Formal Integration of Inheritance with Typed Attributed Graph Transformation for Efficient VL Definition and Model Manipulation

Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer,
Technical University Berlin, Germany
Email: {ehrig,karstene,uprange,gabi}@cs.tu-berlin.de

Abstract

Several approaches exist to define a visual language (VL). Among those the meta-modeling approach used to define the Unified Modeling Language (UML), and the graph transformation approach are very popular. Especially the combination of both, using meta-modeling to define the syntax of a VL and graph transformation for specifying model transformations has been considered conceptually and explored in a number of applications. A formal integration of both approaches has just been started by integrating classical algebraic graph grammars with a node type inheritance concept. In this paper, the integration of inheritance is extending to attributed graph transformation. More precisely, we define attributed type graphs with inheritance leading to a formal integration of inheritance with typed attributed graph transformation.

1 Introduction

There are mainly two different lines to define a visual language (VL): the declarative and the constructive way. The UML is defined by the Meta Object Facilities (MOF) approach [12], which uses classes and associations to define symbols and relations of a VL. Within this meta modeling approach, multiplicities and OCL constraints [14] are additionally used to formulate desired language properties. While constraint-based formalisms provide a declarative approach to VL definitions well-suited to specify language requirements, grammars are more constructive, i.e. closer to the implementation. In [11], a number of textual as well as graph grammar approaches are considered for VL definition. Due to its appealing visual form, graph grammars can directly be used as high-level visual specification mechanisms for VLs [2]. Defining the abstract syntax of visual forms as graphs, a graph grammar directly defines the language grammar. The induced graph language determines the corresponding VL. Visual language parsers can be immediately deduced from such a graph grammar. Furthermore, abstract syntax graphs are also the starting point for model simulation and transformation, i.e. model manipulation [3, 5, 13, 9]. Here again, it is very natural to use graph transformation as a high-level, constructive specification formalism.

In [1] we have already started to consider the integration of meta modeling with graph transformation. The basic integration can be done by identifying symbol classes with node types and associations with edge types. In this way, declarative as well as constructive elements may be used for language definition, but it is still open how single parts of a VL specification are defined.

In addition, symbol classes can be inherited within the meta-modeling approach. To continue the integration of meta-modeling with graph transformation, the concept of class inheritance has to be transferred to node types. Supporting node type inheritance leads to a more dense form of graph transformation systems, since similar productions can be abstracted into one. In [1], we have shown this integration for typed graph transformation without attributes. Since classes usually also have attributes, the integration of inheritance should be extended to typed attributed graph transformation. A solid formal framework for typed attributed graph transformation with attributes for nodes and edges has been presented in [7]. In this paper, we show how to extend this approach to a formal integration of inheritance with typed attributed graph transformation. For this purpose, we first introduce attributed type graphs with inheritance and show how they can be flattened to attributed type graphs without inheritance. This flattening idea is continued for typed attributed graph transformation.

The main results in this paper show that for each graph transformation and grammar GG based on an attributed type graph $ATGI$ with inheritance there is an equivalent typed attributed graph transformation and grammar $\overline{GG}$ without inheritance. Hence there is a direct correspondence to typed attributed graph transformation without inheritance, where fundamental theoretical results have already been shown in [7].

Applying typed attributed graph transformation with in-

Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05)
0-7695-2443-5/05 $20.00 © 2005 **IEEE**

heritance to VL definitions makes the integrated usage of meta-modeling and graph transformation concepts more precise and clear on one hand, and allows the application of theoretical results to VL definitions on the other hand. E.g. using typed attributed graph transformation with inheritance for model transformation opens the possibility to show certain correctness properties for model transformation.

In this paper we focus on the formal framework concerning typed attributed graph transformation with inheritance and a small example only. A slightly larger example is presented in [1] defining a simplified variant of UML state diagrams.

