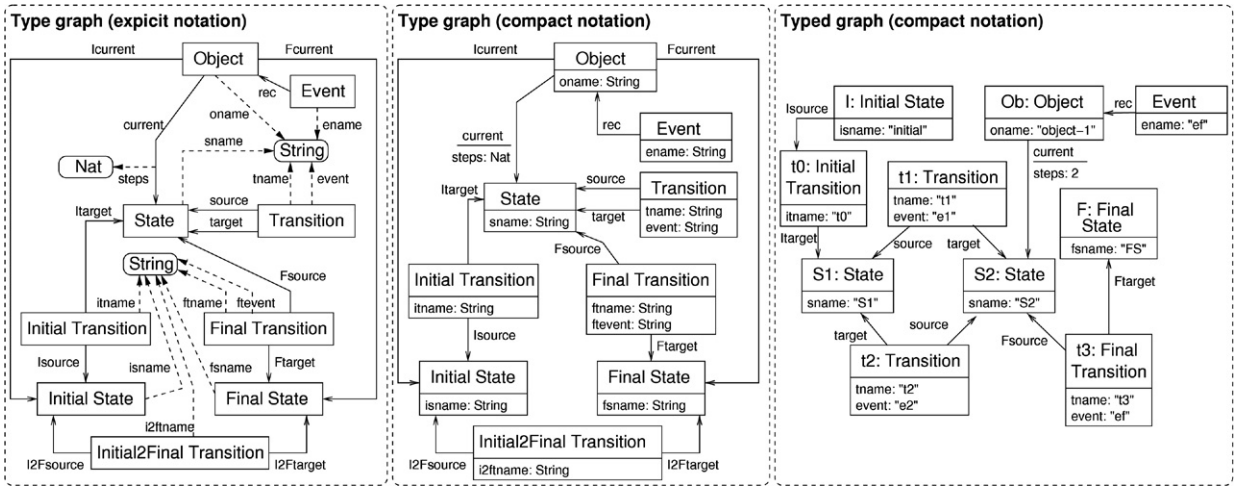


Fig. 5. Attributed type graph, explicit notation (Left). Attributed type graph, compact notation (Centre). Typed attributed graph, compact notation (Right).

The next section extends the concepts presented so far by adding inheritance to the type graphs. This feature will solve some of the problems of the example (repetition of the *name* attribute, different types of transitions, and different types of *current* edge).

4. Attributed type graphs with inheritance

An attributed type graph with inheritance is an attributed type graph in the sense of Definition 8 with a distinguished set of abstract nodes and inheritance relations between the nodes. The inheritance clan of a node represents all its subnodes. The notion of typed graph morphism has to be extended to capture the inheritance clan. Thus, we introduce clan morphisms. For this new kind of objects and morphisms, basic properties are shown. The proof for the main result in this section is given in the Appendix.

Definition 9 (*Attributed Type Graph with Inheritance*). An *attributed type graph with inheritance* $ATGI = (TG, Z, I, A)$ consists of an attributed type graph $ATG = (TG, Z)$ (see Definition 8), where TG is an $E-graph$ $TG = (TG_{V_G}, TG_{V_D}, TG_{E_G}, TG_{E_{NA}}, TG_{E_{EA}}, (source_i, target_i)_{i\in\{G,NA,EA\}})$ with $TG_{V_D} = S'_D$ and Z the final *DSIG*-algebra, and an inheritance graph $I = (I_V, I_E, s, t)$, with $I_V = TG_{V_G}$, and a set $A \subseteq I_V$, called abstract nodes.

For each node $n \in I_V$ the *inheritance clan* is defined by

$$clan_I(n) = \{n' \in I_V \mid \exists \text{ path } n' \xrightarrow{*} n \text{ in } I\} \subseteq I_V \text{ with } n \in clan_I(n).$$

Remark. $x \in clan_I(y)$ implies $clan_I(x) \subseteq clan_I(y)$.

The inheritance graph I could be defined to be acyclic, but this is not necessary for our theory. If n is concrete (i.e. not abstract), we could define all $x \in clan_I(n)$ to be concrete, but again this is not necessary from the theoretical point of view.

Fig. 6 extends the previous examples by adding inheritance to the type graph. In the picture, we have merged graphs TG and I into a single one, where the edges of the latter are depicted with hollow arrows. There is a unique abstract node (*NamedElement*), which is shown in italics (as in the usual UML notation). By adding inheritance we are able to simplify notably the set of edges in the previous type graphs. Please note also that, as there is a unique *current* edge, this contains the *steps* attribute. This is a difference with the type graph in Fig. 5, where *Icurrent* and *Fcurrent* edges did not have such attribute.

In order to benefit from the well-founded theory of typed attributed graph transformation, we flatten attributed type graphs with inheritance to ordinary ones. We define the closure of an attributed type graph with inheritance, leading to an (explicit) attributed type graph, which allows us to define instances of attributed type graphs with inheritance.

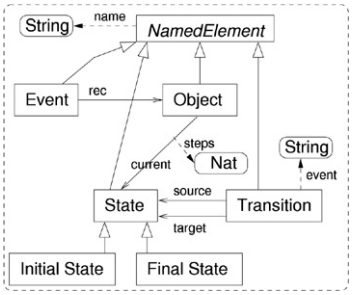


Fig. 6. A type graph with inheritance.

Definition 10 (*Closure of Attributed Type Graphs with Inheritance*). Given an attributed type graph with inheritance $ATGI = (TG, Z, I, A)$ with $ATG = (TG, Z)$ as above, the abstract closure of $ATGI$ is the attributed type graph $\overline{ATG} = (\overline{TG}, Z)$ with $\overline{TG} = (TG_{V_G}, TG_{V_D}, \overline{TG_{E_G}}, \overline{TG_{E_{NA}}}, \overline{TG_{E_{EA}}}, (\overline{source_i}, \overline{target_i})_{i \in \{G,NA,EA\}})$

- $\overline{TG_{E_G}} = \{(n_1, e, n_2) \mid n_1 \in clan_I(source_G(e)), n_2 \in clan_I(target_G(e)), e \in TG_{E_G}\}$
- $\overline{source_G}((n_1, e, n_2)) = n_1 \in TG_{V_G}$
- $\overline{target_G}((n_1, e, n_2)) = n_2 \in TG_{V_G}$
- $\overline{TG_{E_{NA}}} = \{(n_1, e, n_2) \mid n_1 \in clan_I(source_{NA}(e)), n_2 = target_{NA}(e), e \in TG_{E_{NA}}\}$
- $\overline{source_{NA}}((n_1, e, n_2)) = n_1 \in TG_{V_G}$
- $\overline{target_{NA}}((n_1, e, n_2)) = n_2 \in TG_{V_D}$
- $\overline{TG_{E_{EA}}} = \{((n_{11}, e_1, n_{12}), e, n_2) \mid e_1 = source_{EA}(e) \in TG_{E_G}, n_{11} \in clan_I(source_G(e_1)), n_{12} \in clan_I(target_G(e_1)), n_2 = target_{EA}(e) \in TG_{V_D}, e \in TG_{E_{EA}}\}$
- $\overline{source_{EA}}((n_{11}, e_1, n_{12}), e, n_2) = (n_{11}, e_1, n_{12}) \in \overline{TG_{EG}}$
- $\overline{target_{EA}}((n_{11}, e_1, n_{12}), e, n_2) = n_2 \in TG_{V_D}$

The attributed type graph $\widehat{ATG} = (\widehat{TG}, Z)$ with $\widehat{TG} = \overline{TG}|_{TG_{V_G} \setminus A} \subseteq \overline{TG}$ is called the concrete closure of $ATGI$, because all abstract nodes are removed. $\widehat{TG} = \overline{TG}|_{TG_{V_G} \setminus A}$ is the restriction of $\overline{TG}$ to $TG_{V_G \setminus A}$.

Note that in the current theory, we do not consider attribute overriding. Moreover, in the case of diamond-like inheritance structures (with more than one path in the inheritance relation between two nodes), the attributes in the parent class would be copied several times in the child class. This does not present any problem for the theory. The discrimination between the abstract and the concrete closure of a type graph is necessary. The LHS and RHS of abstract productions considered in Section 5 are typed over the abstract closure, while ordinary host graphs and concrete productions are typed over the the concrete closure.

The left of Fig. 7 shows the closure of the type graph in Fig. 6, which corresponds to the type graph in Fig. 5 (note, however, the renaming of attribute edges due to inheritance).

Remark 1.

(1) Note, that we have $TG \subseteq \overline{TG}$ with TG_{V_i} for $i \in \{G, D\}$ and $TG_{E_i} \subseteq \overline{TG_{E_i}}$ if we identify $e \in TG_{E_i}$ with $(source_i(e), e, target_i(e)) \in \overline{TG_{E_i}}$ for $i \in \{G, NA, EA\}$.
Due to the existence of the canonical inclusion $TG \subseteq \overline{TG}$ all graphs typed over TG are also typed over $\overline{TG}$.

(2) Note that there are no inheritance relations in the abstract and the concrete closure of an $ATGI$, and hence no inheritance relations in the instance graphs defined below.

Instances of attributed type graphs with inheritance are typed attributed graphs. As before, we can notice a direct correspondence to object-oriented systems [6], where models consisting of objects with attribute values are instances of class diagram models, containing the corresponding classes, associations and attribute types.

Definition 11 (*Instance of ATGI*). An abstract instance of $ATGI$ is an attributed graph over $\overline{ATG}$, i.e. $(AG, type : AG \to \overline{ATG})$. Similarily, a concrete instance of $ATGI$ is an instance attributed graph over $\widehat{ATG}$, i.e. $(AG, type : AG \to \widehat{ATG})$.

An example of a concrete instance of the type graph with inheritance in Fig. 6 is shown to the right of Fig. 7.

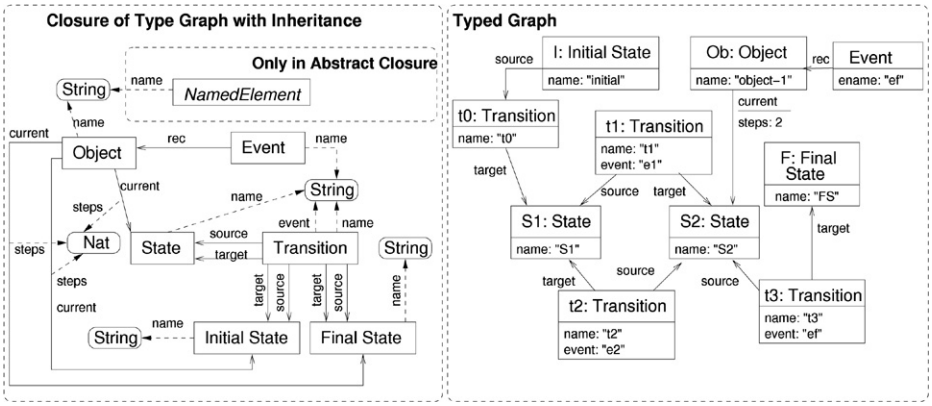


Fig. 7. Abstract and concrete closure of the type graph in Fig. 6 (Left). Instance graph of the type graph in Fig. 6 in compact notation (Right).

4.1. Attributed clan morphisms

To formally define the instance-type relation in the presence of inheritance, we introduce attributed clan morphisms. The choice of triples for the edges of a type graph's closure allows to express 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 a similar mapping called *clan morphism*, uniquely characterizing the type morphism into the flattened type graph.

Given an attributed type graph *ATGI* with inheritance we introduce in this section ATGI-clan morphisms. An ATGI-clan morphism $type : AG \to ATGI$ corresponds uniquely to a normal type morphism $\overline{type} : AG \to \overline{ATG}$, where $\overline{ATG}$ is the abstract closure of *ATGI* as discussed in the previous section.

