# Transformation of Attributed Structures with Cloning*

Dominique Duval[1], Rachid Echahed[2], Frederic Prost[2], and Leila Ribeiro[3]

[1] LJK - Université de Grenoble
[2] LIG - Université de Grenoble
[3] INF - Universidade Federal do Rio Grande do Sul

**Abstract.** Copying, or cloning, is a basic operation used in the specification of many applications in computer science. However, when dealing with complex structures, like graphs, cloning is not a straightforward operation since a copy of a single vertex may involve (implicitly) copying many edges. Therefore, most graph transformation approaches forbid the possibility of cloning. We tackle this problem by providing a framework for graph transformations with cloning. We use attributed graphs and allow rules to change attributes. These two features (cloning/changing attributes) together give rise to a powerful formal specification approach. In order to handle different kinds of graphs and attributes, we first define the notion of attributed structures in an abstract way. Then we generalise the sesqui-pushout approach of graph transformation in the proposed general framework and give appropriate conditions under which attributed structures can be transformed. Finally, we instantiate our general framework with different examples, showing that many structures can be handled and that the proposed framework allows one to specify complex operations in a natural way.

## 1 Introduction

Graph structures and graph transformation have been successfully used as foundational concepts of modelling languages in a wide range of areas related to software engineering. Such a success mainly stems from the intuitive and pictorial features of graphs which ease the writing as well as the understanding of specifications. Several ways to define graph transformation rules have been proposed (see e.g., [22,12,14] for a survey). We can distinguish two main approaches: The algorithmic approach which is rather pragmatic and defines graph transformation rules by means of the algorithms used to transform the graphs (e.g.[3]) and the algebraic approach which is more abstract (e.g. [15]). This latter borrows notions from category theory to define graph transformation rules. The most popular algebraic approaches are the double pushout (DPO) [15,7] and the single pushout (SPO) [13].

* This work has been partially funded by projects CLIMT (ANR-11-BS02-016), TGV (CNRS-INRIA-FAPERGS/156779 and 12/0997-7), VeriTeS (CNPq/FAPERGS 11/2016-2 and 485048/2012-4) and PLATUS (CNPq 306843/2010-2).

S. Gnesi and A. Rensink (Eds.): FASE 2014, LNCS 8411, pp. 310–324, 2014.

Very often, graph structures are endowed with attributes. Such attributes, which enrich nodes and edges with data values, have been proven very useful to enhance the expressiveness of visual modelling frameworks (see, e.g., UML diagrams). These attributes can be simple names of an alphabet (labels) or elaborated expressions of a given language. Several investigations tackling attributed graph transformations have been proposed in the literature, see e.g. [19,18,4,11,20,16]. These proposals follow the so-called double pushout approach to define graph transformation steps. This approach can be used in many applications (see e.g. [7]) but it forbids actions which consist in cloning nodes together with their incident edges (merging of nodes is also usually forbidden). Moreover, this approach also prevents the application of rules that erase a node when there are edges connected to this node in the graph that represents the state (erasing nodes is only possible if all connected arcs are explicitly deleted by the rule). However, there are applications in which these restrictions of DPO would lead to rather complex specifications. For instance, duplicating or erasing some component may be very useful in the development process of an architecture, and should be a simple operation. Also, making a security copy of a virtual machine in a cloud (for fault-tolerance reasons) is a very reasonable operation, as well as switching down a (physical) machine from the infrastructure of a cloud. To model such situations we may profit from cloning/merging as basic operations in a formalism. But we certainly need to use attributed structures to get a suitable formalism for real applications. In this paper, we propose a framework that has both the ability to model cloning/merging of entities in a natural way, and also the feature of using attributes together with the graphs.

To develop our proposal, we follow a more recent approach of graph transformation known as the sesqui-pushout approach (SqPO) [6]. This latter is a conservative extension of DPO with some additional features such as deletion or cloning.

A rule is defined, as in the DPO approach, by means of a span of the form $(l : L \leftarrow K \rightarrow R : r)$ where the morphisms $l$ and $r$ are not necessarily monos. The fact that $l$ is not mono allows one to duplicate some nodes and edges. Notice that most proposals dealing with attributed graphs assume $l$ to be mono. A rewrite step can be depicted as follows where the left square is a final pullback complement and the right square is a pushout. The intuition is analogous to the DPO approach: the left square specifies what is removed (and also what is cloned) by the rule application and the right square creates the new items. The difference, besides allowing non injective rules, is that when applying the rule the so called dangling condition does not need to be checked: if there are edges in $G$ connected to nodes in the image of $m$ that is deleted by the rule, these edges are automatically removed by the rule application. In DPO, in such a situation a rule would not be applicable.

$$\begin{array}{ccccc} L & \xleftarrow{l} & K & \xrightarrow{r} & R \\ \downarrow{\scriptstyle m} & (FPBC) & \downarrow{\scriptstyle d} & (PO) & \downarrow{\scriptstyle h} \\ G & \xleftarrow{l_1} & D & \xrightarrow{r_1} & H \end{array}$$

Sesqui-pushout: $G \overset{\text{sqpo}}{\Longrightarrow} H$

In order to consider different kinds of graphs and attributes, we present our approach in a general setting. That is to say, we consider structures of the form $\widehat{G} = (G, A, \alpha)$ made of an object $G$ whose elements may be attributed, an object $A$ defining attributes and a partial function $\alpha$ which assigns to some elements of $G$ attributes in $A$. The fact that $\alpha$ is partial turns out to be very useful to write transformation rules that change the attributes of some elements of $G$ (see, e.g. [17,4]). We do not assume $G$ to be necessarily a graph nor do we assume $A$ to be necessarily an algebra. We thus elaborate a framework which can be instantiated with different kinds of structures and attributes fulfilling some criteria we introduce in this paper. Therefore we can handle different graphs with various kinds of attributes (algebras, lambda-terms, finite labels, syntactic theories, etc.). Similar objectives, with different outcome, have been recently investigated in [16] for the DPO approach.

The rest of the paper is organized as follows. The next section introduces the category of attributed structures and provides some definitions which may help the understanding of the paper. Section 3 recalls briefly the useful definitions regarding the sesqui-pushout approach. Then, Section 4 shows how to lift SqPO rewriting in the context of attributed structures. Sections 5 and 6 illustrate our approach through some examples while related work are discussed in Section 7. Concluding remarks are given in Section 8. The missing proofs may be found in [10].

## 2 Attributed Structures

In this section we define the notion of attributed structures and set some notations.