2 Attributed Type Graphs with Inheritance

In this section we start to integrate the concept of inheritance into typed graphs, which results in a general concept for *Attributed Type Graphs with Inheritance (ATGI)* and a close relationship of typing with and without inheritance. In [7] we have presented typed attributed graphs with attributes for nodes and edges, which generalizes the concept in [8], where only attributes for nodes have been considered. The new approach in [7] requires the concepts of an *E-graph* where two kinds of vertices, i.e. graph and data vertices, are distinguished. Furthermore, three different kinds of edges are defined to link vertices or to refer to attributes.

Let G be an *E-graph* $G = (G_{V_G}, G_{V_D}, G_{E_G}, G_{E_{NA}}, G_{E_{EA}}, (source_i)_{i\in\{G,NA,EA\}}, (target_i)_{i\in\{G,NA,EA\}})$ where G refers to graph parts, NA to node attribution, and EA to edge attribution, according to the signature in Fig. 1.

An attributed graph AG over a data signature $DSIG = (S_D, OP_D)$ with attribute value sorts $S'_D \subseteq S_D$ is given by $AG = (G, D)$ where G is an *E-graph* as described above and D is a $DSIG$-algebra s.t. $\dot\cup_{s\in S'_D} D_s = G_{V_D}$.

2.1 Attributed Type Graphs with Inheritance

For a clearer correspondence between class concepts on one hand and type graphs on the other hand, we start to extend the existing concept of attributed type graphs by node type inheritance.

Definition 1 (Attributed Type Graph with Inheritance) *Let ATG be an attributed type graph (TG, Z) with TG being 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 being the final $DSIG$-algebra. An* attributed type graph with inheritance (ATGI) *$ATGI = (TG, Z, I, A)$ consists of an attributed type graph ATG, 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 $clan_I(n) = \{n' \in I_V \mid \exists\ path\ n' \overset{*}{\rightarrow} n\ in\ I\} \subseteq I_V$ with $n \in clan_I(n)$.*

The inheritance graph I could be defined to be acyclic, but this is not necessary for our theory.

The running example of this paper is a very small section of a notational model for diagrams. We start with the presentation of the attributed type graph with inheritance. In Fig. 2 we show the compact and in Fig. 3 the explicit notation of the same sample type graph. Solid and dashed arrows represent the graph and attribution edges of the type graph, while solid arrows with white arrowhead belong to the inheritance graph. A *Screen* with a resolution *(width,height)* has geometrical *Figures* which have a position and a visibility *(x,y,visible)*. *Figures* are first abstractly defined. An inheritance relation to concrete figures refines them to *Circle, Rectangle,* and *Line*. A *Circle* has the additional attribute *radius*, a *Rectangle* the additional attributes *width and height* and a *Line* has the relative end point coordinates *(endx, endy)* as additional attributes. The edge attribute $id : Nat$ refers to the identities of figures and demonstrates the usage of edge attributes in attributed type graphs with inheritance. Edge attributes are also useful to establish a list of items, e.g. to enumerate parameters of methods or operations.

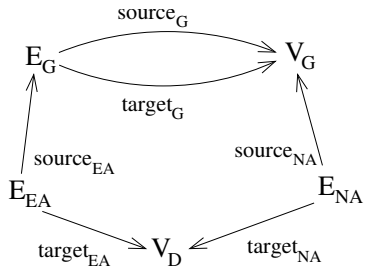


Figure 1. E-Graph

In order to benefit from the well-founded theory of typed attributed graph transformation [7], we flatten attributed type graphs with inheritance to ordinary ones.

Definition 2 (Closure of ATGI) *Given an attributed type graph with inheritance $ATGI = (TG, Z, I, A)$ with TG as above, the* abstract closure *of $ATGI$ is the attributed type graph $\overline{ATG} = (\overline{TG}, Z)$ with $\overline{TG} = (TG_{V_G}, TG_{V_D}, \overline{TG_{E_G}}, \overline{TG_{E_{NA}}}, \overline{TG_{E_{EA}}}, (\overline{source_i}, \overline{target_i})_{i\in\{G,NA,EA\}})$*