Definition 12 (*ATGI-clan Morphism*). Given an attributed type graph with inheritance $ATGI = (TG, Z, I, A)$ with $TG_{V_D} = S_D^{'}$ and $ATG = (TG, Z)$ and an attributed graph $AG = (G, D)$ with $G = ((G_{V_i})_{i\in\{G,D\}}, (G_{E_i}, s_{G_i}, t_{G_i})_{i\in\{G,NA,EA\}})$ and $\uplus_{s\in S_D^{'}} D_s = G_{V_D}$, $type : AG \to ATGI$ with $type = (type_i)_{i\in\{V_G,V_D,E_G,E_{NA},E_{EA},D\}}$ and

- $type_{V_i} : G_{V_i} \to TG_{V_i} \quad (i \in \{G, D\})$
- $type_{E_i} : G_{E_i} \to TG_{E_i} \quad (i \in \{G, NA, EA\})$
- $type_D : D \to Z$ unique final *DSIG*-homomorphism

is called an *ATGI-clan morphism*, if

(0) $\forall s \in S_D^{'}$ the following diagram commutes,

$$\begin{array}{ccc} D_s & \xrightarrow{type_{D,s}} & Z_s = \{s\} \\ \downarrow & = & \downarrow \\ G_{V_D} & \xrightarrow[type_{V_D}]{} & TG_{V_D} = S_D^{'} \end{array}$$

i.e. $type_{V_D}(d) = s$ for $d \in D_s$ and $s \in S_D^{'}$.

(1) $type_{V_G} \circ s_{G_G}(e_1) \in clan_I(source_G \circ type_{E_G}(e_1)) \quad \forall e_1 \in G_{E_G}$
(2) $type_{V_G} \circ t_{G_G}(e_1) \in clan_I(target_G \circ type_{E_G}(e_1)) \quad \forall e_1 \in G_{E_G}$
(3) $type_{V_G} \circ s_{G_{NA}}(e_2) \in clan_I(source_{NA} \circ type_{E_{NA}}(e_2)) \quad \forall e_2 \in G_{E_{NA}}$
(4) $type_{V_D} \circ t_{G_{NA}}(e_2) = target_{NA} \circ type_{E_{NA}}(e_2) \quad \forall e_2 \in G_{E_{NA}}$
(5) $type_{E_G} \circ s_{G_{EA}}(e_3) = source_{EA} \circ type_{E_{EA}}(e_3) \quad \forall e_3 \in G_{E_{EA}}$
(6) $type_{V_D} \circ t_{G_{EA}}(e_3) = target_{EA} \circ type_{E_{EA}}(e_3) \quad \forall e_3 \in G_{E_{EA}}$.

An ATGI-clan morphism $type : AG \to ATG$ is called *concrete* if $type_{V_G}(n) \notin A$ for all $n \in G_{V_G}$.

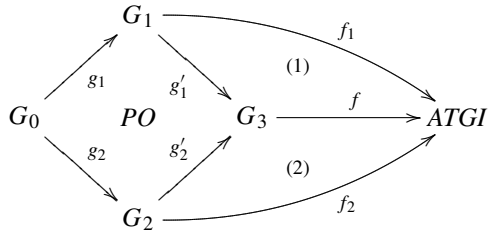


Fig. 8. Pushout w.r.t. concrete clan morphisms.

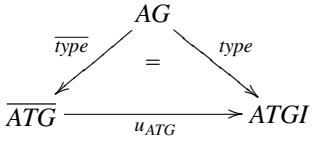
The following technical properties of ATGI-clan morphisms are needed to show the results in Section 5 based on Double Pushout Transformation in the category **AGraphs** of attributed graphs and morphisms. In order to show the bijective correspondence between ATGI-clan morphisms and normal type morphisms $\overline{type} : AG \to \overline{ATG}$ we first define a universal ATGI-clan morphism.

Definition 13 (*Universal ATGI-clan Morphism*). Given an attributed type graph with inheritance $ATGI = (TG, Z, I, A)$ then the *universal ATGI-clan morphism* $u_{ATG} : \overline{ATG} \to ATGI$ with $\overline{ATG} = (\overline{TG}, Z)$ is defined by
$u_{ATG,V_G} = id_1 : \overline{TG_{V_G}} \to TG_{V_G}$,
$u_{ATG,V_D} = id_2 : \overline{TG_{V_D}} \to TG_{V_D}$,
$u_{ATG,E_G} : \overline{TG_{E_G}} \to TG_{E_G}, u_{ATG,E_G}[(n_1, e, n_2)] = e \in TG_{E_G}$,
$u_{ATG,E_{NA}} : \overline{TG_{E_{NA}}} \to TG_{E_{NA}}, u_{ATG,E_{NA}}[(n_1, e, n_2)] = e \in TG_{E_{NA}}$,
$u_{ATG,E_{EA}} : \overline{TG_{E_{EA}}} \to TG_{E_{EA}}, u_{ATG,E_{EA}}[((n_{11}, e_1, n_{12}), e, n_2)] = e \in TG_{E_{EA}}$,
$u_{ATG,D} = id_Z : Z \to Z$.

Lemma 1. *The universal morphism $u_{ATG} : \overline{ATG} \to ATGI$ is an ATGI-clan morphism. ATGI-clan morphisms are closed under composition with attributed graph morphisms, short AG-morphisms. This means: given an AG-morphism $f : AG' \to AG$ and an ATGI-clan morphism $f' : AG \to ATGI$ then $f' \circ f : AG' \to ATGI$ is an ATGI-clan-morphism. If f' is concrete, so is $f' \circ f$.*

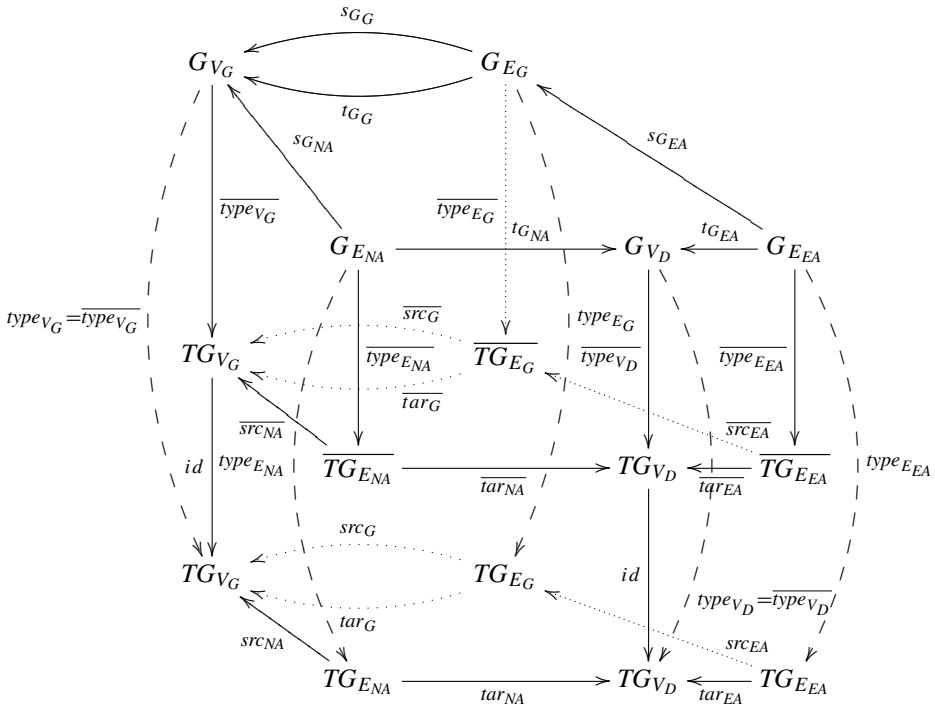
The following theorem is the key property relating ATGI-clan morphisms and AG-morphisms, which is essential to show the main results in this chapter:

Theorem 1 (*Universal ATGI-clan Property*). *For each ATGI-clan morphism $type : AG \to ATGI$, there is a unique AG-morphism $\overline{type} : AG \to \overline{ATG}$ s.t. $u_{ATG} \circ \overline{type} = type$.*



Construction. Given $type : AG \to ATGI$ with $AG = (G, D)$ we construct $\overline{type} : AG \to \overline{ATG}$ as follows:

- $\overline{type_{V_G}} = type_{V_G} : G_{V_G} \to TG_{V_G} = \overline{TG_{V_G}}$
- $\overline{type_{V_D}} = type_{V_D} : G_{V_D} \to TG_{V_D} = \overline{TG_{V_D}}$
- $\overline{type_{E_G}} : G_{E_G} \to \overline{TG_{E_G}}, \overline{type_{E_G}}(e_1) = (n_1, e'_1, n_2)$ with $e'_1 = type_{E_G}(e_1) \in TG_{E_G}, n_1 = \overline{type_{V_G}}(s_{G_G}(e_1)) \in TG_{V_G}, n_2 = \overline{type_{V_G}}(t_{G_G}(e_1)) \in TG_{V_G}$
- $\overline{type_{E_{NA}}} : G_{E_{NA}} \to \overline{TG_{E_{NA}}}, \overline{type_{E_{NA}}}(e_2) = (n_1, e'_2, n_2)$ with $e'_2 = type_{E_{NA}}(e_2) \in TG_{E_{NA}}, n_1 = \overline{type_{V_G}}(s_{G_{NA}}(e_2)) \in TG_{V_G}, n_2 = \overline{type_{V_D}}(t_{G_{NA}}(e_2)) \in TG_{V_D}$
- $\overline{type_{E_{EA}}} : G_{E_{EA}} \to \overline{TG_{E_{EA}}}, \overline{type_{E_{EA}}}(e_3) = ((n_{11}, e''_3, n_{12}), e'_3, n_2)$ with $e'_3 = type_{E_{EA}}(e_3) \in TG_{E_{EA}}, (n_{11}, e''_3, n_{12}) = \overline{type_{E_G}}(s_{G_{EA}}(e_3)) \in \overline{TG_{E_G}}, n_2 = \overline{type_{V_D}}(t_{G_{EA}}(e_3)) \in TG_{V_D}$
- $\overline{type_D} = type_D : D \to Z$



Lemma 1 implies that the composition $u_{ATG} \circ type$ is an ATGI-clan-morphism.

Lemma 2 (*Pushout Property of ATGI-clan Morphisms*).

(1) *A pushout in* **AGraphs** *is also a pushout w.r.t. (concrete) clan morphisms (cf. Fig. 8). This means more precisely: Given a pushout PO in* **AGraphs** *as shown in Fig. 8 with AG-morphisms* g_1, g_2, g'_1, g'_2 *and ATGI-clan morphisms* f_1, f_2 *with* $f_1 \circ g_1 = f_2 \circ g_2$, *then there is a unique ATGI-clan morphism* $f : G_3 \to ATGI$ *with* $f \circ g'_1 = f_1$ *and* $f \circ g'_2 = f_2$.

(2) *Double pushouts in* **AGraphs** *can be extended to double pushouts for attributed graphs with typing by concrete ATGI-clan-morphisms w.r.t. the match morphism and the production (cf. Fig. 9). This means more precisely: given pushouts* (1) *and* (2) *in* **AGraphs** *as shown in Fig. 9 and concrete ATGI-clan morphisms* $type_L$, $type_K$, $type_R$, *and* $type_G$ *for the production and the match graph G s.t.* (3), (4) *and* (5) *commute, then there are also unique concrete ATGI-clan morphisms* $type_D$ *and* $type_H$ *s.t.* (6) *and* (7) *commute.*

5. Typed attributed graph transformation with inheritance

In this section, we show how to adapt the concept of inheritance to the notions of typed attributed graph transformation, graph grammar and graph language. Our goal is to allow abstractly typed nodes in productions, such

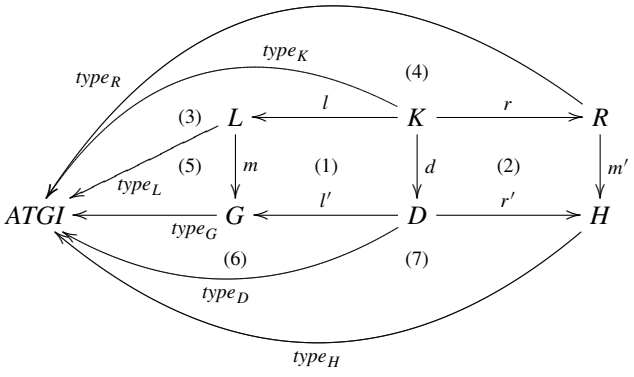


Fig. 9. Double pushout for attributed graphs with typing by concrete clan morphism.

that these abstract productions actually represent a set of structurally similar productions which we call *concrete productions*. In order to obtain all concrete productions for an abstract production, any combination of node types of the corresponding clans in the production's LHS (being of concrete or abstract type) must be considered. Nodes which are preserved by the production have to keep their type. Nodes which are created in the RHS must get a concrete type, since abstract types cannot be instantiated.