**Structures.** Let $\mathbf{G}$ be a category and $S : \mathbf{G} \to \mathbf{Set}$ a functor from $\mathbf{G}$ to the category of sets. For instance, $\mathbf{G}$ may be the category of graphs $\mathbf{Gr}$ [22] and $S$ may be either the *vertex* functor $V$ defined by $V(G) = V_G$ and $V(g) = g_V$, or the *edge* functor $E$ defined by $E(G) = E_G$ and $E(g) = g_E$, or the functor $V + E$ which maps each graph $G$ to the disjoint union $V_G + E_G$ and each morphism $g : G_1 \to G_2$ to the map $g_V + g_E$.

**Attributes.** Let $\mathbf{A}$ be a category and $T : \mathbf{A} \to \mathbf{Set}$ a functor from $\mathbf{A}$ to the category of sets. For instance, $\mathbf{A}$ may be the category $\mathbf{Alg}(\Sigma)$ of $\Sigma$-algebras [23] for some signature $\Sigma = (S, \Omega)$, or more generally the category $\mathbf{Mod}(Sp)$ of models of an equational specification $Sp = (\Sigma, E)$, made of a signature $\Sigma$ and a set of equations $E$. Then the functor $T : \mathbf{A} \to \mathbf{Set}$ may be such that $T(A) = \sum_{s \in S} A_s$, i.e., $T$ maps each $\Sigma$-algebra $A$ to the disjoint union of its carriers, or more generally $T(A) = \sum_{s \in S'} A_s$ for some fixed subset $S'$ of $S$. In the following, we sometimes write $Fx$ instead of $F(x)$ when a functor $F$ is applied to an object or a morphism $x$.

**Definition 1.** *The category of* attributed structures **AttG** *(with respect to the functors $S$ and $T$) is the comma category $(S \downarrow T)$. Thus, an* attributed structure *is a triple $\widehat{G} = (G, A, \alpha)$ made of an object $G$ in $\mathbf{G}$, an object $A$ in $\mathbf{A}$ and a*

*map* $\alpha : S(G) \to T(A)$ *(in* **Set***) ; and a* morphism of attributed structures $\widehat{g} : \widehat{G} \to \widehat{G'}$*, where* $\widehat{G} = (G, A, \alpha)$ *and* $\widehat{G'} = (G', A', \alpha')$*, is a pair* $\widehat{g} = (g, a)$ *made of a morphism* $g : G \to G'$ *in* **G** *and a morphism* $a : A \to A'$ *in* **A** *such that* $\alpha' \circ Sg = Ta \circ \alpha$ *(in* **Set***).*

$$\begin{array}{ccc} \widehat{G} \\ {\scriptstyle \widehat{g}}\downarrow \\ \widehat{G'} \end{array} \quad = \quad \begin{array}{ccc} G \\ {\scriptstyle g}\downarrow \\ G' \end{array} \qquad \begin{array}{ccc} SG & \xrightarrow{\alpha} & TA \\ {\scriptstyle Sg}\downarrow & = & \downarrow{\scriptstyle Ta} \\ SG' & \xrightarrow{\alpha'} & TA' \end{array} \qquad \begin{array}{c} A \\ \downarrow{\scriptstyle a} \\ A' \end{array}$$

**Partial Maps.** Let **Part** be the category of sets with partial maps, which contains **Set**. A partial map $f$ from $X$ to $Y$ is denoted $f : X \rightharpoonup Y$ and its domain of definition is denoted $\mathcal{D}(f)$. The partial order between partial maps is denoted $\leq$, it endows **Part** with a structure of 2-category. By composing $S$ and $T$ with the inclusion of **Set** in **Part** we get two functors $S_p : \mathbf{G} \to \mathbf{Part}$ and $T_p : \mathbf{A} \to \mathbf{Part}$.

**Definition 2.** *The category of* partially attributed structures **PAttG** *(with respect to the functors S and T) is defined as follows. A* partially attributed structure *is a triple* $\widehat{G} = (G, A, \alpha)$ *made of an object G in* **G***, an object A in* **A** *and a partial map* $\alpha : S_p(G) \rightharpoonup T_p(A)$ *(in* **Part***) ; and a* morphism of partially attributed structures $\widehat{g} : \widehat{G} \to \widehat{G'}$*, where* $\widehat{G} = (G, A, \alpha)$ *and* $\widehat{G'} = (G', A', \alpha')$*, is a pair* $\widehat{g} = (g, a)$ *made of a morphism* $g : G \to G'$ *in* **G** *and a morphism* $a : A \to A'$ *in* **A** *such that* $\alpha' \circ S_p g \geq T_p a \circ \alpha$ *(in* **Part***).*

$$\begin{array}{ccc} \widehat{G} \\ {\scriptstyle \widehat{g}}\downarrow \\ \widehat{G'} \end{array} \quad = \quad \begin{array}{ccc} G \\ {\scriptstyle g}\downarrow \\ G' \end{array} \qquad \begin{array}{ccc} S_pG & \xrightarrow{\alpha} & T_pA \\ {\scriptstyle S_pg}\downarrow & \geq & \downarrow{\scriptstyle T_pa} \\ S_pG' & \xrightarrow{\alpha'} & T_pA' \end{array} \qquad \begin{array}{c} A \\ \downarrow{\scriptstyle a} \\ A' \end{array}$$

*Such a morphism of partially attributed structures is called* strict *when* $\alpha' \circ S_p(g) = T_p(a) \circ \alpha$.

*Remark 1.* Clearly, **AttG** is a full subcategory of **PAttG** and every morphism in **AttG** is a strict morphism in **PAttG**. The subcategory **AttG** of **PAttG** is called the subcategory of *totally attributed structures*.

**Definition 3.** *A morphism of (partially) attributed structure* $\widehat{g} : \widehat{G} \to \widehat{G'}$ preserves attributes *if* $\widehat{G} = (G, A, \alpha)$*,* $\widehat{G'} = (G', A, \alpha')$ *and* $\widehat{g} = (g, id_A)$ *for some object A in* **A**.

