Master's Thesis

Attribution Concepts for Graph Transformation

Christoph Peuser

June 10, 2013

Principal Supervisor Prof. Dr. Annegret Habel
Co-Supervisor Dipl.-Inform. Hendrik Radke

Many modeling problems can be solved using graphs. Attribution is an important expansion for graph models. Changes to these models can be formalised using graph transformation and there are several approaches to expanding graph transformation to handle attributed graphs.

In this thesis we briefly survey these existing approaches to attribution in graph transformation. We conduct a more detailed comparison of two of these approaches and present transformations between these attributed graphs.

We further introduce a new attribution concept, based on a generalization of partially labelled graphs and prove their suitability for graph transformation.

Contents

1 Introduction

Graphs are an elegant solution to many modeling problems in computer science. Graph transformation provides a well developed theory to further model the changes these graph models undergo. Attributes play an important role in many graph models, for example UML. While it is technically possible to include additional elements in a graph to model the properties that attributes can be used for, this can easily make the resulting models difficult to understand and therefore less useful.

To enable transformation of attributed graphs, several approaches have been developed:

- In [EEPT06] Typed Attributed Graphs are introduced, expanding the graph by including an algebra for attribute values. To facilitate attribution, Typed Attributed Graphs extend graphs by attribution nodes and attribution edges. All possible data values of the algebra are assumed to be part of the graph. Nodes and edges are attributed by adding an attribution edge that leads to an attribution node.
- [KR12] takes a similar approach, but instead of only encoding the data values, operations and constants are also included in the graph.
- The graph programming language GP, as described in [Plu09], uses a different approach to attribution. Here labels are replaced by sequences of attributes. Rules are complemented by rule schemata in which terms over the attributes are specified. These variables are substituted with attribute values and evaluated during rule application.
- The approach in [LKW93] views graphs as a special case of algebras. These algebras can then additionally specify types for attributes.
- Instead of modifying the definition of graphs and graph transformations to include attributes, [Gol12] defines an attribution concept over arbitrary categories.

We will evaluate two of these approaches, Typed Attributed Graphs [EEPT06] and the graphs used in the graph programming language GP [Plu09], later in this thesis.

Independent of the evaluation of existing approaches, we can already consider some expectations we have of attributed graphs:

- There should be a clear separation between the graph and its attributes, thus giving the user the choice which aspects to model explicitly and which to capture with attributes.
- It should be possible to attribute both nodes and edges.
- There should only be a single value associated with an attribute.
- An attribute should always have a value.
- Ideally, existing rules for unattributed graphs should be usable with minimal changes in an attributed graph.

Additional requirements will be developed later in this thesis, in particular based on our observations during the comparison of existing approaches.

The remainder of this thesis is organized as follows:

Chapter 2 will cover the basic notions of graphs and graph transformation, as well as category theory and algebraic specifications. Existing attribution concepts for graphs will be detailed in chapter 3 and are compared to each other in chapter 4. Further requirements for an attribution concept will be discussed in chapter 5. Also a new concept will be introduced based on these requirements. This new concept will be analyzed in chapter 6. Finally chapter 7 contains conclusions and ideas for future work.

2 Preliminaries

This chapter briefly covers the basic notions of graphs and graph transformation, as well as category theory and algebraic specifications.

2.1 Graph Transformation

This section presents partially labelled graphs and transformation rules with relabelling as introduced in [HP02]. We present this approach instead of the more common approach of rules without relabelling over labelled graphs [EEPT06] since the attribution concept in [Plu09] depends on relabelling and the newly presented concept in chapter 5 has a lot of similarities to graph withs relabelling.

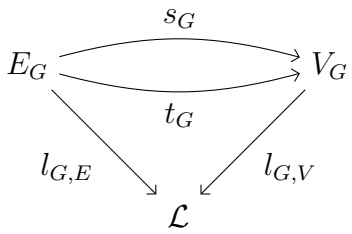
Additionally this section presents definitions based on set theory from [Ehr79], while the literature has moved to definitions based on category theory. For a reader only interested in the applications of graph transformation, the definitions presented here should be sufficient. For the proofs in later chapters however, category theory will be needed.

The definitions based on category theory are presented in section 2.2 later in this chapter.

In the following, directed partially labelled graphs and their morphisms are defined. For an in-depth introduction to Graph Transformation see for example the Handbooks of Graph Grammars and Computing by Graph Transformation [Roz97, EEKR99, EKMR99].

Definition 2.1 (Graph)
A partially labelled graph*, short* graph*, is a system* $G = (V_G, E_G, s_G, t_G, l_{G,V}, l_{G,E})$ *consisting of two finite sets* V_G *and* E_G *of* nodes *and* edges*, two source and target functions* $s_G, t_G : E_G \to V_G$*, and two partial*[1] *labelling functions* $l_{G,V} : V_G \to \mathcal{L}$ *and* $l_{G,E} : E_G \to \mathcal{L}$*, where* $\mathcal{L}$ *is a fixed set of labels.*

[1] We denote the domain of a function f by $Dom(f)$ and the range by $Ran(f)$.



A partially labelled graph G is said to be totally labelled *if $l_{G,V}$ and $l_{G,E}$ are total functions and it is* unlabelled *if the domains of $l_{G,V}$ and $l_{G,E}$ are empty.*

Example 2.1
The graph $H = (V_H, E_H, s_H, t_H, l_{H,V}, l_{H,E})$, with node set $V_H = \{v_1, v_2, v_3, v_4, v_5\}$, edge set $E_H = \{e_1, e_2, e_3, e_4, e_5, e_6\}$, source function $s_H : E_H \to V_H : e_1, e_2, e_3 \mapsto v1, e_4, e_5 \mapsto v_4, e_6 \mapsto v_3$, target function $t_H : E_H \to V_H : e_1 \mapsto v_2, e_2 \mapsto v_4, e_3, e_4 \mapsto v_3, e_5, e_6 \mapsto v_5$, node label function $l_{H,V} : V_H \to \mathcal{L} : v_1 \mapsto a, v_4 \mapsto c, v_5 \mapsto d$ and totally undefined edge label function $l_{H,E}$ is shown in figure 2.1.

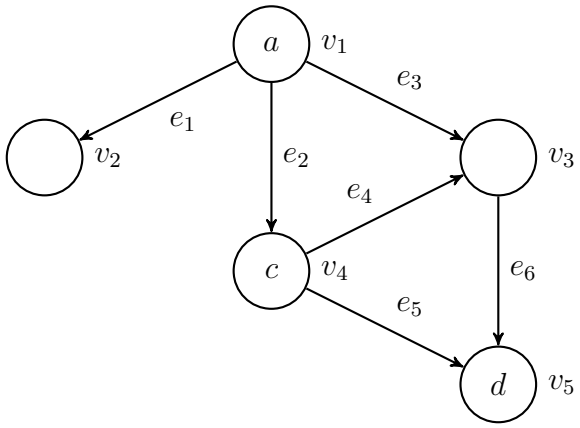
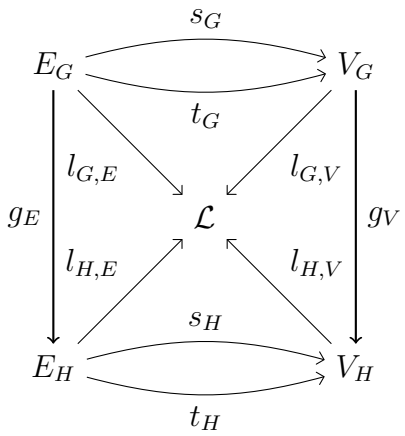


Figure 2.1: Example of a Graph

Remark
If we don't want to distinguish between nodes and edges we use the term item *for either of the two and the notation $x \in G$ means $x \in V_G$ or $x \in E_G$.*

We use functions between the components of graphs that preserve the structure of the graph (i.e. the sources and targets of edges) to express structural similarities between two graphs and we refer to these pairs of functions as graph morphisms.

Definition 2.2 (Graph Morphism)
A graph morphism *$g : G \to H$ from a graph G to a graph H consists of two functions $g_V : V_G \to V_H$ and $g_E : E_G \to E_H$ that preserve sources, targets and labels, that is, $s_H \circ g_E = g_V \circ s_G$, $t_H \circ g_E = g_V \circ t_G$ and $l_H(g(x)) = l_G(x)$ for all x in $Dom(l_G)$, as shown in the diagram below.*



The morphism g preserves undefinedness *if $l_H(g(x)) = \perp$ for all x in $G \setminus Dom(l_G)$, and it* reflects undefinedness *if $g^{-1}(x) \neq \emptyset$ for all x in $H \setminus Dom(l_H)$.*

A morphism g is injective *(*surjective*) if g_V and g_E are injective (surjective), and an* isomorphism *if it is injective, surjective and preserves undefinedness. In the latter case G and H are* isomorphic*, which is denoted by $G \cong H$. Furthermore, we call g an* inclusion *if $g(x) = x$ for all x in G.*

Example 2.2
The morphism shown in figure 2.2 maps the graph $G = (V_G, E_G, s_G, t_G, l_{G,V}, l_{G,E})$, with nodes $V_G = \{s_1, s_2\}$, edges $E_G = \{r_1\}$, source function $s_G : E_G \to V_G : r_1 \mapsto s_1$, target function $t_G : E_G \to V_G : r_1 \mapsto s_2$, node labeling function $l_{G,V} : V_G \to \mathcal{L} : s_1 \mapsto a$ and undefined edge label function $l_{G,E}$ to the graph from example 2.1 as follows: $g_V : V_G \to V_H : s_1 \mapsto v_1, s_2 \mapsto v_2$ and $g_E : E_G \to E_H : r_1 \mapsto e_1$.

The morphism shown in example 2.2 and figure 2.2 is not the only possible morphism between the graphs G and H. We could map s_2 to v_3 or v_4 for example, although $g(r_1)$ would have to change too since its target would not be preserved otherwise. Changing the value of $g(s_1)$ on the other hand is not possible, since there is no node besides v_1 in H that is labelled with a.

Using these morphisms we can now define a rule that is used to change a graph. We create rules by constructing three graphs and two morphisms. To control *where* in

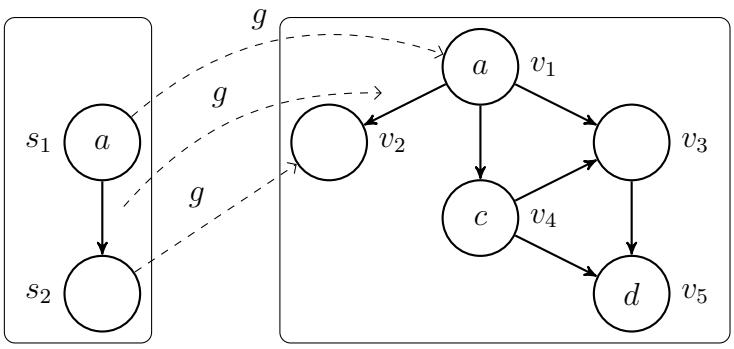


Figure 2.2: Example of a Graph Morphism

a graph we want the changes of a rule to take effect we need to specify a part of the target graph, which we call the *left-hand side*. To specify the elements we want to remove, we construct one graph that contains all but those elements of the left-hand side, we call this graph the *interface*. Finally we create a graph that contains all elements of the interface and any elements we want to add, we call this graph the *right-hand side*. There are now two morphisms from the interface to the left-hand side and the right-hand side respectively.

Definition 2.3 (Rule)
A rule $r = \langle L \leftarrow K \rightarrow R \rangle$ *consists of two inclusions* $K \rightarrow L$ *and* $K \rightarrow R$ *such that*

(1) for all $x \in L$, $l_L(x) = \perp$ *implies* $x \in K$ *and* $l_R(x) = \perp$,

(2) for all $x \in R$, $l_R(x) = \perp$ *implies* $x \in K$ *and* $l_L(x) = \perp$.

We call L *the* left-hand side*,* R *the* right-hand side *and* K *the* interface *of* r*. If* L *and* R *are totally labelled conditions (1) and (2) are trivially satisfied.*

Example 2.3
The rule shown in figure 2.3 consists of a left-hand side with the graph from example 2.2, an interface of a single unlabeled node and a left-hand side of a single node labeled c as seen in figure 2.3.

The rule changes the first node's label from a to c and deletes the edge and the second node.

To apply a rule to a graph we have to find a *match*, that is we find a morphism from the left-hand side of the rule to the target graph.

Not all morphisms from the left-hand side of a rule into a target graph allow an application of the rule. The result of a derivation should still be a graph, i.e. all

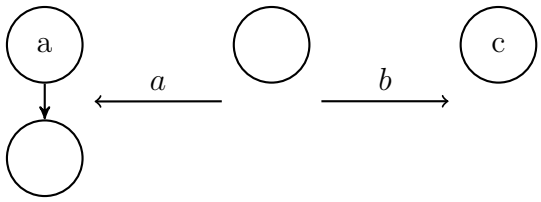


Figure 2.3: Example of a Rule

edges should have a source and a target. For this reason we forbid the application of a rule, if there are edges in the target graph that coincide with nodes that will be deleted by the rule.