We define abstract and concrete transformations for abstract and concrete productions based on attributed type graphs with inheritance. The first main result shows the equivalence of abstract and concrete transformations. This allows us to use safely the more efficient presentation of abstract transformations with abstract productions, because they are equivalent to corresponding concrete transformations with concrete productions. The second main result – presented in the next section – shows the equivalence of attributed graph grammars with and without inheritance.

In the following we consider productions extended by NACs (see Section 2). As done for type graphs with inheritance, we define a flattening of abstract productions to concrete ones. Concrete productions are structurally equal to the abstract production, but their typing morphisms are finer than the ones of the abstract production and are concrete clan morphisms. A typing morphism is finer than another one, if it distinguishes from the other only by more concrete types in corresponding clans.

First we introduce the notion of type refinement in order to formalize the relationship between abstract and concrete productions to be defined below:

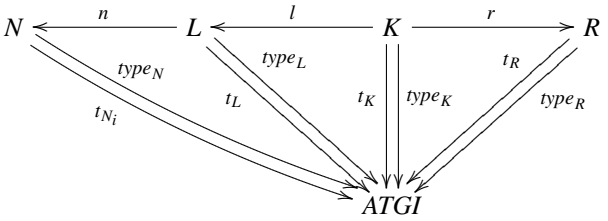
Definition 14 (*ATGI-Type Refinement*). Given an attributed graph $AG = (G, D)$ and ATGI-clan morphisms $type : AG \to ATGI$ and $type' : AG \to ATGI$, then $type'$ is called an *ATGI-type refinement* of $type$, written $type' \leq type$ if

- $type'_{V_G}(n) \in clan_I(type_{V_G}(n)), \quad \forall n \in G_{V_G}$
- $type'_X = type_X, \quad \text{for } X \in \{V_D, E_G, E_{NA}, E_{EA}, D\}$

Remark 2. Given ATGI-clan morphisms $type, type' : AG \to ATGI$ with $type' \leq type$ and an AG-morphism $g : AG' \to AG$, then also $type' \circ g \leq type \circ g$. Note that AG-morphism means morphism in the category **AGraphs**.

Definition 15 (*Abstract and Concrete Production*). An *abstract production* typed over $ATGI$ is given by $p = (L \xleftarrow{l} K \xrightarrow{r} R, type, NAC)$, where l and r are AG-morphisms, type is a triple of typing morphisms, i.e. ATGI-clan morphisms $type = (type_L : L \to ATGI, type_K : K \to ATGI, type_R : R \to ATGI)$, and NAC is a negative application condition, i.e. a set of triples $nac = (N, n, type_N)$ with an attributed graph N, an AG-morphism $n : L \to N$, and a typing ATGI-clan morphism $type_N : N \to ATGI$, s.t. the following conditions hold

- $type_L \circ l = type_K = type_R \circ r$
- $type_{R,V_G}(R'_{V_G}) \cap A = \emptyset$, where $R'_{V_G} := R_{V_G} - r_{V_G}(K_{V_G})$
- $type_N \circ n \leq type_L \quad \text{for all } (N, n, type_N) \in NAC$
- The datatype part of L, K, R and N is $T_{DSIG}(X)$, the term algebra of $DSIG$ with variables X, and l, r and n are data preserving, i.e. l_D, r_D, n_D are identities



A *concrete production* p_t w.r.t. an abstract production p is given by $p_t = (L \xleftarrow{l} K \xrightarrow{r} R, t, \overline{NAC})$, where t is a triple of concrete typing ATGI-clan morphisms $t = (t_L : L \to ATGI, t_K : K \to ATGI, t_R : R \to ATGI)$, s.t.

- $t_L \circ l = t_K = t_R \circ r$
- $t_L \leq type_L, t_K \leq type_K, t_R \leq type_R$
- $t_{R,V_G}(x) = type_{R,V_G}(x) \quad \forall x \in R'_{V_G}$

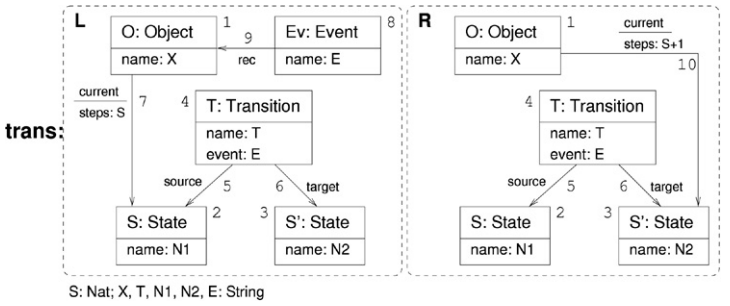


Fig. 10. Abstract production example.

- For each $(N, n, type_N) \in NAC$, we have all $(N, n, t_N) \in \overline{NAC}$ with concrete ATGI-clan morphisms t_N satisfying $t_N \circ n = t_L$ and $t_N \leq type_N$

The set of all concrete productions p_t w.r.t. an abstract production p is denoted by $\widehat{p}$.

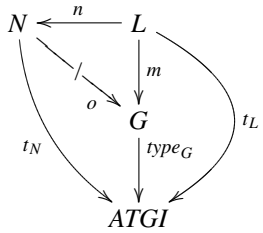
The application of an abstract production can be directly defined or expressed by using the flattening idea, i.e. to apply one of its concrete productions. Both the host graph and the concrete production are typed by concrete clan morphisms such that we can define the application of concrete productions. Later we will also define the application of an abstract production directly and show the equivalence of both.

Fig. 10 shows an example of an abstract production, where variables S, X, T, $N1$, $N2$ and the term $S+1$ are taken as attributes. The production moves an object *current* edge through a transition marked with an event the object has received. In addition, the number of steps in the current edge is increased. This abstract production is equivalent to nine concrete productions, resulting by the substitution of the *State* node by two more concretely typed nodes, of types *Initial State* and *Final State*. We call the production *abstract*, although there is no abstract node in the production, but one of the nodes (*State*) can be substituted by its inheritance clan (which includes itself).

Definition 16 (*Application of Concrete Production*). Let $p_t = (L \xleftarrow{l} K \xrightarrow{r} R, t, \overline{NAC})$ be a concrete production, $(G, type_G)$ a typed attributed graph with a concrete ATGI-clan morphism $type_G : G \to ATGI$ and $m : L \to G$ an AG-morphism. Morphism m is a *consistent match* w.r.t. p_t and $(G, type_G)$, if

- m satisfies the gluing condition [14] w.r.t. the untyped production $L \xleftarrow{l} K \xrightarrow{r} R$ and the attributed graph G,
- $type_G \circ m = t_L$, and
- m satisfies the negative application conditions $\overline{NAC}$, i.e. for each $(N, n, t_N) \in \overline{NAC}$ it holds, that there exists no AG-morphism $o : N \to G$ in $\mathcal{M}'$, such that $o \circ n = m$ and $type_G \circ o = t_N$. $\mathcal{M}'$ is a suitable class of morphisms for application conditions, for example, the class of injective morphisms.

Given a consistent match m, the concrete production can be applied to the typed attributed graph $(G, type_G)$, yielding a typed attributed graph $(H, type_H)$ by constructing the DPO of l, r and m and applying Lemma 2.2.
We write $(G, type_G) \overset{p_t, m}{\Longrightarrow} (H, type_H)$ for such a *direct transformation* (see Definition 4).



The classical theory of typed attributed graph transformations relies on typing morphisms which are normal graph morphisms, i.e. no clan morphisms. For showing the equivalence of abstract and concrete graph transformations, we first have to consider the following: The application of a concrete production typed by concrete clan morphisms is

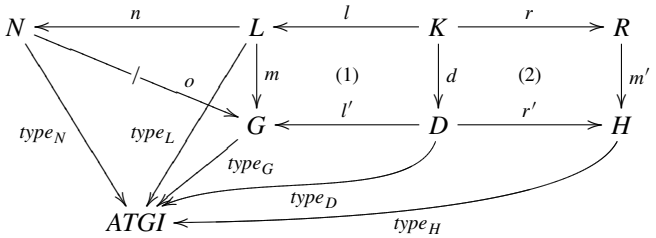


Fig. 11. Match and application of abstract rule.

equivalent to the application of the same production correspondingly typed over the concrete closure of the given type graph. This lemma is formulated and proven in Lemma 2 for productions without NACs.

Although the semantics for the application of an abstract production can be given by the application of its concrete productions, this solution is not efficient at all. For example, a tool implementing graph transformation with node type inheritance would have to check all concrete productions of an abstract production to find the right one to apply to a given instance graph. Thus, as a next step, we want to examine a more direct way to apply an abstract production. Since abstract and concrete productions differ only in typing, but have the same structure, a match morphism from the LHS of a concrete production into a given instance graph is also a match morphism for its abstract production. But of course, the typing morphisms differ. Using the notion of type refinement, however, we can express a compatibility property.

Definition 17 (*Application of Abstract Production*). Let $p = (L \xleftarrow{l} K \xrightarrow{r} R, type, NAC)$ be an abstract production typed over an attributed type graph with inheritance *ATGI*, $(G, type_G)$ a typed attributed graph with a concrete ATGI-clan morphism $type_G : G \to ATGI$ and $m : L \to G$ an AG-morphism. Morphism m is called consistent *match* w.r.t. p and $(G, type_G)$, if:

- m satisfies the gluing condition w.r.t. the untyped production $L \xleftarrow{l} K \xrightarrow{r} R$ and the attributed graph G i.e. pushout (1) in Fig. 11 exists,
- $type_G \circ m \leq type_L$.
- $t_{K,V_G}(x_1) = t_{K,V_G}(x_2)$ for $t_K = type_G \circ m \circ l$ and all $x_1, x_2 \in K_{V_G}$ with $r_{V_G}(x_1) = r_{V_G}(x_2)$.
- m satisfies *NAC*, i.e. for each $nac = (N, n, type_N) \in NAC$ it holds that there exists no AG-morphism $o : N \to G$ in $\mathcal{M}'$ (see Definition 16) such that $o \circ n = m$ and $type_G \circ o \leq type_N$.

Given a consistent match m, the abstract production can be applied to $(G, type_G)$ yielding an *abstract direct transformation* $(G, type_G) \overset{p,m}{\Longrightarrow} (H, type_H)$ with the concrete ATGI-clan morphism $type_H$ as follows:

(1) Construct the (untyped) DPO of l, r and m in **AGraphs** given by pushouts (1) and (2) in Fig. 11.
(2) Construct $type_D$ and $type_H$ as follows:
   - $type_D = type_G \circ l'$;
   - $type_{H,X}(x) = \underline{\text{if}}\ x = r'_X(x')\ \underline{\text{then}}\ type_{D,X}(x')\ \underline{\text{else}}\ type_{R,X}(x'')$, where $m'(x'') = x$ and $X \in \{V_G, V_D, E_G, E_{NA}, E_{EA}, D\}$.

Remark 3. $type_H$ is a well-defined ATGI-clan morphism with $type_H \circ r' = type_D$ and $type_H \circ m' \leq type_R$. Moreover, we have $type_G \circ m \leq type_L$ (as required) and $type_D \circ d \leq type_K$ (see Lemma 3(3)). The third match condition is not needed if r_{V_G} is injective (as it is the case in most examples).