- $\overline{TG_{E_G}} = \{(n_1, e, n_2) \mid n_1 \in clan_I(source_G(e)), n_2 \in clan_I(target_G(e)), e \in TG_{E_G}\}$
- $\overline{source_G}((n_1, e, n_2)) = n_1 \in TG_{V_G}$
- $\overline{target_G}((n_1, e, n_2)) = n_2 \in TG_{V_G}$
- $\overline{TG_{E_{NA}}} = \{(n_1, e, n_2) \mid n_1 \in clan_I(source_{NA}(e)), n_2 = target_{NA}(e), e \in TG_{E_{NA}}\}$

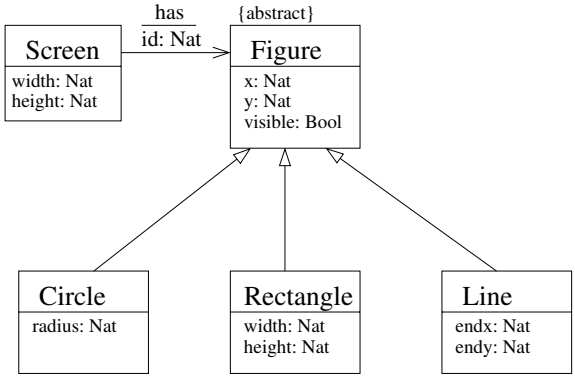


Figure 2. Example of an attributed type graph with inheritance (compact notation).

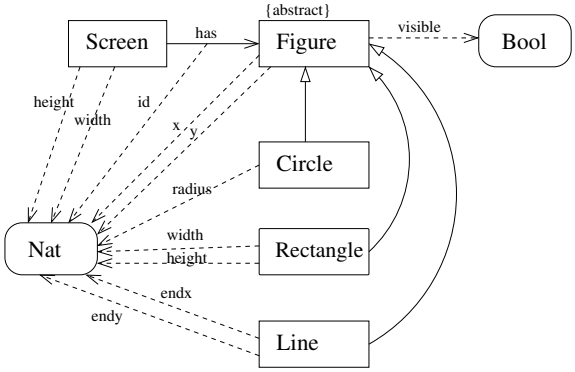


Figure 3. Example of an attributed type graph with inheritance (explicit notation).

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

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* $TG_{V_G} \setminus A$

The discrimination between the abstract and the concrete closure of a type graph with inheritance is necessary. The left-hand side (LHS) and right-hand side (RHS) of an abstract production considered in section 3 are typed over the abstract closure, while ordinary host graphs and concrete productions are typed over the concrete closure.

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. *The abstract and concrete closures of an* $ATGI$ *are attributed type graphs without inheritance.*

Instances of attributed type graphs with inheritance are attributed graphs. Here again, we can notice a direct correspondence to meta-modeling where models consisting of symbols and relations are instances of meta-models containing the correspondent classes and associations.

Fig. 4 shows the compact notation of the abstract and concrete closure of the ATGI example in Fig. 2, which follows from the explicit notation in Fig. 5. Here, we see that the attributes as well as the edges of the abstract type $Figure$ are flattened to its descendant types and the data vertex Nat is depicted twice for clearness.

Definition 3 (Instances of ATGI)
An abstract instance $(AG, type)$ *of* $ATGI$ *is an attributed graph of* $\overline{ATG}$*, i.e.* $(AG, type : AG \to \overline{ATG})$.
Similarly, a concrete instance $(AG, type)$ *of* $ATGI$ *is an attributed graph of* $\widehat{ATG}$*, i.e.* $(AG, type : AG \to \widehat{ATG})$.

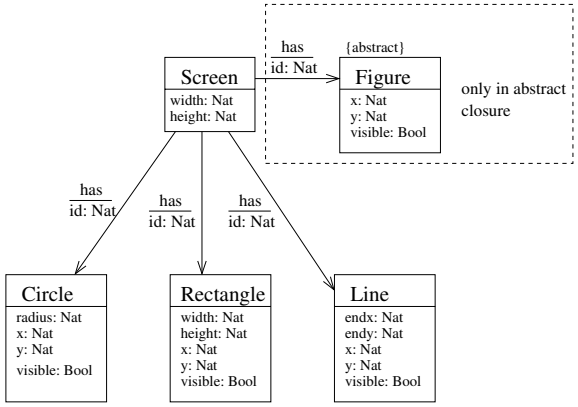


Figure 4. Abstract and concrete closure of the ATGI example in Fig. 2 (compact notation).