Definition 2.4 (Dangling Condition)
Given a rule $r = \langle L \leftarrow K \rightarrow R \rangle$*, an injective morphism* $g : L \rightarrow G$ *satisfies the* dangling condition *if no edge in* $E_G \setminus g_E(E_L)$ *is incident to a node in* $g_V(V_L \setminus V_K)$*.*

If the morphism from example 2.2 were to map the node s_2 in the left-hand side of the rule from example 2.3 to v_4 for example, the subsequent deletion of that node would leave two edges *dangling* without a source.

If we have a match that satisfies the dangling condition, we can construct a new graph using the three parts of a rule. We remove all elements (and labels) that are part of the left-hand side but not of the interface and add all the elements (and labels) that are part of the right-hand side but not part of the interface.

Definition 2.5 (Direct Derivation)
Given a rule $r = \langle L \leftarrow K \rightarrow R \rangle$ *and a graph* G *with an injective graph morphism* $g : L \rightarrow G$ *satisfying the dangling condition, called the* match*, a* direct derivation *(or* graph transformation*) is the transformation of* G *into a graph* H *constructed as follows:*

$$\begin{array}{ccccc} L & \longleftarrow & K & \longrightarrow & R \\ {\scriptstyle g}\downarrow & (1) & \downarrow & (2) & \downarrow \\ G & \longleftarrow & D & \longrightarrow & H \end{array}$$

(1) Delete $g(L \setminus K)$ *and the labels of* $g(K_\perp)$ *from* G *yielding a graph* D*,*

(2) Add $R \setminus K$ *and the labels of* $l_R(K_\perp)$ *to* D *yielding a graph* H*,*

where $K_\perp$ is the set of all unlabelled nodes in K.

We write $G \Rightarrow_{r,g} H$ if there exists such a direct derivation.

Example 2.4
The direct derivation in figure 2.4 is the application of the rule from example 2.3, using the morphism from example 2.2 as the match g. The rule removes the node v_2 and changes the label of v_1 from a to c.

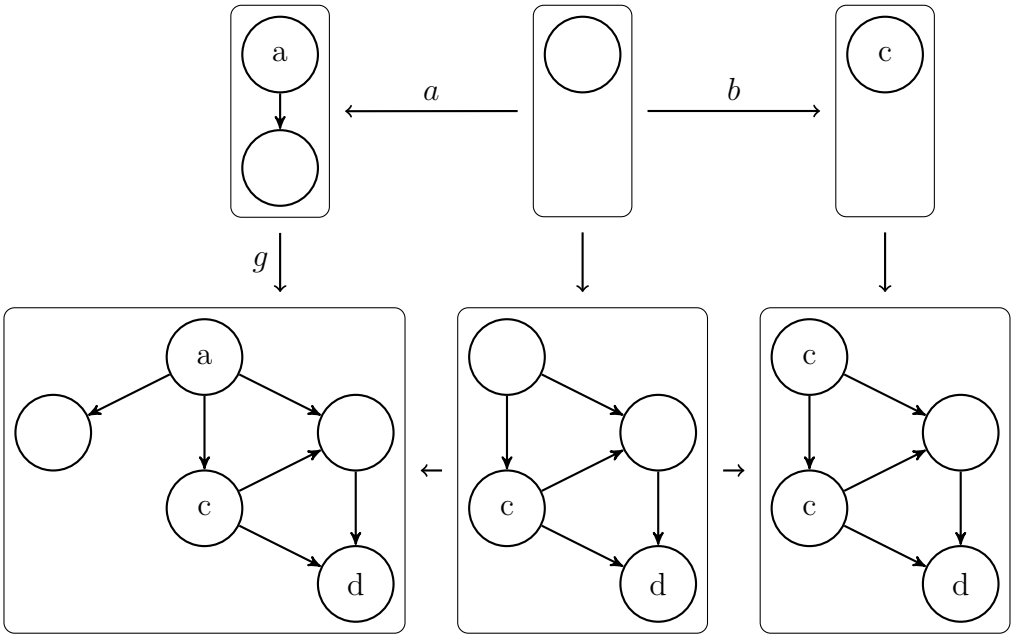


Figure 2.4: Example of a Direct Derivation

2.2 Category Theory

As mentioned in section 2.1 graph transformation is most commonly defined in terms of category theory. In the following we briefly cover some definitions from category theory that will be used throughout the remainder of the thesis. For an in-depth introduction see for example [Sim11] or [AHS09].

Definition 2.6 (Category)
A category *is a system* $A = (obj, morph, src, trgt, id, \circ)$ *consisting of a collection* obj *of* objects*, a collection* $morph$ *of* morphisms[2]*, two assignments* $src, trgt : morph \to obj$*, an assignment* $id : obj \to morph$*, and a partial composition* $\circ : morph \times morph \to morph$*, subject to the following conditions:*

- *(Associativity) Composition of morphisms is associative; i.e. for morphisms* $f : A \to B$*,* $g : B \to C$ *and* $h : C \to D$ *the equation* $h \circ (g \circ f) = (h \circ g) \circ f$ *holds.*
- *(Identity) Identities act as such with respect to composition; i.e. for a morphism* $f : A \to B$*, we have* $id_B \circ f = f$ *and* $f \circ id_A = f$*.*

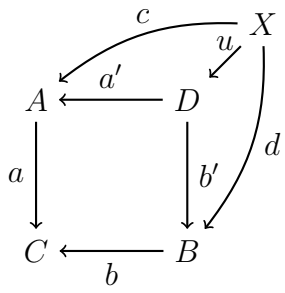
Example 2.5
Partially labelled graphs and graph morphisms, as presented in section 2.1 constitute a category $\texttt{Graphs}_{\texttt{PLG}}$*, where composition of morphisms is defined componentwise as function composition.*

The definitions for pullbacks and pushouts presented here are closer to those in [EEPT06] than in the books mentioned above, this is because these definitions do not require additional definitions before defining pullbacks and pushouts.

A useful intuition for a pullback is that of the intersection of two objects.

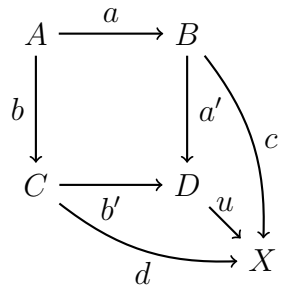
Definition 2.7 (Pullback)
Given morphisms $a : A \to C$ *and* $b : B \to C$ *a* pullback (D, a', b') *over* a *and* b *is defined by a pullback object* D *and morphisms* $a' : D \to A$ *and* $b' : D \to B$ *with* $a \circ a' = b \circ b'$ *such that the following universal property holds: for all objects* X *with morphisms* $c : X \to A$ *and* $d : X \to B$ *with* $a \circ c = b \circ d$*, there is a unique morphism* $u : X \to D$ *such that* $a' \circ u = c$ *and* $b' \circ u = d$*.*

[2] or *arrows*, but more commonly called *morphisms* in the context of graphs and therefore the remainder of this thesis.



A useful intuition for a pushout is an object that represents a union of two objects.

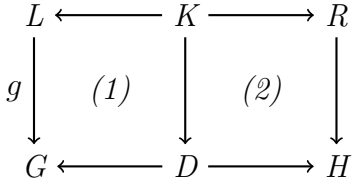
Definition 2.8 (Pushout)
Given morphisms $a : A \to B$ and $b : A \to C$ a pushout *(D, a', b') over a and b is defined by a pushout object D and morphisms $a' : B \to D$ and $b' : C \to D$ with $a' \circ a = b' \circ b$, such that the following universal property holds: for all objects X with morphisms $c : B \to X$ and $d : C \to X$ with $c \circ a = d \circ b$ there is a unique morphism $u : D \to X$ such that $u \circ a' = c$ and $u \circ b' = d$.*



A pushout is said to be natural*, if it is also a pullback.*

Now we can give the category-theoretic definition of a direct derivation:

Fact 2.1 (Direct Derivation[HP02])
A direct derivation *from a graph G to a graph H via a rule $r = \langle L \leftarrow K \rightarrow R \rangle$ consists of two natural pushouts (1) and (2) as given in the diagram below:*



2.3 Algebraic Specifications

Many existing attribution concepts for graph transformation use algebraic specifications. Types and the operations on them are defined in terms of an algebra, which is then used as a base for constructing attributed graphs. An introduction to algebraic specifications can be found in [EM85].

We start with the definition of a system of *sorts*, *constants* and *operations*, which form a *signature*. The sorts will later be used as the types of attributes, while the operations are used if computation over attributes is desired.

Definition 2.9 (Signature)
A signature $\Sigma = (S, OP)$ *consists of a set* S *of* sorts *and a set* OP *of* constant and operation symbols. OP *is the union of pairwise disjoint subsets:* C_s*, the set of* constant symbols *of sorts* $s \in S$ *and* $OP_{w,s}$ *the set of* operation symbols *with argument sort* $w \in S^+$ *and range sort* $s \in S$.

The constants in a signature can also be seen as operations without parameters.

Notation
We also use $OP_{\lambda,s} = C_s$ *with the empty string* $\lambda \in S^*$ *and refer to* OP *by* $OP_{w,s}$ *with* $w \in S^*$ *and* $s \in S$.

For the purposes of this thesis we will use a signature that specifies Boolean values, natural numbers and strings and some operation symbols on these sorts.

Example 2.6
We construct a signature $\Sigma = (S, OP)$ *such that* $S = \{Bool, Nat, String, Alphabet\}$*, that has Boolean values, natural numbers and strings as sorts. The constant and operation symbols for Boolean values Bool are* $OP_{\lambda,Bool} = \{true, false\}$*,* $OP_{Bool,Bool} = \{\neg\}$ *and* $OP_{BoolBool,Bool} = \{\wedge, \vee, \Rightarrow, \Leftrightarrow\}$*. The constant and operation symbols for natural numbers Nat are* $OP_{\lambda,Nat} = \{zero\}$*,* $OP_{Nat,Nat} = \{succ\}$ *and* $OP_{NatNat,Nat} = \{+, -, *, \div\}$*. The constant and operation symbols for strings String are* $OP_{\lambda,String} = \{epsilon\}$*,* $OP_{Alphabet,String} = \{make\}$ *and* $OP_{String,String} = \{concat\}$.

While the definition of sorts provides for types, we still need values for attributes that have these types. An *algebra* defines the sets of values and the concrete operations for the sorts and operation symbols specified in the signature.

Definition 2.10 (Algebra)
A Σ-algebra $A = (S_A, OP_A)$ *of a signature* $\Sigma = (S, OP)$*, consists of two families* $S_A = (A_s)_{s \in S}$ *and* $OP_A = (op_A)_{op \in OP}$ *where:*

(1) A_s *are sets for all* $s \in S$*, called* base sets *of* A*.*

(2) op_A *are elements* $op_A \in A_s$ *for all constant symbols* $op \in OP_{\lambda,s}$*, called* constants *of* A*.*

(3) $op_A : A_{s1} \times \cdots \times A_{sn} \to A_s$ *are functions for all operation symbols* $op \in OP_{s1\ldots sn,s}$ *and* $s1 \ldots sn \in S^+$*,* $s \in S$*, called* operations *of* A*.*

We construct an algebra from the example signature below.

Example 2.7
We construct a Σ*-algebra* $A = (S_A, OP_A)$ *such that* A_{Bool}*,* A_{Nat}*,* A_{String} *and* $A_{Alphabet}$ *are the sets of Boolean values, natural numbers, character strings and characters respectively,* $\neg_A, \wedge_A, \vee_A, \Rightarrow_A, \Leftrightarrow_A$ *are the usual Boolean operations,* $zero_A$*,* $succ_A$ *and* $+_A$ *are* 0*, the successor function and addition for natural numbers and* $epsilon_A$*,* $make_A$ *and* $concat_A$ *are the empty string, the construction of a string from a character and concatenation of strings respectively.*

Assumption 2.1
Whenever we refer to a standard algebra *in the remainder of this thesis,* A *from example 2.7 above is that algebra.*

For more involved computation over the values of attributes we would like to construct *terms* and use *variables* in the usual manner.

Definition 2.11 (Variables and Terms)
Let $\Sigma = (S, OP)$ *be a signature and* X_s *for each* $s \in S$ *a set, called set of variables of sort* s*. We assume that these sets* X_s *are pairwise disjoint and also disjoint with* OP*. Then:*

(1) The union $X = (X_s)_{s \in S}$ *is called the set of* variables *of* Σ*.*

(2) The sets $T_{OP,s}(X)$ *of* terms *of sort* s *are defined by*

1. *The* basic terms $x, c \in T_{OP,s}(X)$ *for all* $x \in X_s$ *and all* $c \in OP_{\lambda,s}$*.*
2. *The* composite terms $op(t_1, \cdots, t_n) \in T_{OP,s}(X)$ *for all operation symbols* $op \in OP_{s_1\ldots s_n,s}$ *and all terms* $t_1 \in T_{OP,s_1}(X), \cdots, t_n \in T_{OP,s_n}(X)$*.*

The set of all terms over Σ *and* X *is denoted by* $T_\Sigma(X)$*.*

These terms can be *evaluated* by successively replacing constants and operations with their values. If variables are used, an *assignment* maps these variables to values, and an *extended assignment* also evaluates the resulting variable-free term.