Fig. 12 shows an example of the application of the abstract production defined in Fig. 10 to a graph. While the S node in the production is matched to the $S2$ node in G with the same type, the S' node is matched to the F node, of type *Final State*.

Lemma 3 (*Construction of Concrete and Abstract Transformations*). *Given an abstract production* $p = (L \xleftarrow{l} K \xrightarrow{r} R, type, NAC)$ *with* $NAC = \{(N_i, n_i, type_{N_i}) | i \in I\}$, *a concrete typed attributed graph* $(G, type_G : G \to ATGI)$ *and a consistent match morphism* $m : L \to G$ *w.r.t.* p *and* $(G, type_G)$, *we have (cf. Fig. 13):*

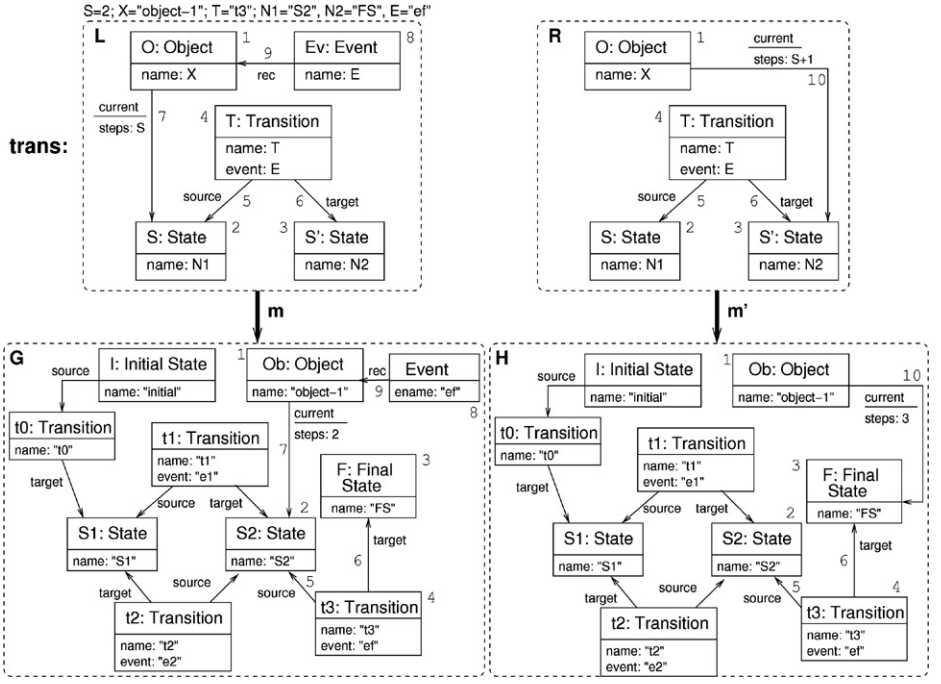


Fig. 12. Transformation example with abstract production.

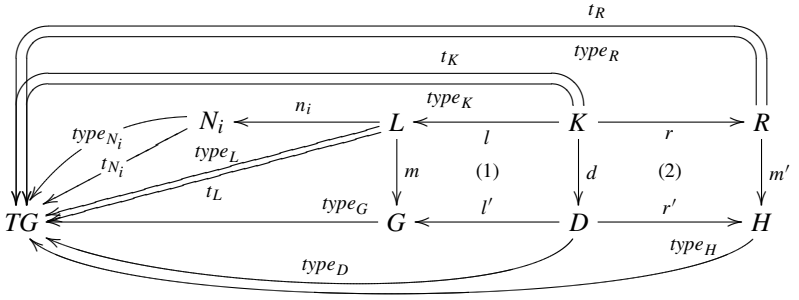


Fig. 13. Matching of abstract and concrete productions.

(1) *There is a unique concrete production* $p_t \in \widehat{p}$ *with* $p_t = (L \xleftarrow{l} K \xrightarrow{r} R, t, \overline{NAC})$ *and* $t_L = type_G \circ m$. *In this case,* t_K, t_R *and* $\overline{NAC}$ *are defined by:*
- $t_K = t_L \circ l$;
- $t_{R,V_G}(x) = \underline{if}\ x = r_{V_G}(x')\ \underline{then}\ t_{K,V_G}(x')\ \underline{else}\ type_{R,V_G}(x)$ *for* $x \in R_{V_G}$;
- $t_{R,X} = type_{R,X} \quad$ *for* $X \in \{V_D, E_G, E_{NA}, E_{EA}, D\}$;
- $\overline{NAC} = \bigcup_{i \in I}\{(N_i, n_i, t_{N_i}) | t_{N_i}$ *is a concrete ATGI-clan morphism with* $t_{N_i} \leq type_{N_i}$ *and* $t_{N_i} \circ n_i = t_L\}$.

(2) *There is a concrete direct transformation* $(G, type_G) \overset{p_t,m}{\Longrightarrow} (H, type_H)$ *with consistent match* m *w.r.t.* p_t, *and* $type_D = type_G \circ l'$ *and* $type_H$ *uniquely defined by* $type_D$, t_R *and pushout properties of (2) (see Lemma 2), where* $type_H : H \to ATGI$ *is a concrete ATGI-clan morphism explicitly given by:*
$type_{H,X}(x) = \underline{if}\ x = r'_X(x')\ \underline{then}\ type_{D(,X}x')\ \underline{else}\ t_{R,X}(x'')$
where $m'(x'') = x$ *and* $X \in \{V_G, V_D, E_G, E_{NA}, E_{EA}, D\}$.

(3) *The concrete direct transformation becomes an abstract direct transformation (see Definition 17):*
$(G, type_G) \overset{p,m}{\Longrightarrow} (H, type_H)$ *with* $type_D = type_H \circ r'$, $type_G \circ m \leq type_L$, $type_D \circ d \leq type_K$ *and* $type_H \circ m' \leq type_R$, *where the typing* $t = (t_L, t_K, t_R)$ *of the concrete production* p_t *is replaced by type* $= (type_L, type_K, type_R)$ *of the abstract production* p.

6. Equivalence results

After having defined concrete and abstract transformations, the question arises how these two kinds of graph transformation are related to each other. Theorem 2 will answer this question by showing that for each abstract transformation applying an abstract production p there is a concrete transformation applying a concrete production w.r.t. p, and vice versa. Thus, an application of an abstract production can also be flattened to a concrete transformation. The result allows us to use the dense form of abstract productions in graph transformations on one hand, and to reason about this new form of graph transformation by flattening it to usual typed attributed graph transformation which comes along with a rich theory. Furthermore, we show the equivalence of typed attributed graph grammars with and without inheritance. A summary of the main results, with the relationships between the theorems is shown in Fig. 18.

In the following all typing morphisms $type : AG \to ATGI$ are ATGI-clan morphisms, unless stated otherwise. With $\overline{type} : AG \to \overline{ATG}$ we denote the corresponding graph morphism.

Theorem 2 (*Equivalence of Transformations*). *Given an abstract production* $p = (L \xleftarrow{l} K \xrightarrow{r} R, type, NAC)$ *over an attributed type graph ATGI with inheritance, a concrete typed attributed graph* $(G, type_G)$ *and a match morphism* $m : L \to G$ *(which satisfies the gluing condition w.r.t. the untyped production* $L \longleftarrow K \longrightarrow R$*). Then the following statements are equivalent, where* $(H, type_H)$ *is the same concrete typed graph in both cases:*

(1) $m : L \to G$ *is a consistent match w.r.t. the abstract production* p *yielding an abstract direct transformation* $(G, type_G) \overset{p,m}{\Longrightarrow} (H, type_H)$.
(2) $m : L \to G$ *is a consistent match w.r.t. the concrete production* $p_t = (L \leftarrow K \to R, t, \overline{NAC})$ *with* $p_t \in \widehat{p}$ *and* $t_L = type_G \circ m$ *(where* t_K, t_R *and* $\overline{NAC}$ *are uniquely defined by Lemma 3(1)) yielding a concrete direct transformation* $(G, type_G) \overset{p_t,m}{\Longrightarrow} (H, type_H)$.

Theorem 2 allows us to use the dense form of abstract productions for model manipulation instead of generating and holding all concrete productions, i.e. abstract transformations are much more efficient than concrete transformations. That means, on the one hand we have an efficient procedure and on the other hand we are sure that the result is the same as using concrete productions. Moreover, as a consequence of Theorem 2, graph languages built over abstract productions are equivalent to graph languages that are built over a corresponding set of concrete productions. Moreover, graph grammars with inheritance are equivalent to corresponding ones without inheritance, where, however, the type graph *ATGI* has to be replaced by the closure $\overline{ATG}$. Before showing these main results we define graph grammars and languages in our context:

Definition 18 (*ATGI Graph Grammar and Language*). Given an attributed type graph *ATGI* and an attributed graph G typed over *ATGI* with a concrete ATGI-clan morphism $type_G$, an *ATGI-graph grammar* is denoted by $GG = (ATGI, (G, type_G : G \to ATGI), P)$, where P is a set of abstract productions that are typed over *ATGI*.

The corresponding *graph language* is defined by the set of all concretely typed graphs which are generated by an abstract transformation (cf. Definitions 16 and 17): $L(GG) = \{(H, type_H : H \to ATGI) \mid \exists\ abstract\ transformation (G, type_G) \overset{*}{\Rightarrow} (H, type_H)\}$.

Remark. $type_H$ is always concrete by Lemma 3 item 2.

Theorem 3 (*Equivalence of Attributed Graph Grammars*). *For each ATGI-graph graph grammar* $GG = (ATGI, (G, type_G), P)$ *with abstract productions* P *there are:*

(1) *An equivalent ATGI-graph grammar* $\widehat{GG} = (ATGI, (G, type_G), \widehat{P})$ *with concrete productions* $\widehat{P}$*, i.e.* $L(GG) = L(\widehat{GG})$*;*
(2) *An equivalent typed attributed graph grammar without inheritance* $\overline{GG} = (\overline{ATG}, (G, \overline{type_G}), \overline{P})$ *typed over* $\overline{ATG}$ *where* $\overline{ATG}$ *is the closure of ATGI, and with productions* $\overline{P}$*, i.e.* $L(GG) \overset{\sim}{=} L(\overline{GG})$*, that means:* $(G, type_G) \in L(GG) \Leftrightarrow (G, \overline{type_G}) \in L(\overline{GG})$.

Construction.

(1) The set $\widehat{P}$ is defined by $\widehat{P} = \cup_{p \in P} \widehat{p}$ with $\widehat{p}$ the set of all concrete productions w.r.t. p;

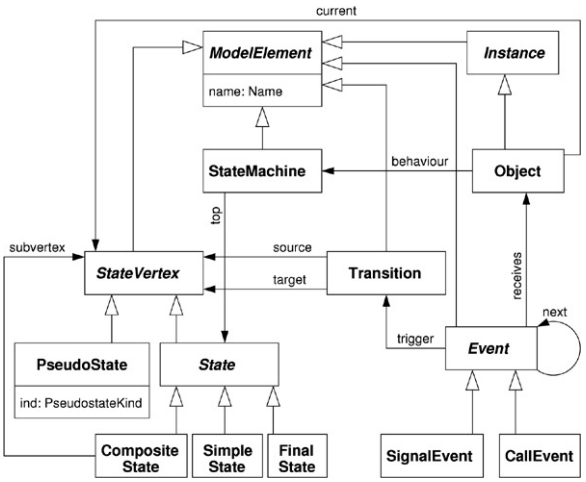


Fig. 14. Type graph with inheritance for statecharts.