**Notations.** We will omit the subscript $p$ in $S_p$ and $T_p$. Let $(G, A, \alpha)$ be a (partially) attributed structure, the notation $x : t$ means that $x \in S(G), t \in T(A)$ and $\alpha(x) = t$ (i.e., $x$ has $t$ as attribute), and the notation $x : \perp$ means that $x \in S(G)$, $x \notin \mathcal{D}(\alpha)$ (i.e., $x$ has no attribute). Let $(G, A, \alpha)$ and $(G', A', \alpha')$ be attributed structures, let $g : G \to G'$ in **G** and $a : A \to A'$ in **A**, then

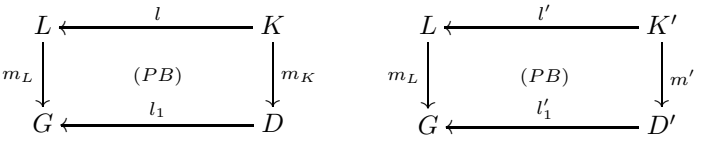
$(g,a) : (G,A,\alpha) \to (G',A',\alpha')$ is a morphism of attributed structures if and only if for all $x \in S(G)$ and $t \in T(A)$ $x : t \implies g(x) : a(t)$. Let $(G,A,\alpha)$ and $(G',A',\alpha')$ be partially attributed structures, let $g : G \to G'$ in $\mathbf{G}$ and $a : A \to A'$ in $\mathbf{A}$, then $(g,a) : (G,A,\alpha) \to (G',A',\alpha')$ is a morphism of partially attributed structures if and only if for all $x \in SG$ and $t \in TA$ $x \in \mathcal{D}(\alpha) \implies g(x) \in \mathcal{D}(\alpha')$ and then $x : t \implies g(x) : a(t)$, and $(g,a)$ is strict if and only if for all $x \in SG$ and $t \in TA$ $x \in \mathcal{D}(\alpha) \iff g(x) \in \mathcal{D}(\alpha')$, and then $x : t \implies g(x) : a(t)$. The notation $x : \bot$ can be misleading: of course we can extend $a : TA \to TA'$ as $a : TA + \{\bot\} \to TA' + \{\bot\}$ by setting $a(\bot) = \bot$, but then it is *false* that $x : t \implies g(x) : a(t)$ for each $x \in SG$ and $t \in TA + \{\bot\}$. In fact, for each morphism of partially attributed structures $(g,a)$ we have $g(x) : \bot \implies x : \bot$, and it is only when $g$ is strict that in addition $x : \bot \implies g(x) : \bot$.

**Definition 4.** *The* underlying structure *functor is the functor* $U_{\mathbf{G}} : \mathbf{PAttG} \to \mathbf{G}$ *which maps an attributed structure* $(G,A,\alpha)$ *to the object* $G$ *and* $(g,a)$ *to the morphism* $g$. *The* underlying attributes *functor is the functor* $U_{\mathbf{A}} : \mathbf{PAttG} \to \mathbf{A}$ *which maps an attributed structure* $(G,A,\alpha)$ *to the object* $A$ *and* $(g,a)$ *to the morphism* $a$.

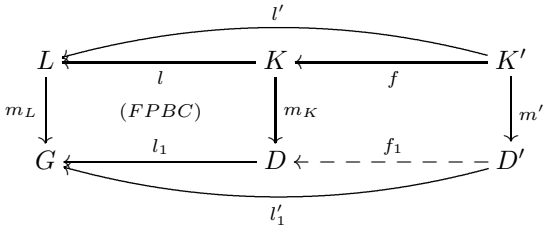
## 3 Sesqui-Pushouts

In this section we briefly recall the definition of sesqui-pushout (SqPO) rewriting, introduced in [6]. A sesqui-pushout rewriting step is made of a final pullback complement (FPBC) followed by a pushout (PO). The definitions of FPBC and SqPO are reminded here, in any category $\mathbf{C}$. The initiality property of POs and the finality property of FPBCs imply that POs, FPBCs and SqPOs are unique up to isomorphism, when they exist.

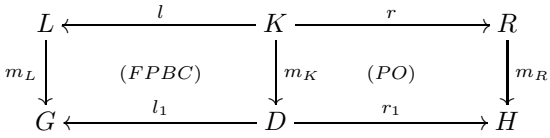
**Definition 5.** *The* final pullback complement *(FPBC) of a morphism* $m_L : L \to G$ *along a morphism* $l : K \to L$ *is a pullback (PB) (below on the left) such that for each pullback (below on the right)*




*and each morphism* $f : K' \to K$ *such that* $l \circ f = l'$ *there is a unique morphism* $f_1 : D' \to D$ *such that* $l_1 \circ f_1 = l'_1$ *and* $f_1 \circ m' = m_K \circ f$.



**Definition 6.** *The* sesqui-pushout *of a morphism $m_L : L \to G$ along a span of morphisms $(l : L \leftarrow K \to R : r)$ is the FPBC of $m_L$ along $l$ followed by the PO of $m_K$ along $r$ (see diagram below).*




A comparison of SqPO with DPO and SPO approaches can be found in [6], where it is stated that "*Probably the most original and interesting feature of sesqui-pushout rewriting is the fact that it can be applied to non-left-linear rules as well, and in this case it models the cloning of structures.*"

In the category of graphs, under the assumption that $m_L : L \to G$ is an inclusion, the result of the sesqui-pushout can be described as follows [6, Section 4.1], [9]. With respect to a rule $(l : L \leftarrow K \to R : r)$, let us call *tri-node* a triple $(n_L, n_K, n_R)$ where $n_L$, $n_K$ and $n_R$ are nodes in $L$, $K$ and $R$ respectively and where $n_L = l(n_K)$ and $n_R = r(n_K)$. Since $m_L$ is an inclusion, $L$ is a subgraph of $G$. Let $\overline{L}$ be the subgraph of $G$ made of all the nodes outside $L$ and all the vertices between these nodes. Let $\widetilde{L}$ be the set of edges outside $L$ with at least one endpoint in $L$ (called the *linking edges*), so that $G$ is the disjoint union of $L$, $\overline{L}$ and $\widetilde{L}$. Then, up to isomorphism, $m_R$ is an inclusion and $H$ is obtained from $G$ by replacing $L$ by $R$ and by "gluing $R$ and $\overline{L}$ in $H$ according to the way $L$ and $\overline{L}$ are glued in $G$", which means precisely that $H$ is the disjoint union of $R$, $\overline{L}$ and the following set $\widetilde{R}$ of linking edges (see [9] for more details):

- if $n$ is a node in $R$ and $p$ a node in $\overline{L}$, there is an edge from $n$ to $p$ in $\widetilde{R}$ for each tri-node $(n_L, n_K, n_R)$ with $n_R = n$ and each edge from $n_L$ to $p$ in $\widetilde{L}$;
- if $n$ is a node in $\overline{L}$ and $p$ a node in $R$, there is an edge from $n$ to $p$ in $\widetilde{R}$ for each tri-node $(p_L, p_K, p_R)$ with $p_R = p$ and each edge from $n$ to $p_L$ in $\widetilde{L}$;
- if $n$ and $p$ are nodes in $R$, there is an edge from $n$ to $p$ in $\widetilde{R}$ for each tri-node $(n_L, n_K, n_R)$ with $n_R = n$, each tri-node $(p_L, p_K, p_R)$ with $p_R = p$ and each edge from $n_L$ to $p_L$ in $\widetilde{L}$.