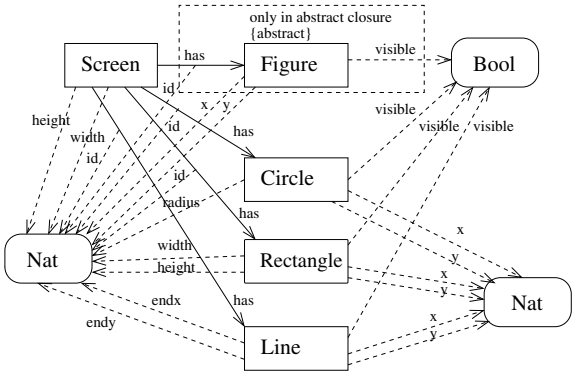


Figure 5. Abstract and concrete closure of the ATGI example in Fig. 3 (explicit notation).

2.2 Attributed Clan Morphisms

Instances of attributed type graphs with inheritance are attributed graphs. Here again, we can notice a direct correspondence to meta-modeling where models consisting of symbols and relations are instances of meta-models containing the correspondent classes and associations. To formally define the instance-type relation, we introduce attributed clan morphisms.

The instance graph is typed over the type graph with inheritance by a pair of functions, one assigning each vertex a vertex type and the other one assigning each edge an edge type. Both are defined canonically. An ordinary graph morphism [7] is not obtained this way, but some mapping called *clan morphism*, enough to uniquely characterizing the type morphism into the flattened type graph (see Lemma 1).

Definition 4 (ATGI-Clan Morphism)

Given an attributed type graph with inheritance $ATGI = (TG, Z, I, A)$ *with* $TG_{V_D} = S_D^{'}$ *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* $\dot{\cup}_{s\in S_D^{'}} D_s = G_{V_D}$ *then*
$type : AG \to ATGI$ *with* $type = (type_{V_G}, type_{V_D}, type_{E_G}, type_{E_{NA}}, type_{E_{EA}}, type_D)$ *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) $type_{V_G} \circ s_{G_G}(e_1) \in clan_I(src_G \circ type_{E_G}(e_1))$
(2) $type_{V_G} \circ t_{G_G}(e_1) \in clan_I(tar_G \circ type_{E_G}(e_1))$
(3) $type_{V_G} \circ s_{G_{NA}}(e_2) \in clan_I(src_{NA} \circ type_{E_{NA}}(e_2))$
(4) $type_{V_D} \circ t_{G_{NA}}(e_2) = tar_{NA} \circ type_{E_{NA}}(e_2)$
(5) $type_{E_G} \circ s_{G_{EA}}(e_3) = src_{EA} \circ type_{E_{EA}}(e_3)$
(6) $type_{V_D} \circ t_{G_{EA}}(e_3) = tar_{EA} \circ type_{E_{EA}}(e_3)$
$\forall e_1 \in G_{E_G}$, $\forall e_2 \in G_{E_{NA}}$, $\forall e_3 \in G_{E_{EA}}$,
where we use abbreviations 'src' and 'tar' for 'source' and 'target' respectively.

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

The following lemma is the key property relating ATGI-clan morphisms and AG-morphisms, which is essential to show the main results in section 3.

Lemma 1 (Universal ATGI-Clan Property)

There exists a universal ATGI-clan morphism u_{ATGI} *s.t. for each ATGI-clan morphism* $type : AG \to ATGI$ *there is a unique AG-morphism* $\overline{type} : AG \to \overline{ATG}$ *with* $u_{ATGI} \circ \overline{type} = type$.

$$\begin{array}{ccccc} & & AG & & \\ & \overline{type} \swarrow & = & \searrow type & \\ \overline{ATG} & & \xrightarrow[u_{ATGI}]{} & & ATGI \end{array}$$

Proof: See technical report [6].