(2) $\overline{type_G} : G \to \overline{ATG}$ is the graph morphism corresponding to the ATGI-clan morphism $type_G$ (see Theorem 1). $\overline{P}$ is defined by $\overline{P} = \cup_{p \in P} \{\overline{p_t} \mid p_t \in \widehat{p}\}$. where for $p_t \in \widehat{p}$ with $p_t = (p, t, \overline{NAC})$ we define $\overline{p_t} = (p, \bar{t}, \overline{NAC'})$ with $u_{ATG} \circ \overline{t_X} = t_X$ for $X \in \{L, K, R\}$ and $\overline{NAC'}$ is defined by NAC as follows:
For each $(N, n, t_N) \in \overline{NAC}$ we have all $(N, n, \overline{t_N}) \in \overline{NAC'}$ with $t_N = u_{ATG} \circ \overline{t_N}$.

Remark 4. In grammar $\overline{GG}$ of Part 2 using the abstract closure $\overline{ATG}$ of $ATGI$, graphs with concrete typing are generated only. In fact there is also an equivalent grammar $\overline{GG'}$ with type graph $\widehat{ATG}$, the concrete closure of $ATGI$.

7. Case study

In this section we extend the previous examples by presenting a more detailed case study of the simulation of Statecharts. The main addition with respect to Fig. 6 is that we consider hierarchical states (composite states have subvertices). In addition, objects have a queue of pending events. The first event in the queue points to the object by means of edge *receives*. Events in the queue point to the next one by means of the *next* edge. The type graph with inheritance is shown in Fig. 14 and it is in fact a simplification of the one shown in the UML specification [26] (thus, we only consider a subset of Statecharts). According to this specification, the *PseudostateKind* is an enumerate type, but we only consider the *initial* value. Note in addition, that the kind of Statecharts we deal with should be constrained more, either by defining extra constraints (like multiplicities) that the instance graphs should verify, or by defining a generation grammar (as we did, for example, in [2]). This grammar ensures that each state machine contains a unique topmost initial state of type *Composite State*, and that each *composite state* has a unique initial state.

Fig. 15 shows an instance graph of the type graph in Fig. 14. We have used abbreviations to depict the node types. The right part of the figure shows a *concrete syntax* representation of the instance graph. In a visual language, the concrete syntax defines how the different elements of the language are graphically represented. In our case, we use the standard UML of representing composite states by placing the substates inside the composite state.

Fig. 16 shows a set of abstract productions for simulating our subset of statecharts. All productions are abstract because the node types *EV* (*Event*), *ST* (*State*) and *SV* (*StateVertex*) are abstract. We have used a condensed notation for NACs (used in tools such as AGG [3] and AToM^3 [22]). In this notation, the NAC only shows: (i) nodes not having a pre-image in the LHS (roughly, those in $N - n(L)$), and their context nodes (those directly connected via edges), or (ii) nodes whose type is refined from the LHS. The rest of the LHS is isomorphically copied in the NAC.

The first production adds the *current* relationship (c) to an object (OB) if it does not already have one. The starting state is the initial state of the *top* state. Production 2 models a state change due to a transition from the current state. In this abstract production, *StateVertex* and *Event* are abstract nodes. This feature allows us to condense in a single abstract production the combinations of all concrete sub-types of *StateVertex* and *Event* nodes. In fact, the number of concrete productions according to Definition 15 is very large, because there are three *Event* nodes with two concrete instantiations and two *StateVertex* nodes with four concrete instantiations each. Altogether we have $2^3 \times 4^2 = 128$

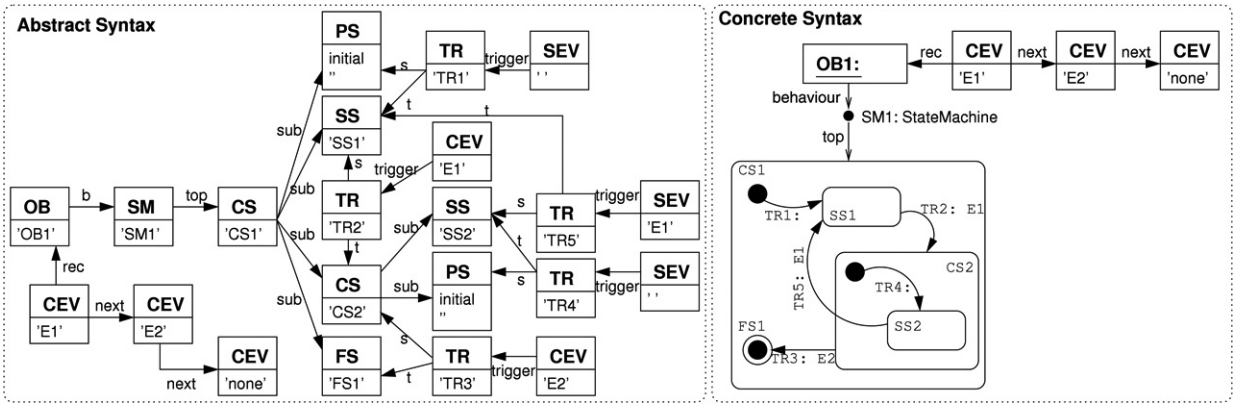


Fig. 15. Statecharts example. Abstract syntax (left), Concrete syntax (right).

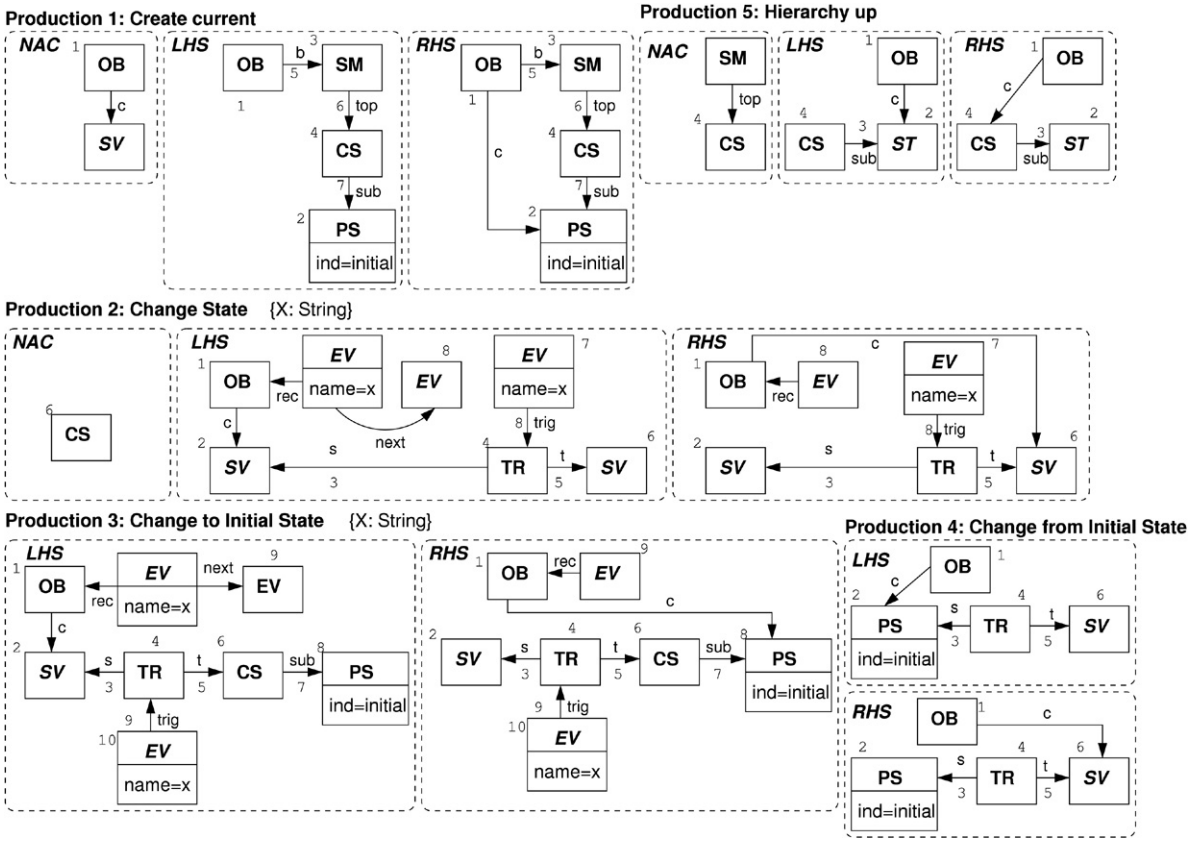


Fig. 16. Productions for the simulation of statecharts.

different concrete productions. The NAC in this production forbids its application if the target node is a *composite node* (the type of node 6 in the LHS is refined in the NAC), in this case, production three should be used.

Production three 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*). Production four 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, production five 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 production allows going up in the *subvertex*

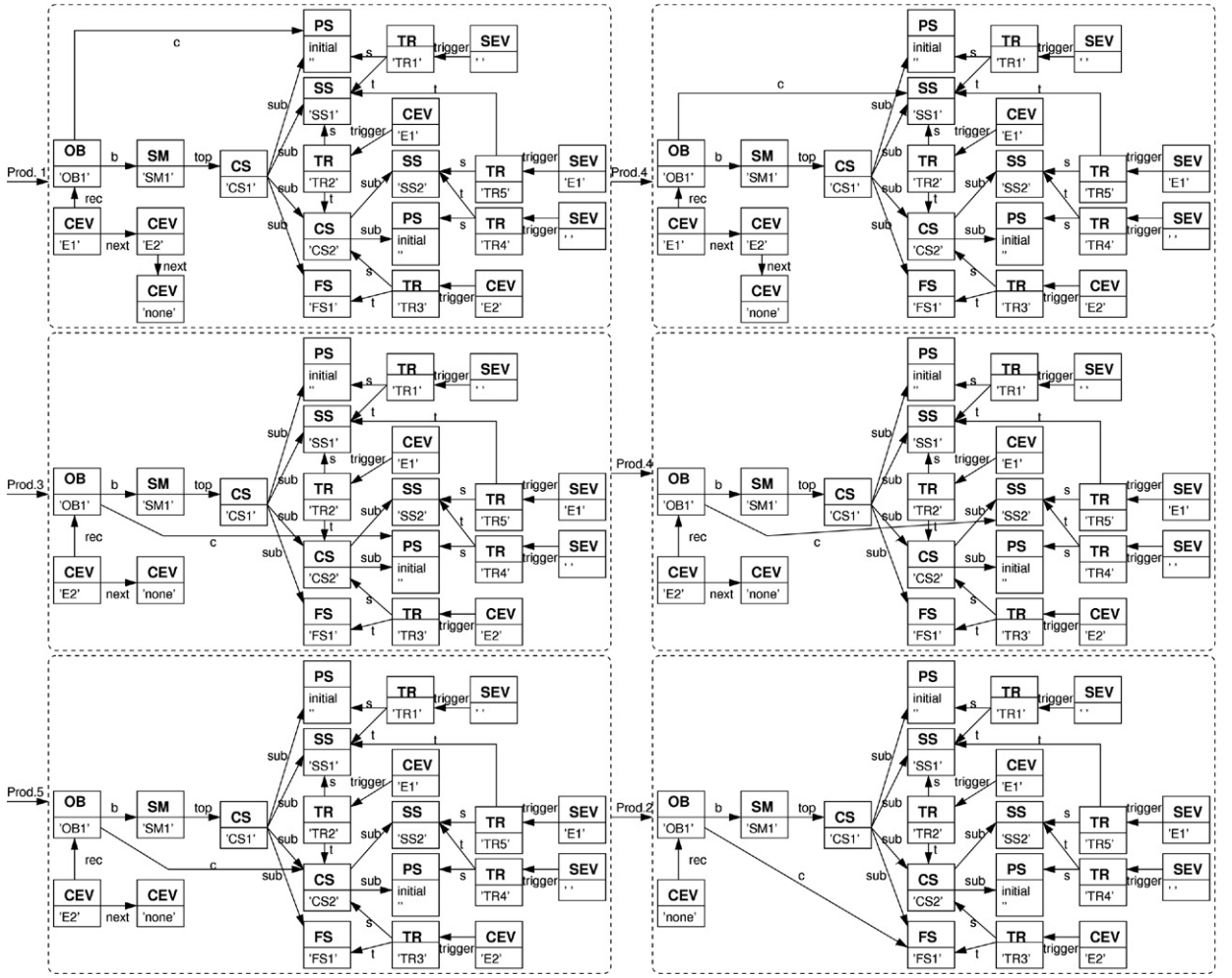


Fig. 17. A transformation example.

hierarchy starting from the *current* state. We cannot apply this production, 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.