## 4 Attributed Sesqui-Pushout Rewriting

In this section we define rewriting of attributed structures based on sesqui-pushouts, then we construct such SqPOs from SqPOs of the underlying (non-attributed) structures.

**Definition 7.** *Given an object $A$ of* **A**, *a* rewriting rule *with attributes in $A$ is a span $(\widehat{l} : \widehat{L} \leftarrow \widehat{K} \to \widehat{R} : \widehat{r})$, or simply $(\widehat{l}, \widehat{r})$, made of morphisms $\widehat{l}$ and $\widehat{r}$ in* **PAttG** *which preserve attributes and such that $\widehat{L}$ and $\widehat{R}$ are totally attributed structures. A* match *for a rule $(\widehat{l}, \widehat{r})$ in an attributed structure $\widehat{G}$ is a morphism $\widehat{m} = (m, a) : \widehat{L} \to \widehat{G}$ in* **AttG** *such that the map $Sm$ is injective. The* SqPO rewriting step *(or simply the* rewriting step*) applying a rule $(\widehat{l}, \widehat{r})$ to a match $\widehat{m}$ is the sesqui-pushout of $\widehat{m}$ along $(\widehat{l}, \widehat{r})$ in the category* **PAttG**.

From the definition above, a rewrite rule is characterised by (i) the object $A$ of attributes, (ii) the attributed structures $\widehat{L}$, $\widehat{K}$and $\widehat{R}$ and (iii) the span of structures $(l : L \leftarrow K \rightarrow R : r)$. A match $\widehat{m}$ must have an injective underlying morphism of structures but it may modify the attributes. In contrast, the morphisms $\widehat{l} = (l, id_A)$ and $\widehat{r} = (r, id_A)$ in a rule have arbitrary underlying morphisms of structures $l$ and $r$, thus allowing items to be added, deleted, merged or cloned, but they must preserve attributes since their underlying morphism on attributes is the identity $id_A$. However, since $\widehat{K}$ is only partially attributed, any element $x \in SK$ without attribute may be mapped to $l(x) : a$ in $\widehat{L}$ and to $r(x) : a'$ in $\widehat{R}$ with $a \neq a'$. Thus the assignment of attributes to vertices/edges may change in the transformation process.

In the following when $(m, a)$ is a match we often assume that $Sm$ is an inclusion, rather than any injection; in this way the notations are simpler while the results are the same, since all constructions (PO, PB, FPBC) are up to isomorphism.

The construction of a sesqui-pushout in **PAttG** can be made in two steps: first a sesqui-pushout in **G**, which depends only on the properties of the category **G**, then its lifting to **PAttG**, which does not depend any more on **G**. Moreover, this lifting is quite simple: since the morphisms $l$ and $r$ do not modify the attributes, it can be proved that $m_K$ and $m_R$ have the same underlying morphism on attributes as $m_L$. This is stated in Theorem 1.

**Theorem 1.** *Let us assume that the functors* $U_{\mathbf{G}} : \mathbf{PAttG} \rightarrow \mathbf{G}$, $U_{\mathbf{A}} : \mathbf{PAttG} \rightarrow \mathbf{A}$, $S : \mathbf{G} \rightarrow \mathbf{Set}$ *and* $T : \mathbf{A} \rightarrow \mathbf{Set}$ *preserve PBs and that the functor* $S$ *preserves POs. Let* $(\widehat{l} : \widehat{L} \leftarrow \widehat{K} \rightarrow \widehat{R} : \widehat{r})$ *be a rewriting rule and* $\widehat{m_L} = (m_L, a) : \widehat{L} \rightarrow \widehat{G}$ *a match. If diagram* $\Delta$ *(below on the left) is a SqPO rewriting step in* **G** *then diagram* $\widehat{\Delta}$ *(below on the right) is a SqPO rewriting step in* **PAttG** *and* $(m_R, a)$ *is a match.*

$\Delta$ :

$$\begin{array}{ccccc} L & \xleftarrow{l} & K & \xrightarrow{r} & R \\ \downarrow{\scriptstyle m_L} & (FPBC) & \downarrow{\scriptstyle m_K} & (PO) & \downarrow{\scriptstyle m_R} \\ G & \xleftarrow{l_1} & D & \xrightarrow{r_1} & H \end{array}$$

$\widehat{\Delta}$ :

$$\begin{array}{ccccc} \widehat{L} & \xleftarrow{(l,id_A)} & \widehat{K} & \xrightarrow{(r,id_A)} & \widehat{R} \\ \downarrow{\scriptstyle (m_L,a)} & (FPBC) & \downarrow{\scriptstyle (m_K,a)} & (PO) & \downarrow{\scriptstyle (m_R,a)} \\ \widehat{G} & \xleftarrow{(l_1,id_{A_1})} & \widehat{D} & \xrightarrow{(r_1,id_{A_1})} & \widehat{H} \end{array}$$

*Proof.* Since a sesqui-pushout is a FPBC followed by a PO, this proof relies on similar results about the lifting of FPBCs and the lifting of POs (see [10]).