Definition 2.12 (Evaluation of Terms)
Let $T_\Sigma(X)$ be the set of terms of a signature $\Sigma = (S, OP)$ and A a Σ-algebra. The evaluation *$eval : T_\Sigma(x) \to A)$ is recursively defined by:*

1. *$eval(op) = op_A$ for all constant symbols $op \in OP_{\lambda,s}$.*
2. *$eval(op(t_1, \cdots, t_n))$=$op_A(eval(t_1), \cdots, eval(t_2))$ for all $op(t_1, \cdots, t_n) \in T_\Sigma(X)$.*

Given a set of variables X for $\Sigma = (S, OP)$ and an assignment *$\alpha : X \to A$ with $\alpha(x) \in A_s$ for $x \in X_s$ and $s \in S$, the* extended assignment*, or simply* extension *$\hat{\alpha} : T_\Sigma(X) \to A$ of the assignment α is recursively defined by*

1. *$\hat{\alpha}(x) = \alpha(x)$ for all variables $x \in X$*
2. *$\hat{\alpha}(op) = op_A$ for all constant symbols $op \in OP_{\lambda,s}$*
3. *$\hat{\alpha}(op(t_1, \cdots, t_n)) = op_A(\hat{\alpha}(t_1), \cdots, \hat{\alpha}(t_n))$ for all $op(t_1, \cdots, t_n) \in T_\Sigma(X)$.*

If t is a variable-free term, then $\hat{\alpha}(t)$ is denoted by t_A.

3 Existing Attribution Concepts

This chapter covers existing concepts for the attribution of graphs. We start with a brief survey of various concepts, then examine two of these, Typed Attributed Graphs by Ehrig et al. and the graphs used in the graph programming language GP by Plump in more detail. To this end a running example is introduced in section 3.1.

All of the attribution concepts presented here use an algebra to specify types and values of their attributes. They differ in the way these attribute values are attached to a graph and how either the graph or the attribute values are represented.

Attributes Encoded in the Graph There are several approaches to attribution that represent attribute values by elements of the graph. In Typed Attributed Graphs [EEPT06] graphs are extended with a special node set that holds a node for every element of an algebras base set. To facilitate attribution of edges Typed Attributed Graphs allow attribution edges to have an edge as a source. This approach is extended in [Ore11] to *symbolic attributed graphs*, which allow the use of variables in Typed Attributed Graphs and further allows to attach constraints over these variables to a graph.

In the same vein, [KR12] also represents attribute values by attribute nodes, although attribution edges between edges and attribute nodes are not allowed. The idea is taken a step further by also encoding the operations specified by the algebra in the graph.

Typed Attributed Graphs are covered in more detail in section 3.2.

Attributes Encoded in Labels The graph programming language GP [Plu09], while not presented as an attribution concept, does allow users to attach several values to both nodes and edges through its rule schemata. Rule schemata allow the decomposition of labels into these values, which can be seen as a form of attribution.

The graphs used in GP are covered in more detail in section 3.3.

Algebra Interpreted as a Graph The approach of [LKW93] can be seen as the reverse of the previous approaches. Instead of including the elements of the base sets of an algebra into a graph, the graph itself is encoded in an algebra. This algebra additionally specifies types for attributes and operations that allow these attributes to be attached to nodes or edges.

Attribution of Arbitrary Categories An arbitrary category (graphs for our purposes) can be used as a base for a $\mathcal{W}$-adhesive category [Gol12]. A set of attribution types determines what attributes are attached to a given node or edge.

3.1 Running Example

This section introduces the running example used in the following sections and chapters.

The example should contain attributed nodes and attributed edges with multiple attributes. An overly simple example with a single attribute of a single type per node or edge risks excluding potential problems with ambiguous attribute identifiers.

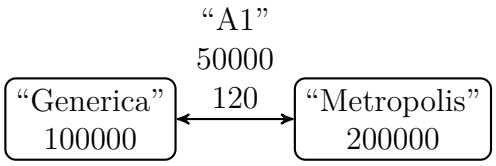


Figure 3.1: The Running Example: Cities and Roads

To this end, the example graph consists of a number of cities and roads connecting these cities. Nodes have a city name and population as attributes, edges are attributed with a name, a length and a speed limit.

Since the road connections are undirected and the roads attributes will be the same for both directions unify the two edges between cities and only show their common attributes once.

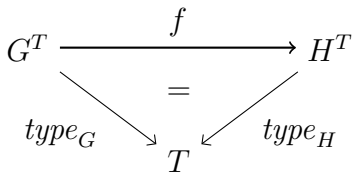
3.2 Typed Attributed Graphs a la Ehrig et al.

Typed Attributed Graphs [EEPT06] are based on graphs *without* labels. Whenever we refer to a graph in this section we therefore assume that it is unlabelled.

To nonetheless enable us to attach names to nodes and edges, graphs can be *typed* instead. In a typed graph edges and labels are mapped to elements of a type graph with a morphism. The name of the object in the type graph that an object is mapped to determines its type.

Definition 3.1 (Typed Graph)
A type graph *is a graph* $TG = (V_{TG}, E_{TG}, s_{TG}, t_{TG})$. V_{TG} *and* E_{TG} *are called the vertex and the edge type alphabets, respectively. A tuple* $(G, type)$ *of a graph* G *together with a graph morphism* $type : G \to TG$ *is called a* typed graph.

Given typed graphs $G^T = (G, type_G)$ *and* $H^T = (H, type_H)$, *a* typed graph morphism $f : G^T \to H^T$ *is a graph morphism* $f : G \to H$ *such that* $type_H \circ f = type_G$.



Example 3.1
An example of a typed graph is shown in figure 3.2.

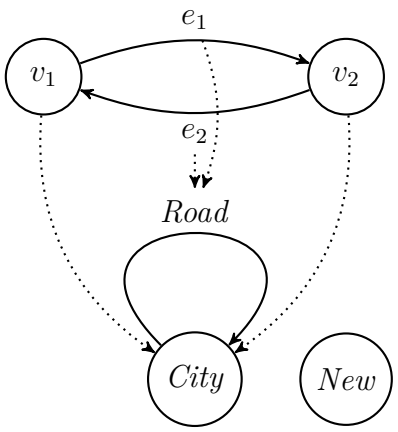


Figure 3.2: A Typed Graph

Note that typed graphs also allow restriction as to which types can be assigned to which edges. In example 3.1, the node v_1 could not be assigned the type *New*, since the typing morphism, being a graph morphism, must preserve sources and targets of edges.

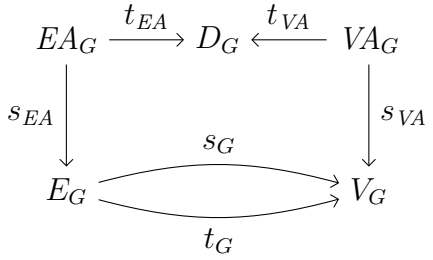
In earlier approaches [HKT02] attribution was done by introducing edges to attribute nodes. In these approaches attribution was limited to nodes. To accommodate attributes for edges as well as nodes the definition of graphs is extended to allow for special edges between edges and attribute nodes.

Definition 3.2 (E-Graph)
An E-graph $G = (V_G, D_G, E_G, VA_G, EA_G, (s_j, t_j)_{j \in \{G,NA,EA\}})$ *consists of the sets*

- V_G *and* D_G*, called the* graph nodes *and* data nodes*;*
- E_G*,* VA_G *and* EA_G*, called the* graph edges*,* node attribute edges *and* edge attribute edges*, respectively;*

and the source and target functions

- $s_G, t_G : E_G \to V_G$ *for graph edges;*
- $s_{NA} : VA_G \to V_G$ *and* $t_{NA} : VA_G \to D_G$ *for node attribute edges;*
- $s_{EA} : EA_G \to E_G$ *and* $t_{EA} : EA_G \to D_G$ *for edge attribute edges:*



We introduce a new set of nodes D_G specifically for the attribute values and two new sets of nodes VA_G and EA_G for edges for the attribution of nodes and edges respectively.

Example 3.2
Figure 3.3 shows the city example as an E-graph. The dashed edges attach attribute nodes to the graph nodes $n1$ *and* $n2$*. Note that the attribute nodes have no defined type and the attribution edges have no names.*

We define morphisms for E-graphs that also preserve the sources and targets of attribution edges.

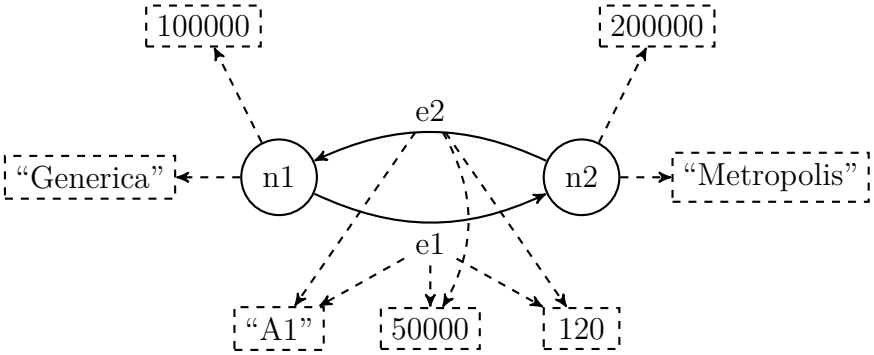
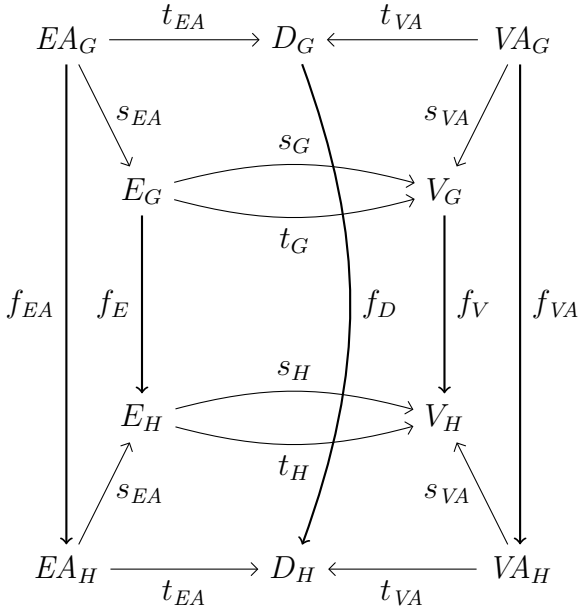


Figure 3.3: E-Graph of the City Example

Definition 3.3 (E-Graph Morphism)
Consider the E-graphs G and H. An E-graph morphism $f : G \to H$ *is a tuple* $(f_V, f_D, f_E, f_{NA}, f_{EA})$ *with* $f_i : i_G \to i_H$ *for* $i \in \{V, D, E, NA, EA\}$ *such that* f *commutes with all source and target functions, for example* $f_V \circ s_G = s_H \circ f_E$*, as seen in the diagram below.*



To construct an *attributed graph* we combine an E-graph with an algebra over a data signature Σ. The base sets corresponding to the sorts of the signature are used as the attribute values and together form D_G.

Definition 3.4 (Attributed Graph and Morphism)
Let $\Sigma = (S_D, OP_D)$ 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 Σ-algebra D such that $\dot{\bigcup}_{s \in S'_D} D_s = D_G$, where $\dot{\cup}$ is a disjoint union.*

For two attributed graphs $GA = (G, E)$ and $HA = (H, F)$, an attributed graph morphism *$f : GA \to HA$ is a pair $f = (f_G, f_A)$ with an E-graph morphism $f_G : G \to H$ and an algebra homomorphism $f_A : E \to F$ such that (1) commutes for all $s \in S'_D$, where the vertical arrows below are inclusions:*

$$\begin{array}{ccc} E & \xrightarrow{f_A} & F \\ \downarrow & (1) & \downarrow \\ D_G & \xrightarrow[f_{G,D}]{} & D_H \end{array}$$

Example 3.3
The E-graph in example 3.2 is also an attributed graph, given an appropriate algebra and assuming the existence of additional nodes for its base sets.

While attributed graphs already allow us to attach an arbitrary number of values of different sorts to both nodes and edges we have no way to refer to specific attributes. To rectify this we use typing, where an E-graph acts as the type graph. We introduce an attribute node for each sort of the algebra and identify all values with their respective sorts via the typing morphism. For each node or edge type we specify in the type graph, we add attribute edges to the attribute nodes we created previously. We can name these edges and thus the attributes in the typed graph.