Fig. 17 shows a sequence of direct transformations of the previous grammar applied to the Statechart in Fig. 15, according to the application of abstract productions in Definition 17. In the first step, we apply production one, setting the *current* state pointer to the *PseudoState* (*initial* kind) of the *top* state. Then, abstract production four moves the *current* state to node *'SS1'*. Node 6 in the production (*StateVertex* type) is matched to node *'SS1'* in the graph, typed over *SimpleState*. Next, abstract production three is applied and the pointer is moved to the initial state of composite state *'CS2'*. Node 2 (of type *StateVertex*) in the production matches node *'SS1'* of type *SimpleState* in the graph; and the *Event* is of type *CallEvent*. Then, abstract production four can be applied, which moves the pointer to node *'SS2'*. The type instantiation is from *StateVertex* in the production to *SimpleState* in the graph. Now, abstract production five is applied, moving the *current* pointer up in the hierarchy to node *'CS2'*. The type of node 2 (*CompositeState*) in the production is instantiated to *SimpleState* of node *'SS2'* in the graph. For the following step, abstract production two 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 production can be applied anymore.

According to Theorems 2, 3(1) and Definition 17, this transformation with abstract productions P, is equivalent to a corresponding transformation with concrete productions $\widehat{P}$ typed over *ATGI* in Fig. 14. Moreover, by Theorems 1 and 3.2, it is equivalent to a transformation with productions $\overline{P}$ typed over the closure $\overline{ATG}$ of *ATGI* according to the theory of typed attriuted graph transformation without inheritance (see [12]). Nonetheless, note that the set P of abstract productions is much smaller than $\widehat{P}$ and $\overline{P}$ as discussed above for production two.

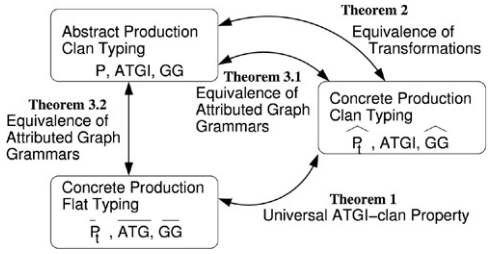


Fig. 18. Summary of main results.

8. Conclusion

In this paper we have presented a formal integration of node type inheritance with typed attributed graph transformation. The new concept allows the definition of abstract productions, in which abstractly typed nodes may appear. These can be matched to nodes of any of their concrete subtypes. The main results of the paper are summarized in Fig. 18 which shows that abstract productions (transformations) can be flattened to concrete productions (transformations), typed over the abstract or the flattened concrete type graph.

The presented inheritance concept is extremely useful in applications, since graph grammars and graph transformation systems can be notably more compact. This has already been demonstrated in our previous paper [2]. However, that work was restricted to graph transformation without an attribution concept. In this extended paper, we have shown how to obtain a formal integration of an inheritance concept with typed attributed graph transformation as presented in [12,10]. The presented concepts have been implemented in the AGG [31] and AToM^3 [22] tools. Altogether, this work is a crucial step towards a precise integration of object-orientation and graph transformation concepts.

In the literature, the inheritance concept has been integrated to a number of graph transformation approaches and corresponding tools already, as e.g. Progres [27], Fujaba [16], ViaTra [33], GReAT [1]. Some of them follow quite a different approach for expressing the inheritance concept. For example, Progres is based on Logic-Based Structure Replacement Systems [28], in which graphs and type graphs (so called structure schemas) are encoded as a set of first-order predicate logic formulae. In this way, inheritance can be expressed as a set of implications. Fujaba uses a UML-like notation for graphs and rules, which are translated into progress models. While those integrations show a lot of similarities with ours from the operational point of view, none of them is thoroughly formalized such that existing theory becomes applicable to this new form of graph transformation.

Another related approach (although for the Single Pushout approach to graph transformation) can be found in [23]. In that work, the inheritance is encoded by considering graphs whose nodes and edges are partially ordered, in such a way that typing and graph morphisms should preserve such order. Although they consider overriding, they are limited to single inheritance and do not consider attribution in our sense.

Although existing theoretical results become applicable to attributed graph transformation with inheritance already, some of the analysis techniques available might not be usable in a seamless way yet. While constraint checking, introduced in [19], has been lifted to graph transformation with inheritance already in [30], this lifting has to be extended to attributed graph transformation in future work. Termination of graph transformation [11] can be easily used on the basis of graph transformation without inheritance, since the results of termination checks do not have to be further interpreted. Instead, critical pair analysis [17] which is useful to, e.g. optimize visual language parsers [7] and to show correctness of model transformation [18], should be lifted to type graphs with inheritance. While minimal conflicts of flat productions can be considered already, a conflict consideration on the level of abstract productions is desirable from the application point of view.

Having extended graph transformation by node type inheritance, one might also extend it to edge type inheritance. This extension has been already considered in [30] for graph transformation without attributes. We consider edge type inheritance to restrict the combinations of source and target sub-types allowed when node type inheritance is used. The formalization of edge inheritance differs completely from that of node type inheritance, since edge inheritance relations are translated to graph constraints, as well as multiplicities for node and edge types. Again this work has to be extended to attributed graph transformation. By integrating all main object-orientation concepts to graph transformation, we aim at a comprehensive and visual formal framework with a mature theory to be applied to object-oriented modelling and meta-modelling.

Acknowledgements

This work has been partially sponsored by the EC's Human Potential Programme under contract HPRN-CT-2002-00275 (SegraVis: Syntactic and Semantic Integration of Visual Modelling Techniques), the IST-2005-16004 Integrated Project SENSORIA (Software Engineering for Service-Oriented Overlay Computers), the German Research Foundation with project "Application of graph transformation to visual modeling languages" and the Spanish Ministry of Science and Education with project MOSAIC TSI2005-08225-C07-06. The authors would also like to thank the referees for their useful comments.

Appendix. Proofs of the theorems

In the following we give the proofs of Theorems 1–3. For the proofs of Lemmas 1–3, we refer to our Technical Report [20].

Proof 1 (*Theorem* 1). By Lemmas 1 and 2, m_{ATG} is an ATGI-clan morphism and composition is well-defined. We have to show

(1) $type : AG \to \overline{ATG}$ is well-defined AG-morphism
(2) $u_{ATG} \circ type = type'$
(3) For each AG-morphism $f : AG \to \overline{ATG}$ with $u_{ATG} \circ f = type'$ we have $f = type$

(1) We have to show that $type : AG \to \overline{ATG}$ is well-defined AG-morphism.
  (a) Well definedness means $type_{E_i}(e_i) \in \overline{TG_{E_i}}$ for $i = G, NA, EA$
    (i) $type_{E_G}(e_G) = (n_1, e'_1, n_2) \in \overline{TG_{E_G}}$ means to show
    $e'_1 \in TG_{E_G}, n_1 \in clan_I(source_G(e'_1)), n_2 \in clan_I(target_G(e'_1))$.
    By definition of $type_{E_G}$ we have $\underline{e'_1} = type'_{E_G}(e_G) \in \underline{TG_{E_G}}$,
    $\underline{n_1} = type_{V_G}(s_{G_G}(e_G)) = type'_{V_G}(s_{G_G}(e_G)) \in clan_I(source_G \circ type'_{E_G}(e_G)) = \underline{clan_I(source_G(e'_1))}$
    $\underline{n_2} = type_{V_G}(t_{G_G}(e_G)) = type'_{V_G}(t_{G_G}(e_G)) \in clan_I(target_G \circ type'_{E_G}(e_G)) = \underline{clan_I(target_G(e'_1))}$
    (ii) $type_{E_{NA}}(e_{NA}) = (n_1, e'_2, n_2) \in \overline{TG_{E_{NA}}}$ means to show
    $e'_2 \in TG_{E_{NA}}, n_1 \in clan_I(source_{NA}(e'_2)), n_2 = target_{NA}(e'_2)$.
    By definition of $type_{E_{NA}}$ we have $\underline{e'_2} = type'_{E_{NA}}(e_{NA}) \in \underline{TG_{E_{NA}}}$,
    $\underline{n_1} = type_{V_G}(s_{G_{NA}}(e_{NA})) = type'_{V_G}(s_{G_{NA}}(e_{NA})) \in clan_I(source_{NA} \circ type'_{E_{NA}}(e_{NA})) = \underline{clan_I(source_{NA}(e'_2))}$
    $\underline{n_2} = type_{V_G}(t_{G_{NA}}(e_{NA})) = type'_{V_G}(t_{G_{NA}}(e_{NA})) = target_{NA} \circ type'_{E_{NA}}(e_{NA}) = \underline{target_{NA}(e'_2)}$
    (iii) $type_{E_{EA}}(e_{EA}) = ((n_{11}, e''_3, n_{12}), e'_3, n_2) \in \overline{TG_{E_{EA}}}$ means to show
    $e'_3 \in TG_{E_{EA}}, e''_3 = source_{EA}(e'_3) \in TG_{E_G}$,
    $n_{11} \in clan_I(source_G(e''_3)), n_{12} \in clan_I(target_G(e''_3))$,
    $n_2 = target_{EA}(e'_3) \in TG_{V_D}$.
    By definition of $type_{E_{EA}}$ we have $\underline{e'_3} = type'_{E_{EA}}(e_{EA}) \in \underline{TG_{E_{EA}}}$,
    $n_2 = type_{V_{NA}}(t_{G_{EA}}(e_{EA}))$, $(n_{11}, e''_3, n_{12}) = type_{E_G}(s_{G_{EA}}(e_{EA}))$, which is in $\overline{TG_{E_G}}$ according to $type_{E_G}$ in step (i).
    By Definition 10 this implies: $\underline{n_{11} \in clan_I(source_G(e''_3))}$, $\underline{n_{12} \in clan_I(target_G(e''_3))}$, $\underline{e''_3 \in TG_{E_G}}$.
  Now using $type'$ ATGI-clan morphism we have
  $\underline{n_2} = type_{V_D}(t_{G_{EA}}(e_{EA})) = type'_{V_D}(t_{G_{EA}}(e_{EA})) = target_{EA} \circ type'_{E_{EA}}(e_{EA}) = \underline{target_{EA}(e'_3)}$ and now $type_{E_G}(s_{G_{EA}}(e_{EA})) = (n_{11}, e''_3, n_{12}) \in \overline{TG_{E_G}}$ implies by definition of $type_{E_G}(e_G)$
  $\underline{e''_3} = type'_{E_G}(s_{G_{EA}}(e_{EA})) \overset{(*)}{=} source_{EA} \circ type'_{E_{EA}}(e_{EA}) = \underline{source_{EA}(e'_3)}$,
  where $(*)$ holds, because $type'$ is ATGI-clan morphism.
  (b) The AG-morphism property of $type : AG \to \overline{ATG}$ requires to show the following properties: (i)–(vii)
    (i) $type_{V_D}(d) = s$ for $d \in D_s$ and $s \in S'_D$
    this is true because corresponding property holds for $type'_{V_D}$ and $type'_{V_D} = type_{V_D}$

(ii) $type_{V_G} \circ s_{G_G}(e_1) = \overline{source_G} \circ type_{E_G}(e_1) \quad \forall e_1 \in G_{E_G}$
By definition of $type_{E_G}$ we have
$type_{E_G}(e_1) = (n_1, e'_1, n_2)$ with $n_1 = type_{V_G}(s_{G_G}(e_1)) \in TG_{V_G} \Rightarrow$
$\overline{source_G} \circ type_{E_G}(e_1) = \overline{source_G}[(n_1, e'_1, n_2)] = n_1 = \overline{type_{V_G}(s_{G_G}(e_1))}$