Let us summarize what may occur for an element $x \in SD$. If $x \notin SK$ then only one case may occur:

$$l_1(x) : t_1 \longleftarrow\!\shortmid\, x : t_1 \shortmid\!\longrightarrow r_1(x) : t_1$$

If $x \in SK$ then two cases may occur:

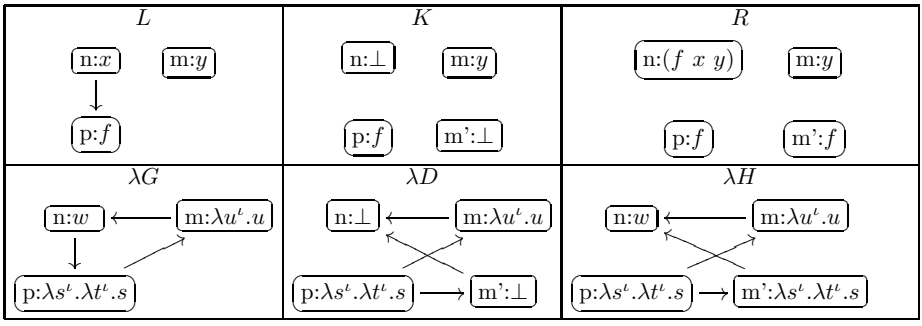
$$\begin{array}{ccccc} l(x) : t & \longleftarrow\!\shortmid & x : t & \shortmid\!\longrightarrow & r(x) : t \\ \downarrow & & \downarrow & & \downarrow \\ l_1(x) : a(t) & \longleftarrow\!\shortmid & x : a(t) & \shortmid\!\longrightarrow & r_1(x) : a(t) \end{array} \qquad \begin{array}{ccccc} l() x : t & \longleftarrow\!\shortmid & x : \bot & \shortmid\!\longrightarrow & r(x) : t' \\ \downarrow & & \downarrow & & \downarrow \\ l_1(x) : a(t) & \longleftarrow\!\shortmid & x : \bot & \shortmid\!\longrightarrow & r_1(x) : a(t') \end{array}$$

## 5 Graph Transformations with Simply Typed λ-terms as Attributes

In this section we consider simply typed λ-terms as attributes. The choice of the λ-calculus can be motivated by the possibility to perform higher-order computations (functions can be passed as parameters). We refer to [2] for more details concerning the simply-typed λ-calculus, though basic notions of λ-calculus are enough to understand the example provided in this section.

First, let us choose the categories $\mathbf{G}$ and $\mathbf{A}$ and the functors $S$ and $T$. Let $\mathbf{G} = \mathbf{Gr}$ be the category of graphs. Let $S : \mathbf{G} \to \mathbf{Set}$ be the functor which maps each graph to the disjoint union of its set of vertices and its set of edges. We define the category $\mathbf{A}$ as the category where objects are sets $\Lambda(X)$ of simply typed λ-terms, à la Church, built over variables in $X$. For the sake of simplicity we only consider one base type $\iota$. Simply typed λ-terms in $\Lambda(X)$, noted $t$, and types, noted $\tau$, are defined inductively by: $\tau ::= \iota \mid \tau \to \tau$ and $t ::= x \mid (t\ t) \mid \lambda x^{\tau}.t$ with $x \in X$. A morphism $m$ from $\Lambda(X)$ to $\Lambda(X')$ is totally defined by a substitution from $X$ to $\Lambda(X')$. The functor $T : \mathbf{A} \to \mathbf{Set}$ is such that $T(\Lambda(X))$ is the set of normal forms of elements in $\Lambda(X)$. Other choices for $T$ are possible, for instance $T(\Lambda(X))$ could be chosen to be $\Lambda(X)$ itself. However in this case there would be no reduction in the attributes while rewriting. With the definitions as above, the functors $U_{\mathbf{G}} : \mathbf{PAttG} \to \mathbf{Gr}$, $U_{\mathbf{A}} : \mathbf{PAttG} \to \mathbf{A}$, $S : \mathbf{Gr} \to \mathbf{Set}$ and $T : \mathbf{A} \to \mathbf{Set}$ preserve pullbacks, and $S$ preserves pushouts.

Graph transformations can be coupled with λ-term evaluation. For instance, a vertex, $n$, of a right-hand side, $R$, of a rule may be attributed with a λ-term, $t$, containing free variables which occur in the left-hand side $L$. A match, $\sigma$ of such a rule instantiates the free variables. Firing the rule will result in (i) the computation of the normal form of the λ-term $\sigma(t)$ and (ii) its attribution to the image of vertex $n$ in the resulting transformed graph. Below we give an example of such a rule and illustrate it on the graph $\lambda G$.




Graph morphisms are represented via vertex name sharing, and $U_{\mathbf{A}}$ can be deduced from them (for instance attribute $x$ in $L$ is instantiated by attribute $w$ in $\lambda G$ because of the match on vertex $n$, likewise $f$ is instantiated by $\lambda s^{\iota}.\lambda t^{\iota}.s$ and $y$ is instantiated by $\lambda u^{\iota}.u$). In this example several features of our framework are underlined. First, notice that vertex $m$ in $L$ is cloned, as a structure, into $m$ and $m'$. This cloning of structure implies that the edges incident to $m$ in $\lambda G$

are to be duplicated for $m$ and $m'$ in $\lambda H$. As for attributes, the example shows that the structure can be cloned while the attributes can be changed (this is the case for the attribute of vertex $m'$). The edge between vertices $n$ and $p$ is erased since it is matched and is not present in $K$ nor in $R$. Furthermore, the attribute of $n$ in $R$ shows a higher-order computation. Via the match, $f$ is substituted by the function $\lambda s^\iota.\lambda t^\iota.s$ and is applied to the instances of $x$ and $y$. In $\lambda H$ the attribute of $n$ is the normal form of $(\lambda s^\iota.\lambda t^\iota.s\ w\ \lambda u^\iota.u)$ which is $w$. Attributes can be easily copied, e.g., $f$ occurs twice in $R$. Finally, attributes of a vertex can be modified thanks to the partiality of the attribution in $K$. It is witnessed on vertices $n$ and even $m'$ which is a clone of $m$. In fact $m'$ clones only the incident edges of $m$, one would have to write $m' : y$ to copy the attribute of $m$ as well. Free variables are used to provide arguments of lambda-terms. This allows us to simulate the attribute dependency relation introduced in [5].

## 6 Graph Transformations with Attributes Defined Equationally: Administration of Cloud Infrastructure

In this section we explore how our framework allows us to take into account attributed graph transformations with attributes built over equational specifications. First we instantiate the definition with appropriate categories and functors, and then model an example.

Let the category $\mathbf{G}$ and functor $S : \mathbf{G} \to \mathbf{Set}$ be defined as in section 5. Let $T : \mathbf{A} \to \mathbf{Set}$ be the functor which maps each model of $Sp = (\Sigma, E)$, with $\Sigma = (S, \Omega)$, to the disjoint union of the carriers sets $A_s$ for $s$ in some given set of sorts $S$. With the definitions as above, the functors $U_{\mathbf{G}} : \mathbf{PAttG} \to \mathbf{Gr}$, $U_{\mathbf{A}} : \mathbf{PAttG} \to \mathbf{Mod}(Sp)$, $S : \mathbf{Gr} \to \mathbf{Set}$ and $T : \mathbf{Mod}(Sp) \to \mathbf{Set}$ preserve pullbacks, and $S$ preserves pushouts.