Definition 3.5 (Typed Attributed Graph)
Given a data signature Σ, an attributed type graph *is an attributed graph $ATG = (TG, Z)$, where Z is the final Σ-algebra. A* typed attributed graph *(AG, t) over ATG consists of an attributed graph AG together with an attributed graph morphism $t : AG \to ATG$.*

Given two typed attributed graphs $(AG, t_G$ and $(AH, t_H$ over ATG, a typed attributed graph morphism *$f : (AG, t_G) \to (AH, t_H)$ is an attributed graph morphism $f : AG \to AH$ such that $t_H \circ f = t_G$.*

Lemma 3.1
Typed Attributed Graphs and their morphisms form a category $\texttt{Graphs}_{\texttt{TAG}}$*, where composition of morphisms is componentwise function composition.*

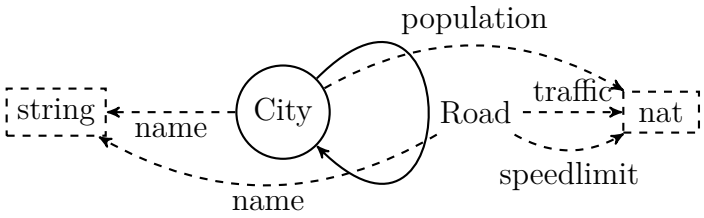


Figure 3.4: Attributed Type Graph of the City Example

Example 3.4
To construct a Typed Attributed Graph from example 3.2 we first construct an appropriate type graph, such as the attributed graph in figure 3.4 and then combine the two using a typing morphism as seen in figure 3.5. For readability the typing morphism is only shown for nodes.

Since figures even for small typed attributed graphs become rather large a simplified notation is provided, as seen in figure 3.6. The figure shows the same graph as figure 3.5, but does not depict the type graph and typing morphism as separate from the graph. Instead their existence is implied by the graphs annotation, such as the name "n1:City" instead of simply "n1" which tells us that the node "n1" is mapped to the type "City" by the typing morphism.

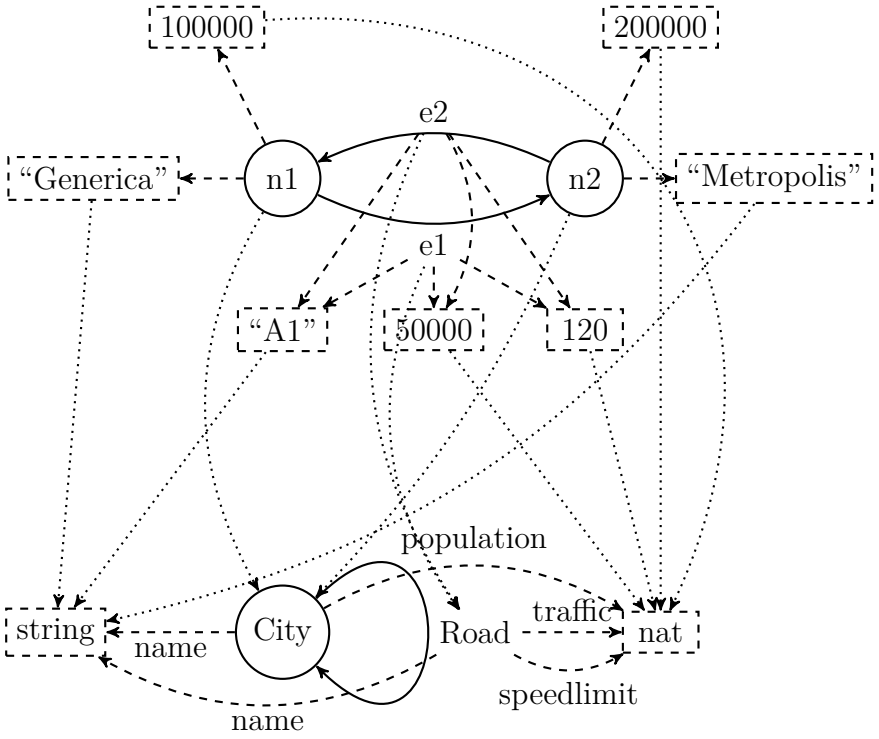


Figure 3.5: Typed Attributed Graph of the City Example

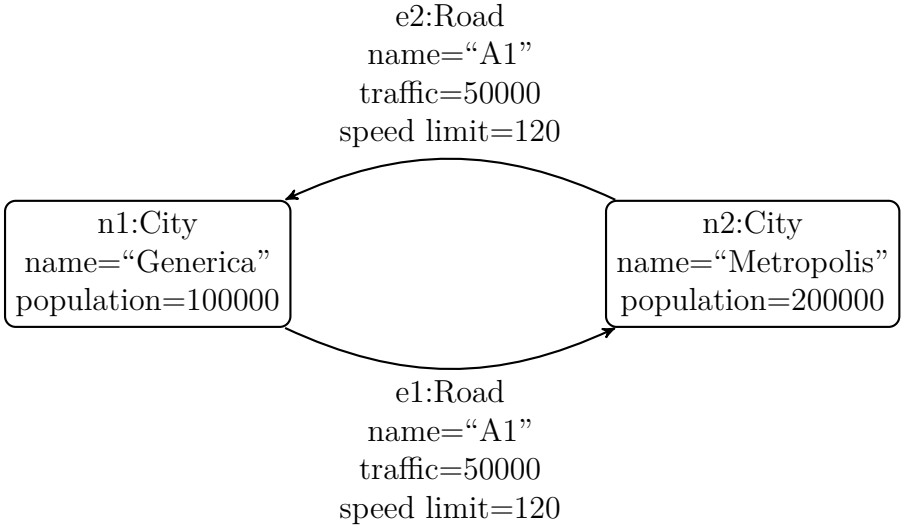


Figure 3.6: Simplified Typed Attributed Graph of the City Example

3.3 Attributed Graphs a la Plump

While the graph programing language GP [Plu09] uses attributes for the nodes and edges of its graphs it does not rely on any of the concepts for the attribution of graphs that were in use previously. GP uses partially labelled graphs with relabelling, as introduced in section 2.1, but instead of changing the definition of graphs to accommodate attributes GP uses rule schemata to similar effect.

Remark
There is a newer version of GP, called GP2 [Plu11]. It does not change the way it handles attribution beyond changing the preset types of attributes that are allowed.

The definition of graphs and graph morphisms for GP follow those in [HP02] as covered earlier in section 2.1, except as follows:

Definition 3.6
Let $\mathbb{S}$ be the set of strings over $\{a, b, \cdots, z\}$. The labels $\mathcal{L}$ in the previous definitions are concretized with the set $(\mathbb{Z} \cup \mathbb{S})^+$ of all nonempty sequences over integers and character strings.

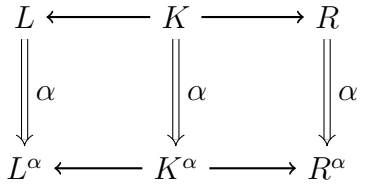
The following definitions are taken from [PS04].

Assumption 3.1
In the following we assume the use of a signature Σ and corresponding Σ-algebra A as defined in the examples in section 2.3.

Since rules for partially labelled graphs do not allow a decomposition of labels, we need some way to access the individual attribute values that are encoded in the labels. To this end *schemata* are introduced that allow the use of variables in labels. Changing a specific attribute is just a matter of replacing this variable with a value or a term in the right-hand side of the rule schema.

Definition 3.7 (Rule Schema and Instances of Rule Schema)
A rule $r = \langle L \leftarrow K \rightarrow R \rangle$ is called a schema *if L, K and R are graphs over $T_\Sigma(X)$.*

Given a graph G over $T_\Sigma(X)$ and an assignment $\alpha : X \rightarrow A$, the instance G^α *of G is the graph over S_A obtained from G by replacing the labelling functions l_G with $\hat{\alpha} \circ l_G$, where $\hat{\alpha}$ is the extension of α. The instance of a rule schema $r = (L \leftarrow K \rightarrow R)$ is the rule $r^\alpha = (L^\alpha \leftarrow K^\alpha \rightarrow R^\alpha)$.*



Besides computation on attributes GP also offers some flow control. To this end rules can be equipped with negative application conditions similar to those in [HHT96]. In later versions the conditions are replaced with nested application conditions, as presented in [HP09].

Definition 3.8 (Conditional Rule)
A conditional rule $q = (r, M)$ *consists of a rule* $r = (L \leftarrow K \rightarrow R)$ *and a set* M *of graph morphisms such that* $M \subseteq \{g : L \rightarrow G | G$ *is a graph over the base sets* S_A *and* g *is a match for* $r\}$.

With the addition of a Boolean term to a rule schema we get a *conditional rule schema*. The term may contain the operation symbol *edge* in addition to those in the signature Σ, this symbol is used to check for the existence of edges.

Definition 3.9 (Conditional Rule Schema)
Given a rule schema $(L \leftarrow K \rightarrow R)$, *extend the signature* Σ *to* $\Sigma^L = (S^L, OP^L)$ *by* $S^L = S \cup Node$, $OP^L_{\lambda,Node} = V_L$, $OP^L_{NodeNode,Bool} = \{Edge\}$, $OP^L_{w,s} = OP_{w,s}$ *if* $w \in S^*$ *and* $s \in S$, *and* $OP^L_{w,s} = \emptyset$ *otherwise. Then a term* c *in* $T_{OP^L,Bool}(X)$ *is a* condition *and* $\langle (L \leftarrow K \rightarrow R), c \rangle$ *is a* conditional rule schema.

Instantiating a conditional rule schema is done by instantiating the rule schema and then evaluating the condition. The instances can only be applied to a graph when the condition evaluates to *true*.

Definition 3.10 (Instance of a Conditional Rule Schema)
Given a conditional rule schema $r = \langle (L \leftarrow K \rightarrow R), c \rangle$, *an assignment* $\alpha : X \rightarrow A$ *and a graph morphism* $g : L^\alpha \rightarrow G$ *with* G *a graph over* S_A, *define the extension* $\alpha_g : T_{\Sigma^L}(X) \rightarrow A$ *as follows:*

(1) $\alpha_g(x) = \alpha(x)$ *and* $\alpha_g(c) = c_A$ *for all variables* x *and all constants* c *in* Σ[1].

(2) $\alpha_g(\textbf{\textit{edge}}(v, w)) = \begin{cases} \textbf{\textit{tt}} & \text{if there is an edge in } G \text{ from } g(v) \text{ to } g(w) \\ \textbf{\textit{ff}} & \text{otherwise} \end{cases}$

[1] Note that α_g is undefined for all constants in $OP^L_{\lambda,Node}$.

(3) $\alpha_g(op(t_1, \cdots, t_n)) = op_A(\alpha_g(t_1), \cdots, \alpha_g(t_n))$ *for all* $op(t_1, \cdots, t_n) \in T_{OP^L, S^L}(X)$ *with* $op \in OP$.

Then the instance r^α *of* r *is the conditional rule* $\langle (L^\alpha \leftarrow K^\alpha \rightarrow K^\alpha), M \rangle$ *where* $M = \{g : L^\alpha \rightarrow G | G$ *is a graph over* S_A, g *is a match and* $\alpha_g(c) = \boldsymbol{tt}\}$.

Example 3.5
Figure 3.7 shows the running example as it could be used in GP. Attributes are encoded in the labels and separated by "_".

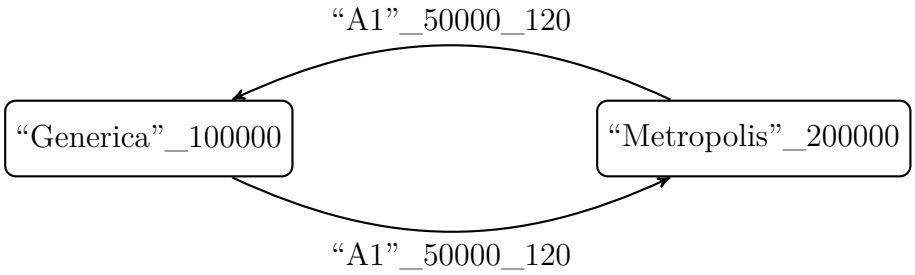


Figure 3.7: Attributed Graph a la Plump of the City Example

As can be seen from the above definitions GP is primarily concerned with computation over attributes in graph programs. It does also, however, introduce an interesting concept for handling attribution in graphs.

4 Comparison of Attribution Concepts

In this chapter, we compare the two concepts covered in greater depth in the previous chapter, Typed Attributed Graphs by Ehrig et al. and the graphs used in the graph programming language GP by Plump. In the remainder of this chapter we refer to the two concepts as *Ehrig-attributed graphs* and *Plump-attributed graphs* respectively. We start with looking at a set of example rules and explore how both concepts work and what issues can arise from their usage. Additionally, transformations between the two concepts are introduced. The chapter concludes with a short summary of the results of the comparison.

4.1 Example Rules

We are interested not only in the theoretical properties of the two concepts, but also the consequences of *using* them. Therefore we start by comparing the implementation of a set of example rules in both concepts and the possible pitfalls.

4.1.1 Changing an Attribute Value

The first example is the change of an attribute value. Our main interest here is to observe the application of rules and how they interact with attributes.

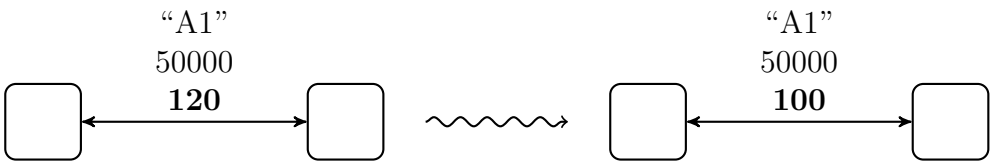


Figure 4.1: Changing an Attribute Value

Figure 4.1 shows such a change, the value “120” in the graph on the left is changed to “100” on the right.