(iii) $type_{V_G} \circ t_{G_G}(e_1) = \overline{target_G} \circ type_{E_G}(e_1) \quad \forall e_1 \in G_{E_G}$
By definition of $type_{E_G}$ we have
$type_{E_G}(e_1) = (n_1, e'_1, n_2)$ with $n_1 = type_{V_G}(t_{G_G}(e_1)) \in TG_{V_G} \Rightarrow$
$\overline{target_G} \circ type_{E_G}(e_1) = \overline{target_G}[(n_1, e'_1, n_2)] = n_1 = \overline{type_{V_G}(t_{G_G}(e_1))}$

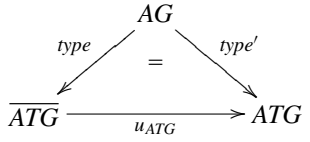
(iv) $type_{V_G} \circ s_{G_{NA}}(e_2) = \overline{source_{NA}} \circ type_{E_{NA}}(e_2) \quad \forall e_2 \in G_{E_{NA}}$
By definition of $type_{E_{NA}}$ we have
$type_{E_{NA}}(e_2) = (n_1, e'_2, n_2)$ with $n_1 = type_{V_G}(s_{G_{NA}}(e_2)) \in TG_{V_G} \Rightarrow$
$\overline{source_{NA}} \circ type_{E_{NA}}(e_2) = \overline{source_{NA}}[(n_1, e'_2, n_2)] = n_1 = \overline{type_{V_G}(s_{G_{NA}}(e_2))}$

(v) $type_{V_D} \circ t_{G_{NA}}(e_2) = \overline{target_{NA}} \circ type_{E_{NA}}(e_2) \quad \forall e_2 \in G_{E_{NA}}$
By definition of $type_{E_{NA}}$ we have
$type_{E_{NA}}(e_2) = (n_1, e'_2, n_2)$ with $n_2 = type_{V_D}(t_{G_{NA}}(e_2)) \in TG_{V_D} \Rightarrow$
$\overline{target_{NA}} \circ type_{E_{NA}}(e_2) = \overline{target_{NA}}[(n_1, e'_2, n_2)] = n_2 = \overline{type_{V_D}(t_{G_{NA}}(e_2))}$

(vi) $type_{E_G} \circ s_{G_{EA}}(e_3) = \overline{source_{EA}} \circ type_{E_{EA}}(e_3) \quad \forall e_3 \in G_{E_{EA}}$
By definition of $type_{E_{EA}}$ we have
$type_{E_{EA}}(e_3) = ((n_{11}, e''_3, n_{12}), e'_2, n_2)$ with $(n_{11}, e''_3, n_{12}) = type_{E_G}(s_{G_{EA}}(e_3)) \Rightarrow$
$\overline{source_{EA}} \circ type_{E_{EA}}(e_3) = \overline{source_{EA}}[((n_{11}, e''_3, n_{12}), e'_3, n_2)] = \overline{(n_{11}, e''_3, n_{12}) = type_{E_G}(s_{G_{EA}}(e_3))}$

(vii) $type_{V_D} \circ t_{G_{EA}}(e_3) = \overline{target_{EA}} \circ type_{E_{EA}}(e_3) \quad \forall e_3 \in G_{E_{EA}}$
By definition of $type_{E_{EA}}$ we have
$type_{E_{EA}}(e_3) = ((n_{11}, e''_3, n_{12}), e'_3, n_2)$ with $n_2 = type_{V_D}(t_{G_{EA}}(e_3)) \Rightarrow$
$\overline{target_{EA}} \circ type_{E_{EA}}(e_3) = \overline{target_{EA}}[((n_{11}, e''_3, n_{12}), e'_3, n_2)] = n_2 = \underline{type_{V_D}(t_{G_{EA}}(e_3))}$

(2) We have to show $u_{ATG} \circ type = type'$



(a) $u_{ATG,V_G} \circ type_{V_G} = type_{V_G} = type'_{V_G}$
(b) $u_{ATG,V_D} \circ type_{V_D} = type_{V_D} = type'_{V_D}$
(c) for $type_{E_G}(e_1) = (n_1, e'_1, n_2) \in \overline{TG_{E_G}}$ with $e'_1 = type'_{E_G}(e_1)$ we have
$u_{ATG,E_G} \circ type_{E_G}(e_1) = u_{ATG,E_G}[(n_1, e'_1, n_2)] = e'_1 = type'_{E_G}(e_1)$
(d) for $type_{E_{NA}}(e_2) = (n_1, e'_2, n_2) \in \overline{TG_{E_{NA}}}$ with $e'_2 = type'_{E_{NA}}(e_2)$ we have
$u_{ATG,E_{NA}} \circ type_{E_{NA}}(e_2) = u_{ATG,E_{NA}}[(n_1, e'_2, n_2)] = e'_2 = type'_{E_{NA}}(e_2)$
(e) for $type_{E_{EA}}(e_3) = ((n_{11}, e''_3, n_{12}), e'_3, n_2) \in \overline{TG_{E_{EA}}}$ with $e'_3 = type'_{E_{EA}}(e_3)$ we have
$u_{ATG,E_{EA}} \circ type_{E_{EA}}(e_3) = u_{ATG,E_{EA}}[(n_{11}, e''_3, n_{12})] = e'_3 = type'_{E_{EA}}(e_3)$
(f) $u_{ATG,D} \circ type_D = type_D = type'_D$

(3) Given AG-morphism $f : AG \to \overline{ATG}$ with $u_{ATG} \circ f = type'$
we have to show $f = type$, which will be shown in (a)–(f) below

(a) $f_{V_G}(n_1) = u_{ATG,V_G} \circ f_{V_G}(n_1) = type'_{V_G}(n_1) = type_{V_G}(n_1) \Rightarrow$
$\underline{f_{V_G} = type_{V_G}}$

(b) $f_{V_D}(n_2) = u_{ATG,V_D} \circ f_{V_D}(n_2) = type'_{V_D}(n_2) = type_{V_D}(n_2) \Rightarrow$
$\underline{f_{V_D} = type_{V_D}}$

(c) Let $f_{E_G}(e_1) = (n_1, e'_1, n_2) \in \overline{TG_{E_G}}$. Now $type'_E = u_{ATG} \circ f$ implies
$\underline{type'_{E_G}(e_1) = u_{ATG,E_G} \circ f_{E_G}(e_1) = u_{ATG,E_G}[(n_1, e'_1, n_2)] = e'_1}$
f AG-morphism implies:
$f_{V_G} \circ s_{G_G}(e_1) = \overline{source_G} \circ f_{E_G}(e_1) = \overline{source_G}[(n_1, e'_1, n_2)] = n_1$
$f_{V_G} \circ t_{G_G}(e_1) = \overline{target_G} \circ f_{E_G}(e_1) = \overline{target_G}[(n_1, e'_1, n_2)] = n_2$
$\Rightarrow \underline{n_1} = f_{V_G} \circ s_{G_G}(e_1) \overset{(a)}{=} \underline{type_{V_G}(s_{G_G}(e_1))}$
$\underline{n_2} = f_{V_G} \circ t_{G_G}(e_1) \overset{(a)}{=} \underline{type_{V_G}(t_{G_G}(e_1))}$
$\Rightarrow f_{E_G}(e_1) = type_{E_G}(e_1)$ by definition of $type_{E_G} \rightarrow \underline{f_{E_G} = type_{E_G}}$

(d) Let $f_{E_{NA}}(e_2) = (n_1, e'_2, n_2) \in \overline{TG_{E_{NA}}}$ for $e'_2 \in TG_{E_{NA}}$ with $n_2 = target_{NA}(e'_2)$.
Now $type' = u_{ATG} \circ f$ implies
$\underline{type'_{E_{NA}}(e_2) = u_{ATG,E_{NA}} \circ f_{E_{NA}}(e_2) = u_{ATG,E_{NA}}[(n_1, e'_2, n_2)] = e'_2}$
f AG-morphism implies:
$f_{V_G} \circ s_{G_{NA}}(e_2) = \overline{source_G} \circ f_{E_{NA}}(e_2) = \overline{source_G}[(n_1, e'_2, n_2)] = n_1$
$f_{V_D} \circ t_{G_{NA}}(e_2) = \overline{target_G} \circ f_{E_{NA}}(e_2) = \overline{target_G}[(n_1, e'_2, n_2)] = n_2$
$\Rightarrow \underline{n_1} = f_{V_G} \circ s_{G_{NA}}(e_2) \overset{(a)}{=} \underline{type_{V_G}(s_{G_{NA}}(e_2))}$
$\underline{n_2} = f_{V_D} \circ t_{G_{NA}}(e_2) \overset{(b)}{=} \underline{type_{V_D}(t_{G_{NA}}(e_2))}$
$\Rightarrow \underline{f_{E_{NA}} = type_{E_{NA}}}$

(e) Let $f_{E_{EA}}(e_3) = ((n_{11}, e''_3, n_{12}), e'_3, nG_2) \in \overline{TG_{E_{EA}}}$.
Now $type' = u_{ATG} \circ f$ implies
$\underline{type'_{E_{EA}}(e_3) = u_{ATG,E_{EA}} \circ f_{E_{EA}}(e_3) = u_{ATG,E_{EA}}[((n_{11}, e''_3, n_{12}), e'_3, n_2)] = e'_3}$
f AG-morphism implies:
$f_{E_G} \circ s_{G_{EA}}(e_3) = \overline{source_{EA}} \circ f_{E_{EA}}(e_3) = \overline{source_{EA}}[((n_{11}, e''_3, n_{12}), e'_3, n_2)]$
$= (n_{11}, e''_3, n_{12})$
$f_{V_D} \circ t_{G_{EA}}(e_3) = \overline{target_{EA}} \circ f_{E_{EA}}(e_3) = \overline{target_{EA}}[((n_{11}, e''_3, n_{12}), e'_3, n_2)]$
$= n_2$
$\Rightarrow \underline{(n_{11}, e''_3, n_{12})} = f_{E_G} \circ s_{G_{EA}}(e_3) \overset{(c)}{=} \underline{type_{E_G}(s_{G_{EA}}(e_3))}$
$\underline{n_2} = f_{V_D} \circ t_{G_{EA}}(e_3) \overset{(b)}{=} \underline{type_{V_D}(t_{G_{EA}}(e_3))}$
$\Rightarrow \underline{f_{E_{EA}} = type_{E_{EA}}}$

(f) $type' = u_{ATG} \circ f$ implies $type'_D = u_{ATG,D} \circ f_D = f_D \rightarrow \underline{f_D = type_D}$

Proof 2 (*Theorem* 2).

"1 ⇒ 2" This follows directly from Lemma 3.

"2 ⇒ 1" If m is a consistent match w.r.t. p_t and $(G, type_G)$ with $t_L = type_G \circ m$ we have $t_L = type_G \circ m \leq type_L$. For $x_1, x_2 \in K_{V_G}$ with $r_{V_G}(x_1) = r_{V_G}(x_2)$ it follows that $t_{K,V_G}(x_1) = t_{R,V_G} \circ r_{V_G}(x_1) = t_{R,V_G} \circ r_{V_G}(x_2) = t_{L,V_G}(x_2)$. Match m satisfies $\overline{NAC}$, i.e. $\forall (N, n, t_N) \in \overline{NAC}$, there is no morphism $o \in \mathcal{M}'$ with $o \circ n = m$ and $type_G \circ o = t_N$. It follows that m also satisfies NAC. Otherwise, there would exist $nac = (N, n, type_N) \in NAC$, $o \in \mathcal{M}'$ with $o \circ n = m$ and $type_G \circ o \leq type_N$. This would contradict that m satisfies $\overline{nac} = (N, n, t_N)$ with $t_N = type_G \circ o \leq type_N$. That means, m is a consistent match w.r.t. p and $(G, type_G)$.