Cloud Computing is very popular nowadays [1]. The general idea is that there is a pool, called cloud, of resources (equipment, services, etc.) that may be requested by users. A user may, for example, request a machine with some specific configuration and services from the cloud. The cloud administrator chooses an actual physical machine that is available and installs on it a virtual machine (short VM) according to the user specification. The user does not have to know neither where this machine is nor how the services are implemented, communication with his machine is done via the cloud. The cloud administrator has many tasks to perform, besides communicating with the clients (users). Typical operations involve load balance among the machines, optimisation of the use of machines, etc. In the following we provide the specification using graph transformations of some operations of a cloud administrator. First we define the static structure, defining data types and the states of the system (as attributed graphs), and then we define the operations (as rules). Since the purpose of this case study is to show the use of our framework, we will not describe a complete set of attributes and rules needed to specify the behaviour of a cloud administrator, but concentrate on those parts that make explicit use of the features of the approach.

### 6.1 Cloud Administration: Static Part

To model this scenario, we will use graphs with many attributes. The approach presented in the previous sections could be easily extended to families of attributes. Alternatively, one could use just one record attribute, but we prefer the former representation since the specification becomes more readable. The attributes that will be used are:

**Vertex Attributes:** nodeType, represent the different entities involved in this system, that is, cloud administrator, users, machines and virtual machines. In the graphical notation, this attribute will be denoted by a corresponding image (  , and  , resp.); ident, models the identifier of the vertex; size, denotes the size of the machine and virtual machine; free, describes the amount of unused space in a machine; type, describes the type of a virtual machine (as a simplification, we assumed that there is a set of standard virtual machines that may be requested by users, identified by their types); config, this models the internal configuration of the cloud administrator, probably this would be a set of tables and variables describing the current state of machines and virtual machines;

**Edge Attributes:** edgeType, some arcs will represent physical relations (like a cloud administrator is connected to all machines monitored by it) or "knows"-relations (like a user may know a cloud administrator) and others will represent messages that are sent in the system. Messages will be denoted by dashed arrows, all other relations will be solid edges; type, analogous to the types of vertices; id, used in messages that require a parameter (identifier of a virtual machine).

The data types used in the state graph are defined in specification *Cloud_Sp* (Figure 1). This specification includes sorts for booleans and natural numbers with usual operations and equations, sort T for the different types of virtual machines, and a sort C to describe configurations of a cloud administrator. Such configurations are records containing the current status of the cloud. Due to space limitations, we will not define details of configurations, just use some basic operations (equations will be also omitted).

For example, the graph **G1** depicted in Fig. 2 describes two users and one cloud administrator that knows one machine, $M1$, and two types of virtual machines, $T1$ and $T2$. Actually, the administrator stores the images of the corresponding virtual machines such that, when a request is done, it creates a copy of this image in an available machine. Images are modelled by a special identifier (0). There are also two request messages, one from each user.

### 6.2 Cloud Administration: Dynamic Part

Figure 2 also shows some rules that describe the behaviour of the cloud administrator. Rule ***CreateVM*** models the creation of a new virtual machine. This may happen when there is a request from a user (dashed edge in **L1**) having as

*Cloud_Sp :*
**sorts** B, N, C,T
**opns**

| | |
|---|---|
| ... | *boolean operators...* |
| ... | *natural numbers operators...* |
| *newId:* $\mathsf{C} \times \mathsf{N} \to \mathsf{B}$ | *checks whether an id is not used in a config* |
| *enoughSpace:* $\mathsf{C} \times \mathsf{N} \to \mathsf{B}$ | *checks if there is enough space in a config* |
| *newVM:* $\mathsf{C} \times \mathsf{N} \times \mathsf{N} \times \mathsf{N} \times \mathsf{T} \to \mathsf{C}$ | *includes a new virtual machine in a config* |
| *replVM:* $\mathsf{C} \times \mathsf{N} \times \mathsf{N} \to \mathsf{C}$ | *replicates a virtual machine in a config* |
| *newMch:* $\mathsf{C} \times \mathsf{N} \times \mathsf{N} \times \mathsf{Nat} \to \mathsf{C}$ | *includes a new machine in a config* |
| *mergeMch:* $\mathsf{C} \times \mathsf{N} \times \mathsf{N} \to \mathsf{C}$ | *merges two machines in a config* |
| *replicateAdm?:* $\mathsf{C} \to \mathsf{B}$ | *checks whether a new administrator is needed* |

**eqns**
...

**Fig. 1.** Specification *Cloud_Sp*

attribute the type of virtual machine that is created and the cloud administrator has a corresponding image and a machine to install this VM. Some additional constraints over the attributes are modelled by equations (written below the rule): the identifier that will be used for the new VM is fresh ($newId(c, idVM)$), there is enough free space in the chosen machine ($nVM \leq f$)[1]. The remaining equations describe the values that some attributes will receive when this rule is applied: variable $f'$ depicts the amount of free space in the machine after the installation of the new VM, and $c'$ is the updated configuration of the could administrator. Note that the two instances of the VM in **K1** are copies of the corresponding vertex in **L1**, just the identifier attribute in the second copy is left undefined, the attributes *config* and *free* are also undefined, since their values will change. Finally, in **R1**, this second copy is updated with the new identifier ($idVM$) and it is installed in the machine and sent to the user, and the attributes of the cloud administrator and machine are updated accordingly. Application of this rule to graph **G1** is given by the span $G1 \leftarrow D \rightarrow G2$ on top of Fig. 2.

Rule ***replicateVM*** creates a copy (replica) of a VM in another physical machine. This operation is important for fault tolerance reasons. When this rule is applied, all references to the original VM will also point to the new VM. The configuration of the cloud administrator is updated because any change in one virtual machine must now be propagated to its copy. Rule ***replicateAdm*** is used to replicate the cloud administrator itself. This kind of operation may be necessary, for example, when the number of clients becomes too large or for dependability reasons. The rule that specifies the operation has an equation that checks whether this replication is needed ($replicateAdm?(c)$). In case this is true in the current configuration, the administrator is copied and the two configurations (the original and the copy) are updated (because now they must

[1] To enhance readability, when working with boolean expressions in equations, we omit the right side of the equation. For example, we write simply $newId(c, Id)$ instead of $newId(c, Id) = \mathsf{true}$.