4.1.2 Removing an Attribute

For our second example we remove an attribute entirely, as shown in figure 4.2. We could instead add a new attribute, with broadly similar results.

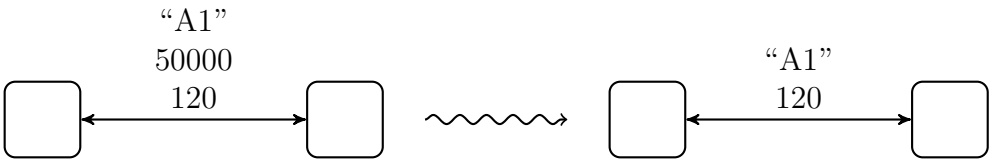


Figure 4.2: Removing an Attribute

4.2 Ehrig-Attributed Graphs

In this section, we attempt to translate the above examples into Ehrig-attributed graphs and rules.

4.2.1 Changing an Attribute Value

Figure 4.3 shows the example rule as implemented in Ehrig-attributed graphs, both in the simplified notation and in full.

Changing an attribute value in Ehrig-attributed graphs means removing an attribution edge and creating another. Note that the attribute nodes 120 and 100 in Figure 4.3 are not removed or added but are implicitly part of all of the graphs of that rule. Additionally, while the figure shows three distinct type graphs, these are only included for readability. The three graphs actually share a single type graph.

Attributes are multisets One side-effect of using edges to implement attribution is the nature of the resulting attributes. These attributes do not always have a single value, but can instead have several different values, no value or even multiples of the same value.

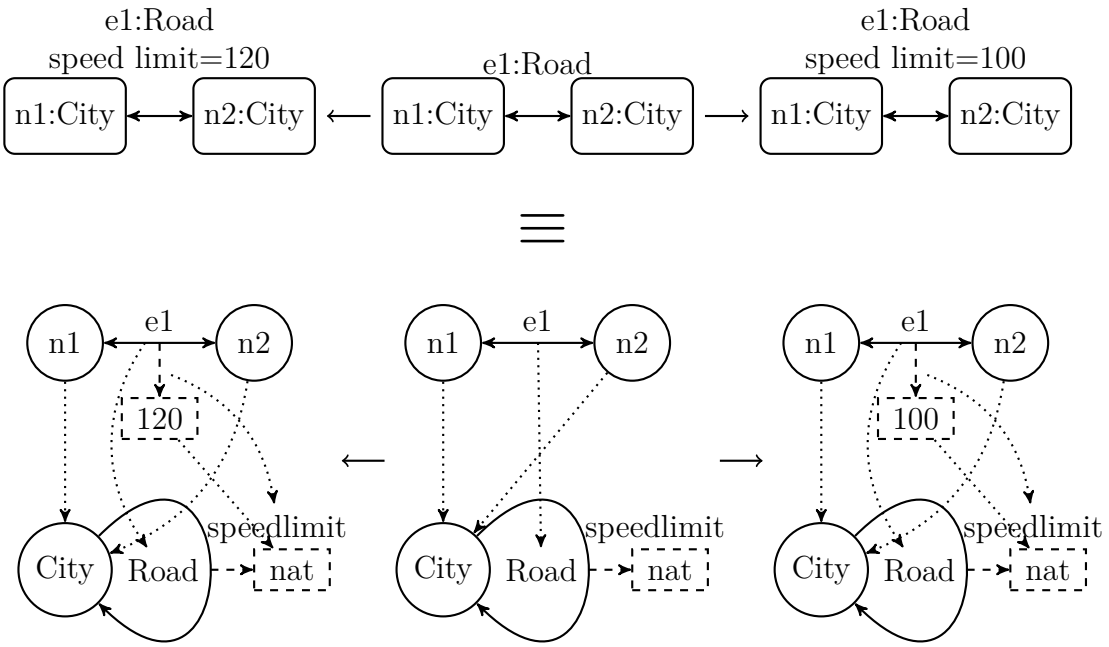


Figure 4.3: Changing an Attribute Value with Ehrig-Attributed Graphs

4.2.2 Removing an Attribute

While this is possible with Ehrig-attributed graphs it is necessary to replace the node with an entirely new one. Since all of the graphs that are part of a rule in Ehrig-attributed graphs share a single type graph and the morphisms that make up the type graph must commute with the typing morphism it is impossible to change the type of a node with a rule. As it is impossible to change that type of a node or edge, adding or removing attributes entirely is also impossible.

4.3 Plump-Attributed Graphs

In this section, we attempt to translate the examples from section 4.1 into Plump-attributed rules and rule schemata.

4.3.1 Changing an Attribute Value

Figure 4.4 shows an implementation of the first example rule in Plump-attributed graphs. Note that the figure does not show a rule, but a rule schema instead.

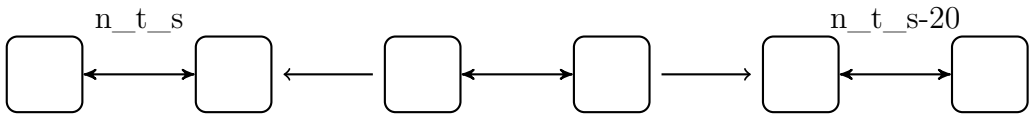


Figure 4.4: Changing an Attribute Value with Plump-Attributed Graphs

Attributes are encoded in labels Plump-attributed graphs are partially labeled graphs with a label alphabet derived from the desired types for attributes. It is not possible to match against a node with only one out of several attributes specified as the whole label is used for finding a match. A rule or rule schema must therefore always contain *all* attributes of a node or edge.

Remark
GP2 [Plu11] eases this restriction a little by allowing lists of arbitrary length in a schema.

The example above uses a schema to work around this limitation by including variables in both the left-hand side and the right-hand side of the schema that do not change for all attributes except the attribute we want to change.

4.3.2 Removing an Attribute

Plump-attributed graphs allow the removal or addition of attributes, as shown in figure 4.5. A rule that changes attribute values always removes all attributes before adding them back (possibly with different values). If an attribute should be removed it is simply not added back to the label when applying the rule.

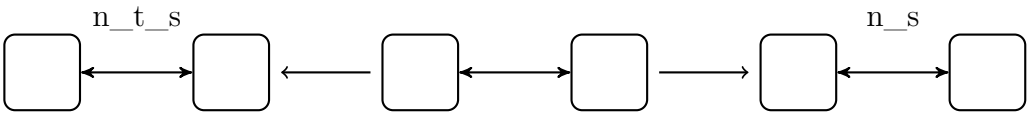


Figure 4.5: Removing an Attribute with Plump-Attributed Graphs

Consequences of attribute addition or removal Since a rule always matches against *all* attributes of a node or edge, addition or removal of attributes prevents or enables the application of other rules. Rules can thus have effects on the applicability of other rules, even if they operate on disjoint sets of attributes.

Other rules may later add an attribute that will occupy the same "place" in the sequence. This might re-enable the application of rules without providing suitable values for those rules.

4.4 Transformations between Ehrig- and Plump-Attributed Graphs

Now that the consequences of using either Ehrig- or Plump-attributed graphs have been explored, we investigate if both concepts are equally powerful. In particular we attempt transformations between both concepts.

It is easy to see that Ehrig-attributed graphs, where the attributes are inherently multisets can not always be represented by Plump-attributed graphs. On the other hand Ehrig-attributed graphs do not support some of the flow control mechanism introduced with GP. We therefore restrict Ehrig-attributed graphs and do not consider conditional schemata.

Assumption 4.1
We restrict Ehrig-attributed graphs as follows:

- Σ *is restricted to the standard algebra.*
- *The multiset-nature of attributes is* not *used, instead we assume a single value per attribute.*
- *Attributes have* at least one value*; but waive this requirement for rule interfaces.*

Additionally, we will not *consider conditional schemata for Plump-attributed graphs, although a transformation of these conditions into nested conditions [HP09] might be possible if these conditions are expanded to allow constraints over attribute values.*

These assumptions allow a single attribute in an Ehrig-attributed graph to be represented by a single attribute in a Plump-attributed graph.

The assumptions described above are manageable in many cases and a behavior where a single attribute represents a single value is indeed the more intuitive one. If sets or multisets are required, they can still be included in the algebra used by either concept.

Formal definitions of the restriction follow:

Definition 4.1 (Restriction to Single-Value Attributes)
A Typed Attributed Graph is said to be restricted to single-value attributes *if, for every node $v \in V_G$ and every edge $e \in E_G$ there is* at most *one edge $e_a \in E_{NA} \cup E_{EA}$ for each edge e_t between $t(v)$ ($t(e)$) and a node $a_t \in V_D$ in the type graph.*

Definition 4.2 (Restriction to Nonempty Attributes)
A Typed Attributed Graph is said to be restricted to nonempty attributes *if, for every node $v \in V_G$ and every edge $e \in E_G$ there is* at least *one edge $e_a \in E_{NA} \cup E_{EA}$ for each edge e_t between $t(v)$ ($t(e)$) and a node $a_t \in V_D$ in the type graph.*

Rules for graphs with single-value, nonempty attributes need to be restricted as well, if we want the results of rule applications to stay restricted in the same way. The left- and right-hand side of a rule need both restrictions, while the interface is restricted only to single-value attributes.

The biggest obstacle for a transformation is the type graph in Ehrig-attributed graphs. If we transform an Ehrig-attributed graph along with a set of rules, these rules should still be bound to the restrictions that typing enforces in Ehrig-attributed graphs. Conversely a Plump-attributed graph that is transformed will have to be typed, since typing for attributes cannot be done without introducing types for the nodes and edges.

To enable a smooth transformation between both types of graphs we will first transform Plump-attributed graphs by introducing a type name (as a string) as their first attribute for every node and edge. These type names will be generated based on the number and types of attributed present in Plump-attributed graphs and based on the type names in the type graph in Ehrig-attributed graphs.

Definition 4.3 (Quasi-Typed Plump-Attributed Graphs)
A Plump-attributed graph G_p is quasi-typed *if, for every two sequences of attribute types ts and $t's'$ in its nodes (edges), with $t, t' \in \mathbb{S}$ and $s, s' \in (\mathbb{Z} \cup \mathbb{S})^*$, $s \neq s'$ implies $t \neq t'$.*

Lemma 4.1
There is a quasi-typed Plump-attributed graph $G_{p,t}$ for every Plump-attributed graph G_p.

Construction 4.1
A Plump-attributed graph G_p is transformed into a quasi-typed Plump-attributed graph $G_{p,t}$ as follows: For every distinct sequence of attribute types s a name t is generated. This name is prepended to the sequence of attributes of all nodes (edges) with matching attribute types.

Proof
By inspection of construction 4.1. □

Remark
The problems described in section 4.3.2 can be solved in a quasi-typed Plump-attributed graph, as different node or edge types can be defined for nodes or edges that have otherwise identical attributes.

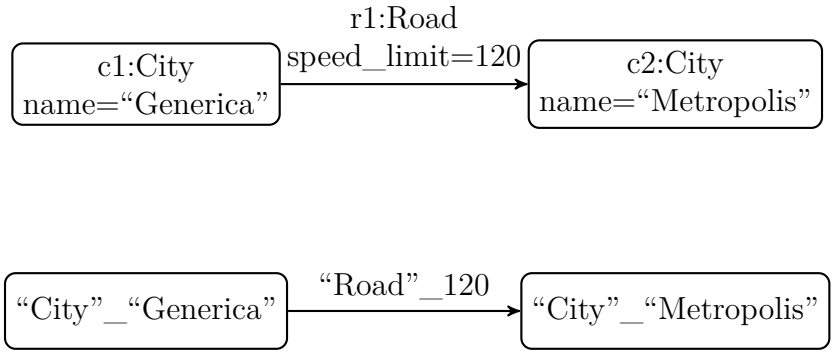


Figure 4.6: Example of the Transformation from Ehrig- to Plump-Attributed Graphs

With these restrictions established, we can now introduce our transformations.

Construction 4.2 (From Ehrig- to Plump-Attributed Graphs)
Given an Ehrig-attributed graph G_e that is restricted to nonempty, single-value attributes, the transformation θ transforms G_e into a quasi-typed Plump-attributed graph G_p as follows:

For every item $x \in G_e$ in the Ehrig-attributed graph:

(a) Create a node (an edge) in the Plump-attributed graph with the name of the type of x – as a string – as the node's (edge's) first attribute.

(b) Add the attributes of x in lexicographical order to the sequence of attributes in the new node (edge).

To transform a rule, convert all its constituent parts individually as above.

An example of this transformation is shown in figure 4.6. Note that the attribute names, such as "speed_limit", are lost during the transformation.

Construction 4.3 (From Plump- to Ehrig-Attributed Graphs)
Given a quasi-typed Plump-attributed graph G_p, the transformation ι transforms G_p into an Ehrig-attributed graph G_e by first creating a type graph (Step 1) followed by creating a graph and a typing morphism (Step 2) as follows:

If more than a single graph is transformed, we need to generate a single type graph for all of these graphs first. Afterwards, we can create the transformed graphs and their typing morphisms over this shared type graph.

Step 1: Creation of the type graph.

To create the type graph, we will look at the nodes and edges both of the graph and of the rules that operate on this graph.

(a) *Create attribute nodes for the types* $\boxed{Nat}$ *and* $\boxed{String}$.