3 Typed Attributed Graph Transformation with Inheritance

In this section, we show how to adapt the concept of inheritance to the concepts of typed attributed graph transformation, graph grammar and graph language. The general approach is to describe graph transformations by graph productions. We follow the *Double Pushout* approach of typed attributed graph transformation [7] extended by negative application consitions (NACs).

As in well-formedness rules, e.g. formulated in the Object Constraint Language (OCL) [14], the usage of abstract classes is helpful to formulate concise language properties,

the usage of abstract types in graph transformation is helpful to formulate concise graph productions.

3.1 Productions and Transformations

Our goal is to allow abstractly typed nodes in productions, such that these abstract productions actually represent a set of structurally similar productions which we call *concrete productions*. To get 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 have to have a concrete type, since abstract types should not be instantiated.

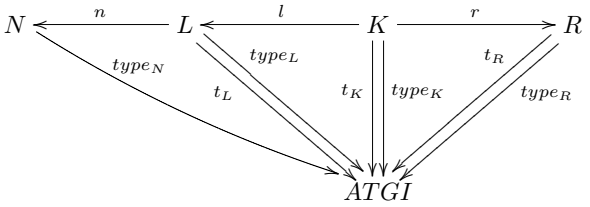
As done for type graphs with inheritance we define a flattening of abstract productions to concrete ones. Concrete productions are structurally equal to the abstract production, but their typing morphisms are finer than the ones of the abstract production and are concrete clan morphisms. A typing morphism 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 5 (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*

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

Definition 6 (Abstract and Concrete Productions)
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 ATGI-clan morphisms $type = (type_L : L \to ATGI, type_K : K \to ATGI, type_R : R \to ATGI)$ and NAC is a set of triples $nac = (N, n, type_N)$ with an attributed graph N, an injective 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* $R'_{V_G} := R_{V_G} - r_{V_G}(K_{V_G})$
- $type_N \circ n \leq type_L \quad$ *for all* $(N, n, type_N) \in NAC$
- *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 $p_t = (L \xleftarrow{l} K \xrightarrow{r} R, t, NAC)$, 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.*

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

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.

Definition 7 (Application of Concrete Productions)
Let $p_t = (L \xleftarrow{l} K \xrightarrow{r} R, t, NAC)$ 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.
m is a match *with respect to p_t and $(G, type_G)$, if*

- *m is a match with respect to 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 NAC, i.e. for each $(N, n, type_N) \in NAC$ it holds, that there exists no AG-morphism $o : N \to G$, such that $o \circ n = m$ and $type_G \circ o \leq type_N$.*

Given a 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 (see Fig. 6). We write $(G, type_G) \overset{p_t, m}{\Longrightarrow} (H, type_H)$ for such a direct concrete transformation. *In general, a concrete transformation $(G, type_G) \Longrightarrow (H, type_H)$ is either direct or a concatenation of two concrete transformations $(G, type_G) \Longrightarrow (K, type_K) \Longrightarrow (H, type_H)$.*

The classical theory of typed attributed graph transformations relies on typing morphisms which are normal graph

morphism, 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 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 [6] for productions without NAC's.

Although the semantics for the application of abstract production can be given by the application of its concrete productions, this solution is not efficient at all. Imagine a tool which implements graph transformation with node type inheritance, it would have to check all concrete productions of an abstract productions to find the right one to apply to a given instance graph.

Thus, as a next step, we want to examine more direct ways 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 8 (Application of Abstract Productions)
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.
m is a match *with respect to p and $(G, type_G)$, if*

- *m is a match with respect to the untyped production $L \xleftarrow{l} K \xrightarrow{r} R$ and the attributed graph G,*
- *$type_G \circ m \leq type_L$.*
- *$t_{K,V_G}(x_1) = t_{K,V_G}(x_2)$ 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 $o : N \to G$ such that $o \circ n = m$ and $type_G \circ o \leq type_N$.*