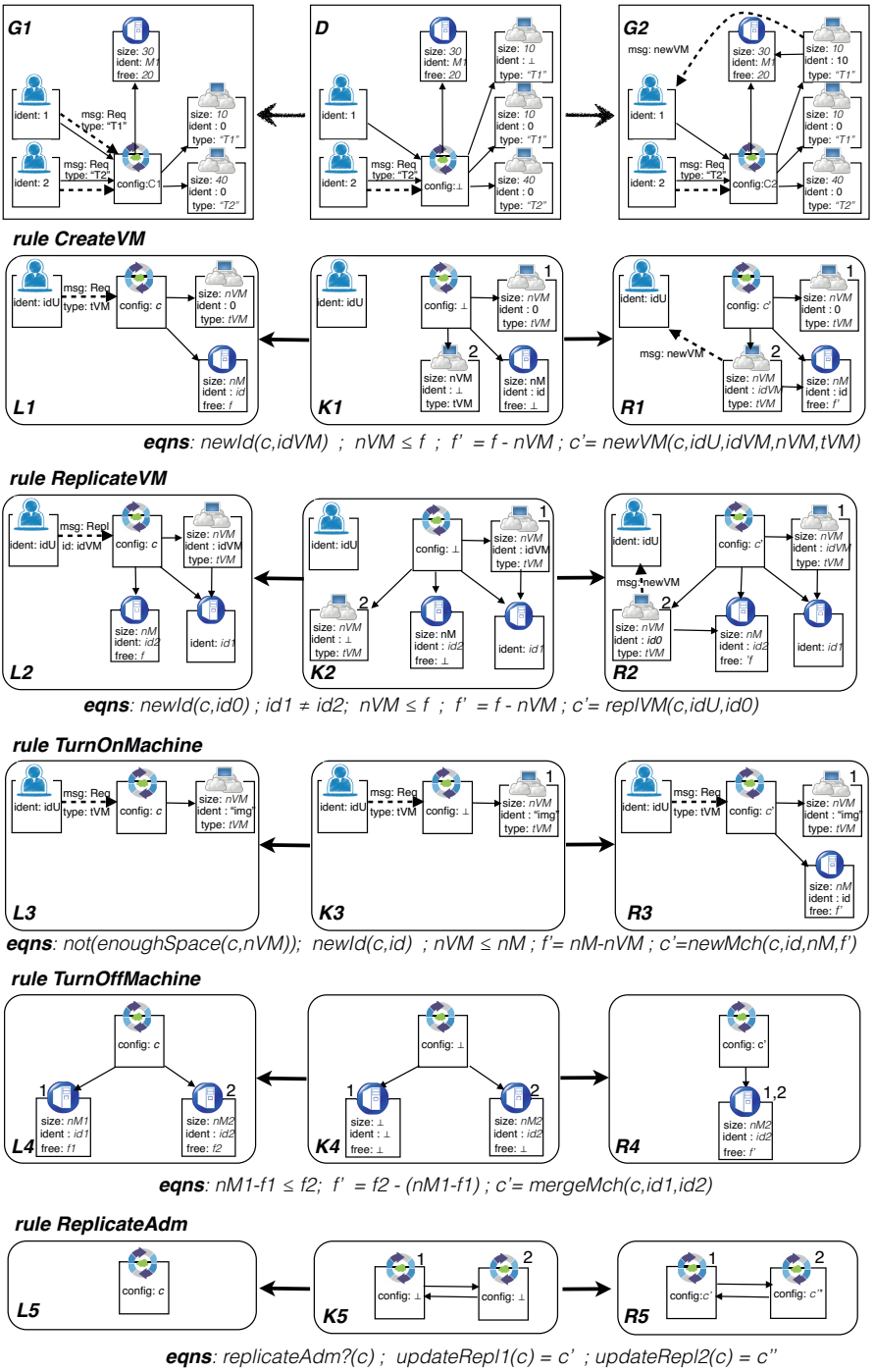



**Fig. 2.** Graph and Rules of the Cloud Administrator

know that some synchronisation is needed to perform the operations). Since these are copies, they manage the same machines and VMs, but now clients may send requests to either of the administrators (when this rule is applied, all edges that were connected to one administrator will also be connected to the copy).

Rules ***TurnOnMachine*** and ***TurnOffMachine*** model the creation and deletion of machines in the system. We assumed that there is an unlimited number of machines that may be connected to the system, and thus there is a need for more capacity ($not(enoughSpace(c, nVM)$ is true), a new machine may be added. We specified a simple version of turning off a machine by merging the vertices that correspond to two different machines. This can be done if the administrator notices that there is enough free space in one machine to accommodate VMs that are in another machines ($nM1 - f1 \leq f2$ is true). When this rule is applied, all VMs that were in both machines will end up in the machine with identifier $id2$.

## 7 Related Work

Various definitions of attributed graphs have been proposed in the literature. Labelled graphs, e.g. [17], where attributes are limited to a simple set of a vocabulary, could be considered as a first step towards attributed graphs. Such a set of vocabulary can be replaced by a specific, possibly infinite, set (of attributes) such as integers yielding particular definition of attributed graphs. This approach has been proposed for instance in [21] and could be considered as a particular case of the definition of attributed graphs we proposed in this paper.

The most popular way to define the data part in attributed graphs is based on algebraic specifications, see e.g. [19,18,4,11]. E-Graphs [11] is one of the principal contribution in this perspective, where an attributed graph gathers, in addition to its own vertices and edges, additional vertices and edges corresponding to the attribution part. The latter vertices correspond to possible attribution values. Such vertices might be infinite whenever the set of attributes is infinite. An attribution edge goes from a vertex or an edge of the considered graph to an attribution vertex. Attribution edges are used to represent graphically attribution functions. Due to the representation of each attribute as a vertex, an E-graph is infinite in general.

To overcome the infinite structures of E-graphs, Symbolic graphs [20] have been proposed. They are E-graphs which have variables as attributes. Such variables can be constrained by means of first order logic formulae. Hence a symbolic graph represents in concise way a (possibly infinite) set of (ground) E-graphs.

In this paper, we have proposed a general definition of attributed structures where the data part is not necessarily specified as an algebra. Our approach is very close to the recent paper by U. Golas [16] where an attributed graph is also defined as a tuple $(G, A, \alpha)$ where $G$ is a given structure, $A$ consists of attribution values and $\alpha$ is a *family* of partial attribution functions. The main difference with our proposal lies in the consideration of attribution functions $\alpha$. For sake of simplicity, we considered simply partial functions for $\alpha$. Generalization to families of functions as in [16] is straightforward.