(b) *For each distinct string that is the first attribute of a node (edge):*

(1) *Create a node (edge) with this name in the type graph.*

(2) *Add attribution edges to the attribute types* $\boxed{Nat}$ *and* $\boxed{String}$ *based on the attribute types present in the sequence of attributes for that node (edge). Name these edges after the attribute's position in the sequence, i.e. "1" for the first attribute, "2" for the second.*

Step 2: Creation of the graph and typing morphism

(c) *Create attribute nodes for every possible value of* $\boxed{Nat}$ *and* $\boxed{String}$.

(d) *These attribute nodes are mapped to their respective types by the typing morphism.*

(e) *For every node (edge) in the Plump-attributed graph:*

(1) *Create a node (edge) in the Ehrig-attributed graph.*

(2) *The new node (edge) is mapped to a node (edge) according to the type name that is its first attribute.*

(3) *For all other attributes, create an attribution edge from the newly created node (edge) to the attribute node of the attributes value.*

(4) *The attribution edges from Step (3) above are mapped to attribution edges in the type graph according to their position in the sequence.*

Conversion of rule schemata:

A rule schema can not generally be represented by a single rule over Ehrig-attributed graphs. We therefore convert all instances of a rule schema instead, which can lead to an infinite set of rules. If a finite representation is desired, extensions to Ehrig-attributed graphs, like symbolic attributed graphs [Ore11] could be used.

Rules that change the type of a node or edge:

Rules for Ehrig-attributed graphs do not support changing the type of a node or edge. To be able to convert rules for Plump-attributed graphs that do *change a type we need to generate a (potentially infinite) set of rules that replaces the node or edge with one of a different type in an arbitrary context.*

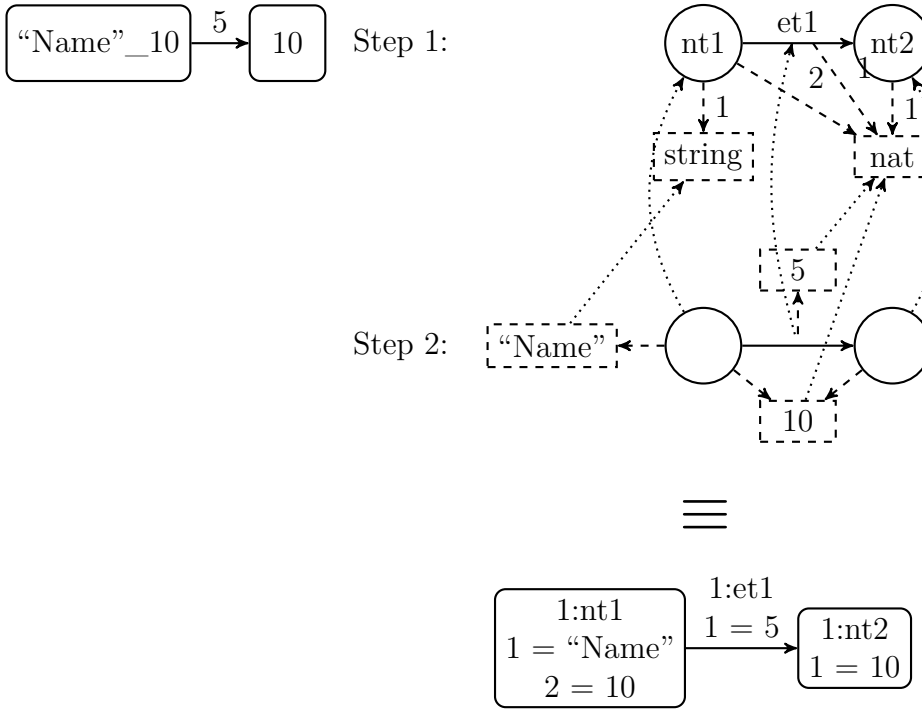


Figure 4.7: Example Transformation from Plump- to Ehrig-Attributed Graph

In the following proofs we use for example (Step $\iota.e.1$) to refer to "(1) Create a node (edge) in the Ehrig-attributed graph." in the definition above.

Figure 4.7 shows an example of this transformation. In the first step the type graph on the upper right is constructed, in step 2 the graph below is added and identified with its types via the typing morphism. Finally, the Ehrig-attributed graph is shown in its simplified notation in the lower right.

Lemma 4.2
The transformations θ and ι result in Plump-attributed and Ehrig-attributed graphs respectively.

Proof
The different steps of θ and ι only create valid elements for the resulting graphs, as can be easily seen from the transformations. □

Ideally we would want $\iota(\theta(G_e)) = G_e$ and $\theta(\iota(G_p)) = G_p$ to hold. That is we would want the application of both transformations to result in the graph we started with. This is not the case for two reasons:

(1) The type graph for $\iota(\theta(G_e))$ is potentially smaller than the type graph for G_e, since unused types will not be recreated by ι.

(2) The names of attributes in $\iota(\theta(G_e))$ and G_e differ.

We do however retain the number and types of attributes in the type graph. To express this relationship we introduce a notion of *weak isomorphism* for Typed Attributed Graphs over different type graphs.

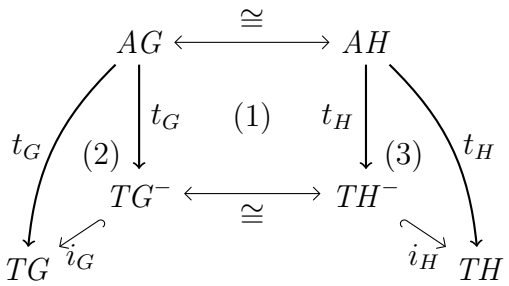
Definition 4.4 (Weak Isomorphism for Typed Attributed Graphs)
Two Typed Attributed Graphs $G = (AG, t_G)$ over ATG and $H = (AH, t_H)$ over ATH are called weakly isomorphic *if:*

(1) The type graphs $ATG = (TG, Z)$ and $ATH = (TH, Z)$ share a common algebra Z,

(2) AG and AH are isomorphic,

(3) TG^- and TH^- are isomorphic, where for TG, TG^- is the subgraph of TG constructed by restricting TG to the range of t_G. The restrictions induce two inclusions $i_G : TG^- \to TG$ and $i_H : TH^- \to TH$,

such that (1), (2) and (3) in the following diagram commute:

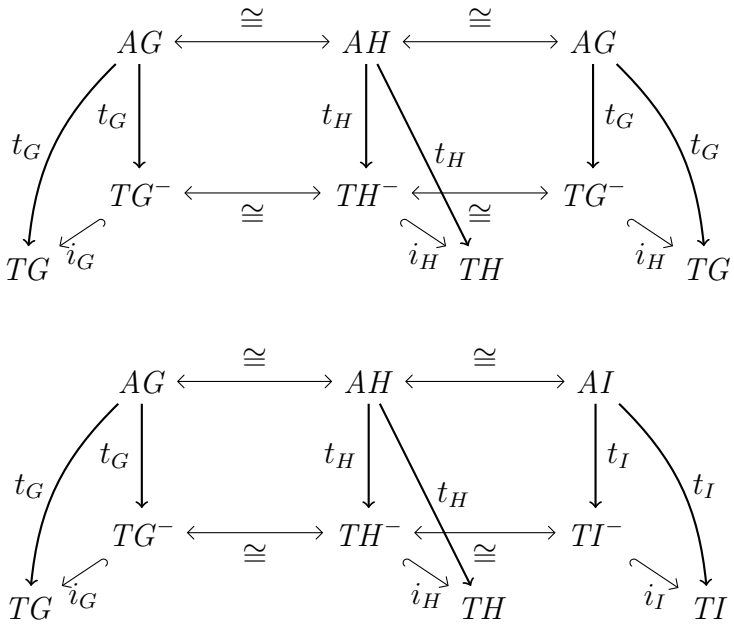


We write $G \cong_\omega H$.

Remark
Every two isomorphic Ehrig-attributed graphs are weakly isomorphic, since there is an isomorphism from the attributed graph to itself and a trivial isomorphism from the attributed type graph to itself.

The reverse is not true, since an isomorphism requires that both attributed graphs share a single type graph.

Remark
$\cong_\omega$ defines an equivalence relation. It is trivially reflexive, for symmetry and transitivity consult the following two diagrams:

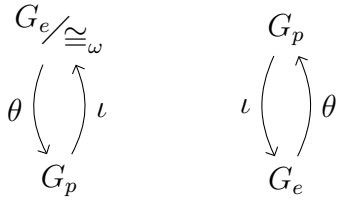


We write $G/\cong_\omega$ for the equivalence class of G induced by $\cong_\omega$.

With weak isomorphisms for Ehrig-attributed graphs we can formally describe the relationship between the original graph and result of the application of both transformations:

Lemma 4.3 (Round Trip)
Using both transformations on a graph results in a weakly isomorphic or equivalent graph, specifically:

1. *For any Ehrig-attributed graph G_e that is restricted to nonempty, single-value attributes, $\iota(\theta(G_e)) \cong_\omega G_e$ holds.*
2. *For any quasi-typed Plump-attributed graph G_p, $\theta(\iota(G_p)) = G_p$ holds.*



Proof (Round Trip for Ehrig-Attributed Graphs)
Given an Ehrig-attributed graph $G_e = (AG, t)$ over $ATG = (TG, Z)$ that is restricted to nonempty, single-value attributes, $\iota(\theta(G_e))$ results in a graph $G'_e = (AG', t')$ over $ATG' = (TG', Z)$ with the standard algebra Z:

(1) Common algebra:
The condition trivially holds, since we restricted our Ehrig-attributed graphs to a fixed algebra and the transformation ι uses that same algebra when constructing a new type graph.

(2) Existence of an isomorphism $i : TG^- \leftrightarrow TG'$:

- *Every node $n_{t,e} \in Ran(t)$ results in a node $n_p \in G_p$ with the type name as its first attribute (Step θ.a) and a sequence of attributes ordered by name (Step θ.b).*
- *The transformation of the node $n_p \in G_p$ results in a new node $n'_{t,e} \in ATG'$ (Step ι.b.1), that has attribution edges to the same types as the original node $n_{t,e} \in ATG$ (Step ι.b.2).*

Mapping each node $n'_{t,e}$ to the original $n_{t,e}$ and doing the same for edges and attribution nodes and edges results in an isomorphism $i : TG^- \leftrightarrow TG'$, where TG^- is the restriction of TG to $Ran(t)$ with the inclusion $i_G : TG^- \to TG$. The inclusion $i_{G'}$ is simply $id_{G'}$.

(3) Existence of an isomorphism $m : AG \leftrightarrow AG'$:
Step θ.c creates a node $n_p \in G_p$ for every node $n_e \in AG$. Step ι.e.1 in turn creates a node $n'_e \in AG'$ for every node $n_p \in G_p$. The attribution edges, which do not survive the transformation θ (Step θ.b), are recreated by ι (Step ι.e.3).

Mapping the nodes $n_e \in AG$ to $n'_e \in AG'$ and doing the same for edges and attribution nodes and edges, results in an isomorphism $m : AG \leftrightarrow AG'$.

(4) Diagrams (1), (2) and (3) commute:

(2) and (3) commute by construction, for (1) consider the following:

m	*maps every node (edge) to its transformed counterpart.*
t'	*maps $m(o)$ to its type.*
i	*maps these types to their original counterparts (as can be seen in (2) above).*

□

Proof (Round Trip for Plump-Attributed Graphs)
Given a quasi-typed Plump-attributed graph G_e, $\theta(\iota(G_p))$ results in a graph G'_p:

1. Nodes and Edges:

- *Every node $n_p \in G_p$ causes a node $n_e \in G_e$ to be created (Step $\iota.e.1$). This node is mapped to a node in ATG (Step $\iota.e.2$), which is named with the first attribute of n_p (Step $\iota.b.1$).*
- *Step $\theta.b.2$ transforms n_e into a node $n'_p \in G'_p$ with the name of $t(n_e)$ as its first attribute.*

2. Attributes:

- *The attributes of a node $n_p \in G_p$ are transformed into a set of attribution edges originating at n_e (Step $\iota.e.3$) which are mapped to an attribution edge in ATG which is named according to the attributes position in the sequence in n_p (Step $\iota.e.4$).*
- *The values these edges point to are added as the attributes of $n'_p \in G'_p$ according to the name of the edge (Step $\theta.b.2$).*

Thus, every element of the original graph G_p is recreated in G'_p and no additional elements are created. □

We would further want that the application of rules in one concept can be matched by the application of a converted rule in the other concept. Such that for every Ehrig-attributed graph that is restricted to nonempty, single-value attributes, there is a quasi-typed Plump-attributed graph such that for all rules applicable in the Ehrig-attributed graph there is a corresponding transformed rule that can be applied to the Plump-attributed graph. The same relationship holds in the opposite direction.

To be able to *track* an item of one graph through the application of a rule we use a *track morphism* as defined below:

Definition 4.5 (Track Morphism [Plu99])
Given a derivation $G \Rightarrow_r H$, the track morphism *$tr(r) : G \to H$ is the partial morphism defined by*

$$tr(r) = \begin{cases} d(c^{-1}(x)) & \text{if } x \in c(D) \\ \bot & \text{otherwise} \end{cases}$$

Here $c : D \to G$ and $d : D \to H$ are the morphisms in the lower row of a derivation.