Given a 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 given by pushouts (1) and (2) in Figure 6.*
2. *Construct $type_D$ and $type_H$ as follows*
   - *$type_D = type_G \circ l'$*
   - *$type_{H,X}(x) =$ if $x = r'_X(x')$ then $type_{D,X}(x')$ else $type_{R,X}(x'')$, where $m'(x'') = x$ and $X \in \{V_G, V_D, E_G, E_{NA}, E_{EA}, D\}$.*

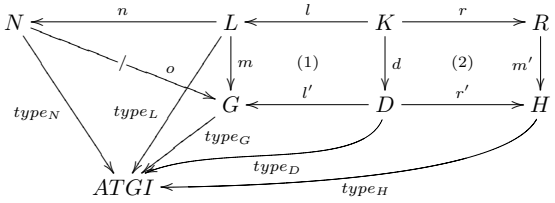


Figure 6. Match and application of an abstract production.

In general, an abstract transformation $(G, type_G) \Longrightarrow (H, type_H)$ is either direct or a concatenation of two abstract transformations $(G, type_G) \Longrightarrow (K, type_K) \Longrightarrow (H, type_H)$.

Remark 2 *$type_H$ 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 $type_D \circ d \leq type_K$. The third match condition is not needed if r_{V_G} is injective (as it is the case in most of the examples).*

3.2 Sample Productions and Transformations

Fig. 7 shows sample productions for the simple type graph in Fig. 2. Production $moveFigure(dx, dy : Nat)$ is an example of an abstract production, e.g. the production $moveFigure$ has to be defined only once and can be applied to concrete graphical objects of types $Circle$, $Rectangle$ and $Line$. This abstract production has three concrete productions according to the clan of type $Figure$. Please note that due to the positive values of dx and dy, the figures can only be moved up and right.

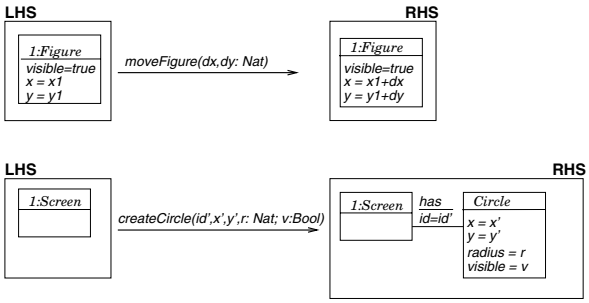


Figure 7. Example productions for ATGI example in Fig. 2.

$createCircle(id', x', y', r : Nat; v : Bool)$ is an example of a concrete production which creates a $Circle$ object. Note that the production has to take care of the abstract attributes x, y, $visible$ derived from abstract class $Figure$

as well as of the concrete attribute $radius$ and the edge attribute id. A rule $createFigure$ is not possible, because instances of abstract classes cannot be created.

Fig. 8 shows a sample derivation sequence for $createCircle$ and $moveFigure$ starting with an empty $Screen$ with resolution $width = 100$ and $height = 100$. First a new circle is created at position $(10, 10)$ applying production $createCircle$. Then, production $moveFigure(50, 0)$ instantiates attributes $x1$ and $y1$ with values $x1 = 10$ and $y1 = 10$. After calculating the results $x = x1 + dx$ and $y = y1 + dy$ using input parameters $dx = 50$ and $dy = 0$, the attributes x and y are assigned to new values. Note that the abstract production $moveFigure$ can be directly applied to the instance of concrete class $Circle$ derived from the abstract class $Figure$.

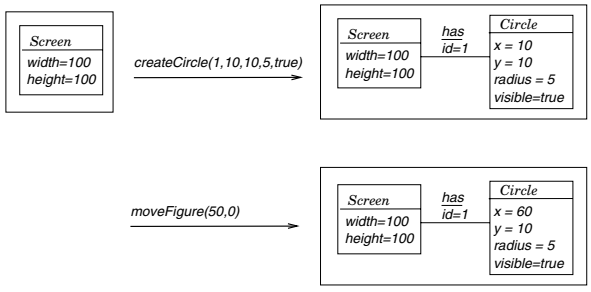


Figure 8. Sample derivation sequence.