Now we apply Lemma 3, where the induced concrete production in Item 1 coincides with the given on, and obtain the abstract direct transformation $(G, type_G) \overset{p,m}{\Longrightarrow} (H, type_H)$.

Proof 3 (*Theorem* 3).

(1) With Theorem 2 the abstract direct transformation $(G_1, type_{G_1}) \overset{p,m}{\Longrightarrow} (G_2, type_{G_2})$ and the concrete direct transformation $(G_1, type_{G_1}) \overset{p_t,m}{\Longrightarrow} (G_2, type_{G_2})$ with $t_L = type_{G_1} \circ m$ are equivalent and if one exists, so does the other one. That means if $(G_1, type_{G_1}) \in L(GG) \cap L(\widehat{GG})$ then $(G_2, type_{G_2}) \in L(GG) \cap L(\widehat{GG})$. Since we start in both grammars with the same start graph, $L(GG) = L(\widehat{GG})$.

(2) We show, that

(a) for a concrete direct transformation $(G_1, type_{G_1}) \overset{p_t,m}{\Longrightarrow} (G_2, type_{G_2})$ in $\widehat{GG}$ there is a corresponding direct transformation $(G_1, \overline{type_{G_1}}) \overset{\overline{p_t},m}{\Longrightarrow} (G_2, \overline{type_{G_2}})$ in $\overline{GG}$ with $u_{ATG} \circ \overline{type_{G_i}} = type_{G_i}$ for $i = 1, 2$ and

(b) if a production $\overline{p_t}$ can be applied to $(G_1, \overline{type_{G_1}})$ via m in $\overline{GG}$ then p_t can be applied to $(G_1, u_{ATG} \circ \overline{type_{G_1}})$ via m in $\widehat{GG}$.

(a) For all objects $(X, type_X)$ in the DPO diagram corresponding to the concrete direct transformation $(G_1, type_{G_1}) \overset{p_t,m}{\Longrightarrow} (G_2, type_{G_2})$ Theorem 1 gives us a morphism $\overline{type_X} : X \to \overline{ATG}$. The DPO diagram with these new morphisms corresponds to the direct transformation $(G_1, \overline{type_{G_1}}) \overset{\overline{p_t},m}{\Longrightarrow} (G_2, \overline{type_{G_2}})$ in $\overline{GG}$.

It remains to show that $\overline{p_t}$ can by applied to G_1 via m, i.e. m satisfies the negative application condition $\overline{NAC}'$. Suppose not, and we have a negative application condition $(N, n, \overline{t_N}) \in \overline{NAC}'$, that is not satisfied by m and corresponds to $(N, n, t_N) \in \overline{NAC}$ with $u_{ATG} \circ \overline{t_N} = t_N$. Then there is a morphism $o : N \to G_1$ with $o \circ n = m$ and since o is a typed attributed graph morphism $\overline{type_{G_1}} \circ o = \overline{t_N}$. Then $type_{G_1} \circ o = u_{ATG} \circ \overline{type_{G_1}} \circ o = u_{ATG} \circ \overline{t_N} = t_N$. According to Definition 17 that means m does not satisfy $\overline{NAC}$, which is a contradiction.

(b) The application of $\overline{p_t}$ to G_1 via m leads to a direct transformation $(G_1, \overline{type_{G_1}}) \overset{\overline{p_t},m}{\Longrightarrow} (G_2, \overline{type_{G_2}})$. For all objects $(X, type_X)$ in the corresponding DPO diagram we define $type_X = u_{ATG} \circ \overline{type_X}$ and get a new DPO diagram corresponding to the concrete direct transformation $(G_1, type_{G_1}) \overset{p_t,m}{\Longrightarrow} (G_2, type_{G_2})$.

We have to check that m satisfies $\overline{NAC}$. Suppose not, then there is a negative application condition $(N, n, t_N) \in \overline{NAC}$ and an AG-morphism $o : N \to G$ such that $o \circ n = m$ and $type_{G_1} \circ o = t_N$. Then the negative application condition $(N, n, \overline{t_N}) \in \overline{NAC}'$ with $t_N = type_{G_1} \circ o$ is not satisfied by m. This is a contradiction.

For an concrete transformation $(G, type_G) \overset{*}{\Rightarrow} (H, type_H)$ in $\widehat{GG}$ item (a) gives us the corresponding transformation $(G, \overline{type_G}) \overset{*}{\Rightarrow} (H, \overline{type_H})$ in $\overline{GG}$. Item (b) guarantees, that for a transformation $(G, \overline{type_G}) \overset{*}{\Rightarrow} (H, \overline{type_H})$ in $\overline{GG}$ there is a corresponding concrete transformation $(G, type_G) \overset{*}{\Rightarrow} (H, type_H)$ in $\widehat{GG}$. Combining (a) and (b) we have $L(\widehat{GG}) \cong L(\overline{GG})$. By part 1 we have $L(GG) = L(\widehat{GG})$, which implies $L(GG) \cong L(\overline{GG})$ as required.

References

[1] A. Agrawal, G. Karsai, F. Shi, Graph transformations on domain-specific models, Technical Report ISIS-03-403 of the Institute for Software Integrated Systems, Vanderbilt University, November, 2003.

[2] R. Bardohl, H. Ehrig, J. de Lara, G. Taentzer, Integrating meta modelling aspects with graph transformation for efficient visual language definition and model manipulation, in: Proc. FASE'04, in: LNCS, vol. 2984, Springer, 2004, pp. 214–228.

[3] R. Bardohl, A visual environment for visual languages, Science of Computer Programming 44 (2002) 181–203.

[4] R. Bardohl, G. Taentzer, M. Minas, A. Schürr, Application of graph transformation to visual languages, in: H. Ehrig, G. Engels, H.-J. Kreowski, G. Rozenberg (Eds.), Handbook of Graph Grammars and Computing by Graph Transformation, Volume 2, World Scientific, 1999, pp. 105–181.

[5] L. Baresi, M. Pezze, A toolbox for automating visual software engineering, in: Proc. FASE'02, in: LNCS, vol. 2306, Springer, 2002, pp. 189–202.

[6] G. Booch, Object Oriented Design, Benjamin-Cummings, 1991.

[7] P. Bottoni, G. Taentzer, A. Schürr, Efficient parsing of visual languages based on critical pair analysis and contextual layered graph transformation, in: Proc. IEEE International Symposium on Visual Languages, VL'00, 2000, pp. 59–60.

[8] A. Corradini, U. Montanari, F. Rossi, Graph processes, Fundamenta Informaticae 26 (3–4) (1996) 241–265.

[9] H. Ehrig, K. Ehrig, U. Prange, G. Taentzer, Fundamentals of Algebraic Graph Transformation, in: Monographs in Theoretical Computer Science, Springer, 2006.

[10] H. Ehrig, K. Ehrig, U. Prange, G. Taentzer, Formal integration of inheritance with typed attributed graph transformation for efficient vl definition and model manipulation, in: Proc. 2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), Dallas, USA, 2005, pp. 71–78.
[11] H. Ehrig, K. Ehrig, J. de Lara, T. Taentzer, D. Varró, S. Varró-Gyapay, Termination criteria for model transformation, in: Proc. FASE'05, in: LNCS, vol. 3442, Springer, 2005, pp. 49–63.
[12] H. Ehrig, U. Prange, G. Taentzer, Fundamental theory for typed attributed graph transformation, in: Proc. ICGT'04, in: LNCS, vol. 3256, Springer, 2004, pp. 161–177.
[13] H. Ehrig, K. Ehrig, A. Habel, K.-H. Pennemann, Constraints and application conditions: From graphs to high-level structures, in: Proc. ICGT'04, in: LNCS, vol. 3256, Springer, 2004, pp. 287–303.
[14] H. Ehrig, G. Engels, H.-J. Kreowski, G. Rozenberg, Handbook of Graph Grammars and Computing by Graph Transformation. Vol 1. Foundations, World Scientific, 1999.
[15] H. Ehrig, B. Mahr, Fundamentals of Algebraic Specifications 1: Equations and Initial Semantics, in: EATCS Monographs on Theoretical Computer Science, vol. 6, Springer, 1985.
[16] T. Fischer, J. Niere, L. Torunski, A. Zuendorf, Story diagrams: A new graph rewrite language based on the unified modeling language, in: Proc. TAGT'98, in: LNCS, vol. 1764, Springer, 1998, pp. 296–309. See also the Fujaba home page http://www.fujaba.de.
[17] H. Heckel, J. Küster, G. Taentzer, Towards automatic translation of UML models into semantic domains, in: Proc. AGT 2002, 2002, pp. 11–22.
[18] R. Heckel, J. Küster, G. Taentzer, Confluence of typed attributed graph transformation systems, in: Proc. ICGT'02, in: LNCS, vol. 2505, Springer, 2002, pp. 161–176.
[19] H. Heckel, A. Wagner, Ensuring Consistency of Conditional Graph Grammars — A Constructive Approach, in: ENTCS, vol. 2, Elsevier, 1995.
[20] J. de Lara, R. Bardohl, H. Ehrig, K. Ehrig, U. Prange, G. Taentzer, Attributed graph transformation with node type inheritance: Long version. Technical Report 2005/3, TU Berlin, 2005.
[21] J. de Lara, H. Vangheluwe, Defining visual notations and their manipulation through meta-modelling and graph transformation, in: Domain-Specific Modeling with Visual Languages, Journal of Visual Languages and Computing 15 (3–4) (2004) 309–330 (special issue).
[22] J. de Lara, H. Vangheluwe, AToM^3: A tool for multi-formalism modelling and meta-modelling, in: Proc. FASE'02, in: LNCS, vol. 2306, Springer, 2002, pp. 174–188. See also the AToM^3 home page http://atom3.cs.mcgill.ca.
[23] A.P. Lüdtke, L. Ribeiro, Derivations in object oriented grammars, in: Proc. ICGT'04, in: LNCS, vol. 3256, Springer, 2004, pp. 416–430.
[24] K. Marriot, B. Meyer, Visual Language Theory, Springer, 1998.
[25] T. Mens, S. Demeyer, D. Janssens, Formalising behaviour preserving program transformations, in: Proc. ICGT'02, in: LNCS, vol. 2505, Springer, 2002, pp. 286–301.
[26] MDA, MOF and UML specifications at the OMG web page: http://www.omg.org.
[27] A. Schürr, Introduction to PROGRES, an attribute graph grammar based specification language, in: Proc. WG89, in: LNCS, vol. 411, Springer, 1990, pp. 151–165.
[28] A. Schürr, Programmed graph replacement systems, in: [14], 1999, pp. 479–546.
[29] J. Sprinkle, G. Karsai, A domain-specific visual language for domain model evolution, Journal of Visual Languages and Computing 15 (3–4) (2004) 291–307.
[30] G. Taentzer, A. Rensink, Ensuring structural constraints in graph-based models with type inheritance, in: Proc. FASE'05, in: LNCS, vol. 2984, Springer, 2005, pp. 64–79.
[31] G. Taentzer, AGG: A graph transformation environment for modeling and validation of software, in: Proc. AGTIVE'03, in: LNCS, vol. 3062, Springer, 2003, pp. 446–453. See also the AGG home page http://tfs.cs.tu-berlin.de/agg/.
[32] D. Varro, A. Pataricza, Generic and meta-transformations for model transformation engineering, in: Proc. UML'04, in: LNCS, vol. 3273, Springer, 2004, pp. 290–304.
[33] D. Varro, A. Pataricza, VPM: A visual, precise and multilevel metamodeling framework for describing mathematical domains and UML (The Mathematics of Metamodeling is Metamodeling Mathematics), Software and Systems Modeling 2 (3) (2003) 187–210.
[34] D. Varro, A formal semantics of UML Statecharts by model transition systems, in: Proc. ICGT'02, in: LNCS, vol. 2505, Springer, 2002, pp. 378–392.