Using these track morphisms, we can formalise our requirements regarding transformed rules:

Theorem 4.4 (Mutual Simulation)
Using the transformations θ and ι as defined earlier:

1. *From Ehrig- to Plump-attributed (left diagram)*
   For every rule r_e and derivation $G_e \Rightarrow_{r_e} G'_e$ exists a transformed rule $r_p = \theta(r_e)$, such that r_p applied to the graph $G_p = \theta(G_e)$ yields a graph G'_p, such that $\iota(G'_p) \cong_\omega G'_e$.
2. *From Plump- to Ehrig-attributed (right diagram)*
   For every rule r_p and derivation $G_p \Rightarrow_{r_p} G'_p$ exists a transformed rule $r_e = \iota(r_p)$, such that r_e applied to the graph $G_e = \iota(G_p)$ yields a graph G'_e, such that $\theta(G'_e) = G'_p$.

$$\begin{array}{ccc} G_e & \xrightarrow{tr(r_e)} & G'_e \\ {\scriptstyle\theta}\downarrow & \cong_\omega & \uparrow{\scriptstyle\iota} \\ G_p & \xrightarrow[tr(r_p)]{} & G'_p \end{array} \qquad \begin{array}{ccc} G_e & \xrightarrow{tr(r_e)} & G'_e \\ {\scriptstyle\iota}\uparrow & = & \downarrow{\scriptstyle\theta} \\ G_p & \xrightarrow[tr(r_p)]{} & G'_p \end{array}$$

Proof (Mutual Simulation)
1. From Ehrig- to Plump-attributed graphs:

Given a derivation $G_e \Rightarrow_{r_e} G'_e$:

(1) $r_p = \theta(r_e)$ has a match m_p in $G_p = \theta(G_e)$ that does not violate the dangling condition:

- *Every element of the match m_e is transformed into a corresponding element in G_p, these elements form a match m_p for r_p.*
- *m_p does not violate the dangling condition if m_e doesn't since θ creates no additional edges.*

If there is a match m_p the existence of a derivation $G_p \Rightarrow_{r_p} G'_p$ follows.

(2) $\iota(G'_p) \cong_\omega G'_e$:

- *All elements of G_e that are not in m_e are unchanged in G'_e, for all those elements lemma 4.3 holds.*

- *The remaining elements of G'_e are the right-hand side of r_e, with lemma 4.3, $r_e \cong_\omega \iota(r_p)$ holds.*

2. From Plump- to Ehrig-attributed graphs:

Given a derivation $G_p \Rightarrow_{r_p} G'_p$:

(1) $r_e = \iota(r_p)$ has a match m_e in $G_e = \iota(G_p)$ that does not violate the dangling condition:

- *Every element of the match m_p is transformed into a corresponding element in G_e, these elements form a match m_e for r_e.*
- *m_e does not violate the dangling condition since ι does not introduce additional edges and the rule always references* all *attribution edges.*

If there is a match m_e the existence of a derivation $G_e \Rightarrow_{r_e} G'_e$ follows.

(2) $\theta(G'_e) = G'_p$:

- *All elements of G_p that are not in m_p are unchanged in G'_p, for all those elements lemma 4.3 holds.*
- *The remaining elements of G'_p are the right-hand side of r_p, with lemma 4.3 $r_p = \theta(r_e)$ holds.* □

We can therefore *simulate* the application of rules in Ehrig-attributed graph that are restricted to single-value, nonempty attributes using quasi-typed Plump-attributed graphs. Thus either concepts allows us to express similar classes of attributed graphs.

4.5 Analysis

We have compared Ehrig-attributed and Plump-attributed graphs, both with respect to issues that can arise when using these concepts, as well as investigating their compatibility by introducing transformations between subsets of them.

The concept of Ehrig et al. allows named attributes and allows specifying only a part of the attributes that a node or edge has for the purpose of finding a match. The graph must be typed, however, which means the attributes of a certain node or edge are fixed and adding or removing attributes is impossible in a rule. Because the values of attributes are determined by the presence or absence of edges, an attribute in typed attributed graphs behaves like a multiset. It can have no value at all, a single value or multiple values, even multiples of the same value. Intuitively we would expect an attribute to have only one value, unless it is specifically meant to be a multiset. Lastly, typed attributed graphs require changes to the definition of graphs that add a lot of complexity. Most of that complexity is hidden by the simplified notation that is introduced with typed attributed graphs, hiding these additional elements can make their behavior all the more baffling though.

The concept of Plump allows attributes with different types and enables this with only few additions to (partially) labelled graphs in the form of rule schema. The labels of nodes and edges are only decomposed into attributes after finding a match, which makes it impossible to specify only a part of the attributes of a node or edge. GP allows adding or removing attributes, doing so can lead to ambiguity since the attributes have no names, only positions in a sequence. Removing an attribute can therefore lead to its successor being used in a rule schema.

The properties of both concepts are summarized in table 4.1 below.

For both concepts there are cases where one is the better choice. Typed attributed graphs are typed and are better suited to applications where adding or removing attributes would be counterproductive. GP was introduced to express algorithms over graphs in a graphical way. In many of these algorithms the ability to add or remove attributes is helpful and sometimes necessary for holding temporary values.

Some of the undesirable properties described above can be mitigated. Empty attributed or multiple values in typed attributed graphs can be avoided by restricting the number of attribution edges per node or edge both in the graph and in the rules that operate on it. The ambiguity that can arise in GP can be avoided by prepending a string to the attributes, as described in definition 4.3, with somewhat similar results to typing.

	Ehrig et al. [EEPT06]		Plump [Plu09]
+	Named attributes	–	No way to refer to only one attribute out of several
–	Attributes are multisets	–	Possible ambiguity when attributes are removed/added
–	Graph and attributes must be typed	+	Types for attributes
–	Fixed attributes per node due to typing	+	Ability to add/remove attributes
–	Changes definition of graph & morphism	+	Uses (partially) labelled graphs with relabelling
++	$\mathcal{M}$-adhesive	+	$\mathcal{M},\mathcal{N}$-adhesive

Table 4.1: Comparison of the Concepts

We have also shown that Ehrig-attributed graphs and Plump-attributed graphs with these restrictions can be transformed into each other. The restrictions to Ehrig-attributed graphs result in a proper subset while quasi-typing the Plump-attributed graph is always possible. This relationship is illustrated in figure 4.8

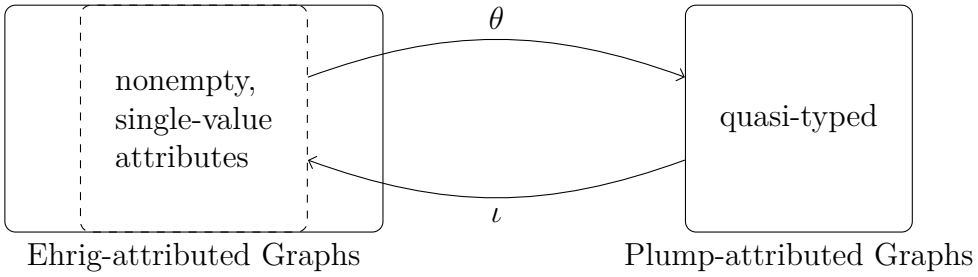


Figure 4.8: Relationship between Ehrig-Attributed and Plump-Attributed Graphs

Beside the aspects investigated earlier in this chapter, we look at the theory developed for both concepts.

There is a well-developed theory for typed attributed graphs in the form of $\mathcal{M}$-adhesive categories [EGH10, EGH$^+$12, EGH$^+$13].

The category of partially labelled graphs with relabelling, that is used as the basis for attribution in GP, is $\mathcal{M},\mathcal{N}$-adhesive [HP12]. $\mathcal{M},\mathcal{N}$-adhesive categories are a restriction of $\mathcal{M}$-adhesive categories, with fewer results published.

5 New Attribution Concept

In this chapter, we introduce a new concept for attribution in graph transformation. We start with the requirements we have for such a concept. We continue with the definition of a suitable category of collections of attributes, attach such attributes to graphs and prove that the resulting category has direct derivations that are unique up to isomorphism.

5.1 Requirements

The following requirements are derived from the comparison in chapter 4. We list the properties we require of an attribution concept followed by a short explanation why these properties are desirable.

- Single-value attributes
  As discussed in chapter 1 attributes with multiple values - or even multiples of the same value - are not the intuitively expected behavior.

- Nonempty attributes
  Similarly, attributes that have no value at all can lead to unexpected behavior.

- Ability to add/remove attributes
  For many algorithms on graphs, typing the graph is unnecessary and might even be overly restrictive. Additionally we would like to simulate the graphs used in GP.

- Ability to restrict addition/removal of attributes
  In other cases, for example UML diagrams, it is necessary to restrict or prohibit the addition or removal of attributes. Additionally we would like to simulate Typed Attributed Graphs.

- Named attributes
  Since we want several attributes per node or edge we need some way to distinguish between these attributes. In graphs as used in GP this is done by

position in a sequence, this can, however, lead to ambiguity, as seen in section 4.3.

- Types for attributes
  In order to enable computation over attribute values, types should be specified. Typed Attributed Graphs allow the specification of different types for attributes. These types are bound to typing for the nodes and edges of a graph however. Since we want typing for the graph to be optional, we have to provide an alternative for specifying attribute types.

- Ability to re-use existing results for graph transformation
  Ideally all of the above requirements should not prevent our concept from using existing results form the literature.

5.2 General Idea

In Typed Attributed Graphs attributes behave like multisets since attribute values are nodes and there can be any number of edges between nodes. In contrast, the labels in graphs used in GP have very little structure to them, they are just a sequence of attributes. These graphs otherwise have many of the properties we want to have for attributes.

We therefore generalize (partially) labelled graphs with relabelling by allowing more complex labels.

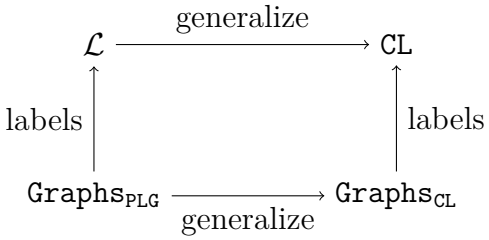


Figure 5.1: Visualization of the General Idea

To achieve this we first focus on the labels and construct an appropriate category, the objects of which are all the attributes of a node or edge. Afterwards we consider graphs labelled with these objects and construct a category $\texttt{Graphs}_{\texttt{CL}}$ of such graphs.

5.3 Attribute Collections

We start by constructing a category of collections of attributes and show that this category has pullbacks, pushouts, pushout-complements and finally direct derivations. These collections of attributes should satisfy as many of the above requirements as possible.

Such an *attribute collection* consists of a set of names, that are mapped to types and values. We require a name to always have a type and further require that values mapped to a name are of this type.

Definition 5.1 (Attribute Collection)
Let Σ be an algebra and T its sorts (called types in the following) and $V = (V_t)_{t \in T}$ a family of values for the types. An attribute collection *is a system $A = (N_A, t_A, v_A)$ over Σ and a set N of possible attribute names, where $N_A \subset N$ is a finite set of attribute names, $t_A : N \to T$ is a total function that assigns types to names and $v_A : N_A \times T \to V_T$ is a partial function that maps a pair of a name and a type to a value.*

Additionally, the following conditions hold:

(1) $\forall (n,t) \in Dom(v_A) : t_A(n) = t$, i.e. typing is correct for names.

(2) $\forall v \in Ran(v_A)$ with $v_A(n,t) = v : v \in V_t$, i.e. typing is correct for values.

An attribute collection is complete *if $\forall n \in N_A \exists t \in T : (n,t) \in Dom(v_A)$.*

These attribute collections already satisfy some of our requirements. Each attribute has a name, a type and can have *at most one* value. Most of the other requirements only become relevant once we have a way of applying rules to attribute collections.

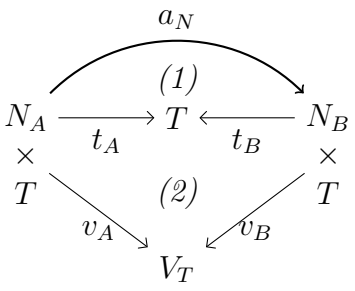
Example 5.1
We create an attribute collection from a node from the city example from section 3.1. $N_A = \{name, population\}$, $t_A = \{name \mapsto String, population \mapsto Nat\}$, $v_A = \{(name, String) \mapsto$ "Generica", $(population, Nat) \mapsto 100000\}$. The example is shown in figure 5.2.

$$name = \text{"Generica"}$$
$$population = 100000$$

Figure 5.2: An Example of an Attribute Collection

To form a category of these attribute collections we need morphisms for them. While the morphisms should preserve the structure of an attribute collection, it is useful to allow morphisms to *not* preserve values. This can later be used to realize typing in a similar way to typed graphs.

Definition 5.2 (AC-Morphism)
A AC-morphism $a : A \to B$ *from an attribute collection* A *to an attribute collection* B *is a function* $a_N : N_A \to N_B$ *such that diagram (1) commutes:*



The AC*-morphism:*

preserves values *if diagram (2) commutes.*

preserves undefinedness *if* $v_A(n,t) = \perp$ *implies* $v_B(a_N(n),t) = \perp$.