3.3 Main Results

In this section we show the main results for the formal integration of inheritance with typed attributed graph transformation.

After having defined concrete and abstract transformations, the question arises how these two kinds of graph transformation are related to each other. Theorem 1 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.

Theorem 1 (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 structural match morphism $m : L \to G$ (i.e. a match with respect to 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 match with respect to 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 match with respect to the concrete production $p_t = L \leftarrow K \to R$ with $p_t \in \widehat{p}$ and $t_L = type_G \circ m$ yielding a concrete direct transformation $(G, type_G) \overset{p_t,m}{\Longrightarrow} (H, type_H)$.*
   In this case, t_K and t_R are defined by:
   - *$t_K = t_L \circ l$*
   - *$t_{R,V_G}(x) =$ <u>if</u> $x = r_{V_G}(x')$ <u>then</u> $t_{K,V_G}(x')$ <u>else</u> $type_{R,V_G}(x)$ for $x \in R_{V_G}$*
   - *$t_{R,X} = type_{R,X}$ for $X \in \{V_D, E_G, E_{NA}, E_{EA}, D\}$*

Proof: See technical report [6].

As a consequence of Theorem 1, we lift this result to graph grammars and graph languages below. Graph grammars have been used to define the syntax of visual languages. Different approaches exist, e.g. the hyperedge replacement which uses non-terminal labels to control the transformation process. The algebraic approach presented in this paper does not distinguish non-terminal and terminal types (labels), but controls the transformation process just by causal dependencies within the host graph. Please note that non-terminal and abstract types are two different features which can be combined in one approach, even though not done in this paper. A similar result can also be shown for graph transformation systems (graph grammars without start graph), w.r.t. a fixed set of input graphs.

Definition 9 (Graph Grammar and Language) *Given an attributed type graph $ATGI$ with inheritance and an attributed graph G typed over $ATGI$ by 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 typed over $ATGI$.*

The corresponding graph language *is defined by the set of all concrete typed graphs which are generated by an abstract transformation (cf. definitions 7 and 8):*
$L(GG) = \{(H, type_H : H \to ATGI) \mid \exists\ abstract\ transformation\ (G, type_G) \overset{*}{\Rightarrow} (H, type_H)\ and\ type_H\ is\ concrete\}$.

Theorem 2 (Equivalence of Graph Grammars)
For each ATGI-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 $\overline{ATG}$ graph grammar without inheritance $\overline{GG} = (\overline{ATG}, (G, \overline{type_G}), \overline{P})$ with closure $\overline{ATG}$ of $ATGI$ and productions $\overline{P}$, i.e. $L(GG) \stackrel{\sim}{=} L(\overline{GG})$, 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.
2. $\overline{type_G} : G \rightarrow \overline{ATG}$ is the graph morphism corresponding to the ATGI-clan morphism $type_G$ (see Lemma 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 $p_t = (p, t, NAC)$ we define $\overline{p_t} = (p, \overline{t}, \overline{NAC})$ with $u_{ATGI} \circ \overline{t_X} = t_X$ for $X \in \{L, K, R\}$ and $\overline{NAC}$ is defined by NAC as follows:
   For each $(N, n, type_N) \in NAC$ we have all $(N, n, \overline{t_N}) \in \overline{NAC}$ with $type_N \geq t_N = u_{ATGI} \circ \overline{t_N}$.

   Proof: See technical report [6].

Remark 3 *In the grammar $\overline{GG}$ of part 2 using the abstract closure $\overline{ATG}$ of $ATGI$ only graphs with concrete typing are generated. In fact there is also an equivalent grammar $\overline{GG'}$ with type graph $\widehat{ATG}$, the concrete closure of $ATGI$.*

4 Conclusion

In this paper we have presented a formal integration of inheritance with typed attributed graph transformation. The new concept allows the definition of abstract productions, in which abstractly typed nodes can appear. These can be matched to nodes of any of its concrete subtypes. This inheritance concept is extremely useful in applications as graph grammars and graph transformation systems can be notably more compact. This has already been demonstrated in [1] by showing the generation and simulation productions for a simplified variant of UML state diagrams. However, the formalism in [1] was restricted to graph transformation without an attribution concept. In this paper, we have shown how to obtain a formal integration of an inheritance concept with typed attributed graph transformation as presented in [7]. This work is a crucial step towards a precise integration of meta-modeling and graph transformation concepts.