Besides the variety of definitions of attributed graphs as mentioned above, attributed graph transformation rules have been based mainly on the double pushout approach which departs from the sesquipushout approach we have used in our framework. For a comparison of the double and the sesquipushout approaches we refer the reader to [6]. As far as we are aware of, the present paper presents the first study of attributed graph transformations following the sesquipushout approach and thus featuring the possibility of vertex and edge cloning in presence of attributes. Thanks to partial morphisms, rules allow also deletion and change of attributes.

## 8 Conclusion

In this paper we presented an approach to transformations of attributed structures that allows cloning and merging of items. This approach is based on the SqPO approach to graph transformations, and thus also allows deletion in unknown context. Concerning the attributes, our framework is general in the sense that many different kinds of attributes can be used (not just algebras, as in most attributed graph transformation definitions) and allows that rules change the attributes associated to vertices/edges. The resulting formalism is very interesting and we believe that it can be used to provide suitable specifications of many classes of applications like cloud computing, adaptive systems, and other highly dynamically changing systems.

As future work, we plan to develop more case studies to understand the strengths and weaknesses of this formalism for practical applications. We also want to study analysis methods. Since we are allowing non-injective rules, great part of the theory of graph transformations can not be used directly and we need to investigate which results may hold. Concerning verification of properties, we intent to extend the analysis of graph transformations using theorem provers [8] to attributed SqPO-rewriting.

## References

1. Armbrust, M., Fox, A., Griffith, R., Joseph, A.D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., Zaharia, M.: A view of cloud computing. Commun. ACM 53(4), 50–58 (2010)
2. Barendregt, H., Dekers, W., Statman, R.: Lambda Calculus with Types. Cambridge University Press (2013)
3. Barendregt, H., van Eekelen, M., Glauert, J., Kenneway, R., Plasmeijer, M.J., Sleep, M.: Term graph rewriting. In: de Bakker, J.W., Nijman, A.J., Treleaven, P.C. (eds.) PARLE 1987. LNCS, vol. 259, pp. 141–158. Springer, Heidelberg (1987)
4. Berthold, M.R., Fischer, I., Koch, M.: Attributed graph transformation with partial attribution (2002)
5. Boisvert, B., Féraud, L., Soloviev, S.: Typed lambda-terms in categorical attributed graph transformation. In: Procs of AMMSE 2011. EPTCS, vol. 56, pp. 33–47 (2011)
6. Corradini, A., Heindel, T., Hermann, F., König, B.: Sesqui-pushout rewriting. In: Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozenberg, G. (eds.) ICGT 2006. LNCS, vol. 4178, pp. 30–45. Springer, Heidelberg (2006)

7. Corradini, A., Montanari, U., Rossi, F., Ehrig, H., Heckel, R., Löwe, M.: Algebraic approaches to graph transformation - part I: Basic concepts and double pushout approach. In: Handbook of Graph Grammars, pp. 163–246 (1997)
8. da Costa, S.A., Ribeiro, L.: Verification of graph grammars using a logical approach. Sci. Comput. Program. 77(4), 480–504 (2012)
9. Duval, D., Echahed, R., Prost, F.: Graph transformation with focus on incident edges. In: Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G. (eds.) ICGT 2012. LNCS, vol. 7562, pp. 156–171. Springer, Heidelberg (2012)
10. Duval, D., Echahed, R., Prost, F., Ribeiro, L.: Transformation of attributed structures with cloning (extended version). CoRR, abs/1401.2751 (2014)
11. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamental theory for typed attributed graphs and graph transformation based on adhesive hlr categories. Fundam. Inform. 74(1), 31–61 (2006)
12. Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G. (eds.): Handbook of Graph Grammars and Computing by Graph Transformations, vol. 2: Applications, Languages and Tools. World Scientific (1999)
13. Ehrig, H., Heckel, R., Korff, M., Löwe, M., Ribeiro, L., Wagner, A., Corradini, A.: Algebraic approaches to graph transformation - part ii: Single pushout approach and comparison with double pushout approach. In: Handbook of Graph Grammars, pp. 247–312 (1997)
14. Ehrig, H., Kreowski, H.-J., Montanari, U., Rozenberg, G. (eds.): Handbook of Graph Grammars and Computing by Graph Transformations, vol. 3: Concurrency, Parallelism and Distribution. World Scientific (1999)
15. Ehrig, H., Pfender, M., Schneider, H.J.: Graph-grammars: An algebraic approach. In: 14th Annual Symposium on Foundations of Computer Science (FOCS), The University of Iowa, USA, October 15-17, pp. 167–180. IEEE (1973)
16. Golas, U.: A general attribution concept for models in $\mathcal{M}$-adhesive transformation systems. In: Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G. (eds.) ICGT 2012. LNCS, vol. 7562, pp. 187–202. Springer, Heidelberg (2012)
17. Habel, A., Plump, D.: Relabelling in graph transformation. In: Corradini, A., Ehrig, H., Kreowski, H.-J., Rozenberg, G. (eds.) ICGT 2002. LNCS, vol. 2505, pp. 135–147. Springer, Heidelberg (2002)
18. Heckel, R., Küster, J.M., Taentzer, G.: Confluence of typed attributed graph transformation systems. In: Corradini, A., Ehrig, H., Kreowski, H.-J., Rozenberg, G. (eds.) ICGT 2002. LNCS, vol. 2505, pp. 161–176. Springer, Heidelberg (2002)
19. Löwe, M., Korff, M., Wagner, A.: An algebraic framework for the transformation of attributed graphs. In: Sleep, R., Plasmeijer, M., van Eekelen, M. (eds.) Term Graph Rewriting: Theory and Practice, ch. 14, pp. 185–199. John Wiley & Sons Ltd. (1993)
20. Orejas, F., Lambers, L.: Symbolic attributed graphs for attributed graph transformation. ECEASST 30 (2010)
21. Plump, D., Steinert, S.: Towards graph programs for graph algorithms. In: Ehrig, H., Engels, G., Parisi-Presicce, F., Rozenberg, G. (eds.) ICGT 2004. LNCS, vol. 3256, pp. 128–143. Springer, Heidelberg (2004)
22. Rozenberg, G. (ed.): Handbook of Graph Grammars and Computing by Graph Transformations, vol. 1: Foundations. World Scientific (1997)
23. Sannella, D., Tarlecki, A.: Foundations of Algebraic Specification and Formal Software Development. EATCS Monographs on theoretical computer science. Springer (2012)