It remains to lift analysis techniques such as constraint checking [10] and critical pair analysis [9] to type graphs with inheritance, useful to e.g. optimise visual language parsers [4] and to show correctness of model transformation [8].

References

[1] R. Bardohl, H. Ehrig, J. de Lara, and G. Taentzer. Integrating meta modelling aspects with graph transformation for efficient visual language definition and model manipulation. In M. Wermelinger and T. Margaria-Steffens, editors, *Proc. Fundamental Aspects of Software Engineering 2004*. Springer LNCS 2984, 2004.

[2] R. Bardohl, G. Taentzer, M. Minas, and A. Schürr. Application of Graph Transformation to Visual Languages. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 2: Applications, Languages and Tools*, pages 105–181. World Scientific, 1999.

[3] L. Baresi and M. Pezze. A Toolbox for Automating Visual Software Engineering. In R. Kutsche and H. Weber, editors, *Proc. Fundamental Approaches to Software Engineering (FASE'02), Grenoble, April 2002*, pages 189 – 202. Springer LNCS 2306, 2002.

[4] P. Bottoni, A. Schürr, and G. Taentzer. Efficient Parsing of Visual Languages based on Critical Pair Analysis and Contextual Layered Graph Transformation. In *Proc. IEEE Symposium on Visual Languages*, September 2000. Long version available as technical report SI-2000-06, University of Rom.

[5] J. de Lara and H. Vangheluwe. ATOM^3: A Tool for Multi-Formalism Modelling and Meta-Modelling. In R. Kutsche and H. Weber, editors, *Proc. Fundamental Approaches to Software Engineering (FASE'02), Grenoble, April 2002*, pages 174 – 188. Springer LNCS 2306, 2002.

[6] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. Attributed type graphs with inheritance. In *Technical Report 2005/3*. Technical University Berlin, 2005.

[7] H. Ehrig, U. Prange, and G. Taentzer. Fundamental theory for typed attributed graph transformation. In F. Parisi-Presicce, P. Bottoni, and G. Engels, editors, *Proc. 2nd Int. Conference on Graph Transformation (ICGT'04), Rom e, Italy*, pages 161–177. LNCS 3256, Springer, October 2004.

[8] R. Heckel, J. Küster, and G. Taentzer. Confluence of Typed Attributed Graph Transformation Systems. In A. Corradini, H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *Proceedings of ICGT 2002*, volume 2505 of *LNCS*, pages 161–176. Springer, 2002.

[9] R. Heckel, J. Küster, and G. Taentzer. Towards Automatic Translation of UML Models into Semantic Domains . In H.-J. Kreowski, editor, *Proc. of APPLIGRAPH Workshop on Applied Graph Transformation (AGT 2002)*, pages 11 – 22, 2002.

[10] R. Heckel and A. Wagner. Ensuring Consistency of Conditional Graph Rewriting - a Constructive Approach. In *Proceedings of SEGRAGRA 1995*, volume 2 of *ENTCS*, 1995.

[11] K. Marriott and B. Meyer. *Visual Language Theory*. Springer, 1998.

[12] Object Management Group (OMG). MDA, MOF and UML specifications at the OMG web page. URL: http://www.omg.org.

[13] D. Varro. A formal semantics of UML statecharts by model transition systems. In A. Corradini, H. Ehrig, H. Kreowski, and G. Rozenberg, editors, *Proc. International Conference on Graph Transformation (ICGT'02), Barcelona, 2002*, pages 378–392. Springer LNCS 2505, 2002.

[14] J. B. Warmer and A. Kleppe. The Object Constraint Language: Precise Modeling with UML. *Addison-Wesley Object Technology Services*, 1998.