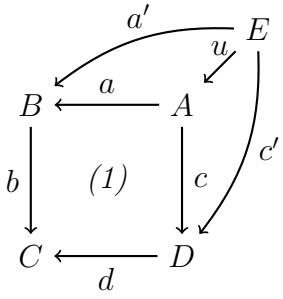
An AC*-morphism* a *is injective (surjective) if* a_N *is injective (surjective) and an isomorphism if it is injective, surjective and preserves undefinedness. An* AC*-morphism* a *an inclusion if* $a_N(n) = n$ *for all* $n \in N_A$.

Fact 5.1
Attribute collections and AC*-morphisms constitute a category* AC*, where composition of morphisms is function composition.*

In the following we show the existence of pullbacks, pushouts and direct derivations that preserve values for the category AC of attribute collections. For the existence of pullbacks, pushouts and direct derivation that *do not* preserve values we can simply refer to the proofs for totally labelled graphs (without relabelling) as presented in [Ehr79]. It is easy to see the parallels: an attribute collection, where names are mapped to types, is very similar to a simple graph, where nodes are mapped to labels. For the attribute values the situation is somewhat similar. In this case the proofs are close to those for partially labelled graphs with relabelling [HP02]. We provide full proofs for the value preserving concepts, since we do not require all of the properties of partially labelled graphs, allowing us to simplify the proofs.

We start with *pullbacks*, which we will need to define *natural pushouts* later.

Lemma 5.2 (Existence of Pullbacks)
Given AC*-morphisms $b : B \to C$ and $d : D \to C$ that preserve values, there exists an attribute collection A and* AC*-morphisms $a : A \to B$, $c : A \to D$ such that the diagram below is a pullback:*



Proof
(1) Construct A:

The names of A are constructed as $\{\langle n, m\rangle \in N_B \times N_D | b(n) = d(m)\}$, the typing function as $t_A(\langle n, m\rangle) = t_B(n) = t_D(n)$ and the value function is defined as

$$v_A(\langle n, m\rangle, t) = \begin{cases} v_B(n, t) & \text{if } v_B(n, t) = v_D(m, t) \neq \bot \\ \bot & \text{otherwise} \end{cases}$$

Note that $t_A(\langle n, m\rangle)$ is well defined, since t_B and t_D are total functions and b, d preserve types.

A satisfies the conditions for attribute collections:

(1) $\forall (n, t) \in Dom(v_A) : t_A(n) = t$*:*
If $(n, t) \in Dom(v_A)$ then there is a $(n, t) \in Dom(v_B)$ for which (1) holds, then by construction of t_A, $t_A(n) = t$.

(2) $\forall v \in Ran(v_A)$ *with* $v_A(n, t) = v$ *and* $v \in V_t$*:*
If $v \in Ran(v_A)$ then there is a $v' \in Ran(v_B)$ for which (2) holds, then by definition of v_A, $v = v'$ and (2) holds for v_A.

(2) Construct AC*-morphisms a,c:*

Let $a : A \to B$ and $c : A \to D$ be the projections from $N_B \times N_D$ to N_B and N_D respectively, that is, $a(\langle n, m\rangle) = n)$ and $c(\langle n, m\rangle) = m$.

For the preservation of types (i.e. a, c are actually morphisms), refer to the proofs for totally labelled graph without relabelling [Ehr79] as outlined above.

To show that a,c preserve values, consider $(\langle n, m\rangle), t) \in Dom(v_A)$.
Then $v_B(a(\langle n, m\rangle, t)) = v_B(n, t) = v_A(\langle n, m\rangle, t)$ *and* $v_B(c(\langle n, m\rangle, t)) = v_D(m, t) = v_B(m, t) = v_A(\langle n, m\rangle, t)$ *by definition of a, c and* v_A *and since* $(\langle n, m\rangle, t) \in Dom(v_A)$.

Hence a and c are AC*-morphisms that preserve values.*

(3) The square (1) commutes:

By construction in (1),(2).

(4) Universal property:

Let $a' : E \to B$ *and* $c' : E \to D$ *be morphisms that preserve values with* $b \circ a' = d \circ c'$.
There is only one choice to define $u : E \to A$ *such that* $a \circ u = a'$ *and* $c \circ u = u'$:
$u(n, t) = (\langle a'(n), c'(n)\rangle, t)$ *for all* $n \in N_E$.

It remains to show that u is an AC*-morphism (i.e. u preserves values). To show that u preserves values, let* $(n, t) \in Dom(v_E)$.

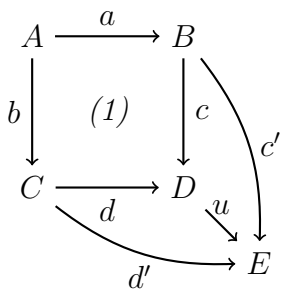
If $v_B(a'(n), t) = v_D(c'(n), t) \neq \perp$, *we have* $v_A(u(n), t) = v_A(\langle a'(n), c'(n)\rangle, t) = v_B(a'(n), t) = v_E(n, t)$ *by definition of u,* v_A *and the fact that* a' *preserves values.*

If $v_B(a'(n), t) = \perp$ *or* $v_D(c'(n), t) = \perp$, *then* $v_A(u(n), t) = v_A(\langle a'(n), c'(n)\rangle, t) = \perp = v_E(n, t)$ *by definition of u,* v_A *and the fact that* a' *and* c' *preserve values.*

Thus u is a AC*-morphism.* □

Since our rules will consist of two *pushouts*, we define them next:

Lemma 5.3 (Existence of Pushouts)
Let $a : A \to B$ *be an inclusion that preserves values and* $b : A \to C$ *be an* AC*-morphism that preserves values, such that B is complete and for all* (n, t), *where* $n \in N_B$ *and* $t_B(n) = t$, $\{v_B(n, t)\} \cup v_C(b(n), t)$ *contains at most one element. Then there exists an attribute collection D and* AC*-morphisms* $c : B \to D$, $d : C \to D$, *such that the following diagram is a pushout:*



Proof
(1) Construct attribute collection D:

$D = (N_D, t_D, v_D)$, *where* $N_D = N_C \cup N_B$ $t_D(n) = t_C(n) = t_B(n)$ *and*

$$v_D(n,t) = \begin{cases} v_C(n,t) & \text{if } n \in (N_C \setminus b(N_A)) \\ v_B(n,t) & \text{otherwise} \end{cases}$$

Note that $t_D(n)$ *is well-defined since* t_C, t_B *are total functions, and* a, b *preserve types.*

A satisfies the conditions for attribute collections:

(1) $\forall (n,t) \in Dom(v_D) : t_D(n) = t$*:*
If $(n,t) \in Dom(v_D)$ *then there is a* $(n,t) \in Dom(v_B)$ *or* $(n,t) \in Dom(v_C)$ *for which (1) holds, then by construction of* t_D, $t_D(n) = t$.

(2) $\forall v \in Ran(v_D)$ *with* $v_D(n,t) = v$ *and* $v \in V_t$*:*
If $v \in Ran(v_D)$ *then there is a* $v' \in Ran(v_B)$ *or a* $v' \in Ran(v_C)$ *for which (2) holds, then by definition of* v_D, $v = v'$ *and (2) holds for* v_D.

(2) Construct morphisms c, d*:*

Let $c(n) = n$ *and* $d(n) = \begin{cases} a(n') & \text{if } n = b(n') \text{ for some } n' \in N_A \\ n & \text{otherwise} \end{cases}$.

For the preservation of types (i.e. c, d *are actually morphisms), refer to the proofs for totally labelled graph without relabelling [Ehr79] as outlined above.*

To show that c *and* d *preserve values:*

(c) Let $(n,t) \in Dom(v_B)$. *Then* $v_D(c(n),t) = v_B(n,t)$ *by definition of* c, v_D *and since whenever* $n \notin (N_C \setminus b(N_A))$, $n \in N_B$ *and therefore* $(n,t) \in Dom(v_B)$ *for* $t = t_B(n)$, *so* c *is value-preserving.*

(d) Let $(n,t) \in Dom(v_C)$.

- *If* $n \in N_C$ *and* $t_C(n) = t$ *and* $n \notin b(N_A)$, *then* $v_D(d(n),t) = v_C(n,t)$ *by definition of* d *and* v_D.
- *If* $n \in b(N_A)$, *then* $n' = b(n)$ *for some* $n' \in N_A$. *Then* $v_D(d(n),t) = v_B(a(n'),t) = v_C(b(n'),t) = v_C(n,t)$ *by definition of* v_D, *the fact that for all* (n,t), *where* $n \in N_B$ *and* $t_B(n) = t$, $\{v_B(n,t)\} \cup v_C(b(n),t)$ *contains at most one element, the fact that* b *preserves values and the fact that* $(n,t) \in Dom(v_C)$.

Thus c, d are AC*-morphisms that preserve values.*

(3) The square (1) commutes:

By construction of (1),(2).

(4) Universal property:

Let $c' : B \to E$ and $d' : C \to E$ be morphisms that preserve values with $d' \circ b = c' \circ a$. There is only one choice to define $u : D \to E$ such that $u \circ d = d'$ and $u \circ c = c'$:

$$u(n) = \begin{cases} c'(n) & \text{if } n \in N_B \\ d'(n) & \text{otherwise} \end{cases}$$

It remains to show that u is a morphism (i.e. u preserves values). To show that u preserves values, let $(n, t) \in Dom(v_D)$. We have two cases:

- *(n, t), where $n \in N_B$ and $t_B(n) = t$ and $v_B(n, t) \neq \bot$. Then $v_E(u(n), t) = v_E(c'(n), t) = v_B(n, t) = v_D(n, t)$ by definition of u, the fact that c' preserves values and the fact that since whenever $n \notin (N_C \setminus b(N_A)$, $n \in N_B$ and therefore $(n, t) \in Dom(v_B)$ for $t = t_B(n)$, and the definition of v_D.*
- *(n, t), where $n \in N_C$ and $t_C(n) = t$ and $n \in b(N_A)$. Then $v_E(u(n), t) = v_E(d'(n), t) = v_C(n, t) = v_D(n, t)$ by definition of u, the fact that d' preserves values and the definition of v_D.*

Thus u is a morphism. □

A pushout that is simultaneously pullback is called *natural*. We will use the naturalness of pushouts later to prove that the results of our derivations are unique up to isomorphism.

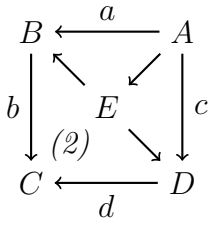
Lemma 5.4 (Characterisation of Natural Pushouts)
Given two morphisms $a : A \to B$ and $c : A \to D$ such that a is injective, the pushout (1) below is natural (i.e. it is simultaneously a pullback) if and only if

$$\begin{array}{c} \text{for all } (n, t), \text{ where } n \in N_A \text{ and } t_A(n) = t : \\ v_A(n, t) = \bot \text{ implies } v_B(n, t) = \bot \text{ or } v_D(n, t) = \bot \end{array} \qquad (*)$$

$$\begin{array}{ccc} B & \xleftarrow{a} & A \\ {\scriptstyle b}\downarrow & (1) & \downarrow{\scriptstyle c} \\ C & \xleftarrow[d]{} & D \end{array}$$

Proof
Let the diagram (1) be a natural pushout with morphisms $b : B \to C$ and $d : D \to C$. Since it is also a pullback, $\{\langle n, m\rangle \in N_B \times N_D | b(n) = d(m)\}$, is an explicit construction of N_A up to isomorphism, where $v_A(\langle n, m\rangle, t) = v_B(n, t)$ if and only if $v_B(n, t) = v_D(m, t) \neq \bot$. It follows that $v_A(\langle n, m\rangle, t) = \bot$ if and only if $v_B(n, t) = \bot$ or $v_D(m, u) = \bot$. Hence condition $(*)$ is satisfied.

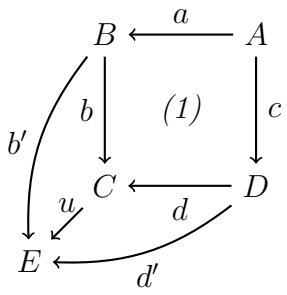
Conversely let (1) be a pushout satisfying condition $(*)$. In the diagram below, let (2) be a pullback of $b : B \to C$ and $d : D \to C$, let $a' : E \to B$ and $c' : E \to D$ be morphisms, by the universal property of (2) there exists a unique morphism $u : A \to E$ such that $a' \circ u = a$ and $c' \circ u = c$.



We show that u is an isomorphism. Injectivity follows from injectivity of $a' \circ u = a$. To see that u is surjective, consider some (n', t), where $n \in N_E$ and $t_E(n') = t$. Then $b(a'(n'), t) = d(c'(n'), t)$ by commutativity of (2). Hence, by injectivity of a and the pushout characterisation of [EK79] (Theorem 1.2) applied to diagram (1) there is some $n \in N_A$ such that $a(n, t) = a'(n', t)$ and $c(n, t) = c'(n', t)$. Applying the pullback characterisation of [EK79] (Theorem 1.7) to (2) gives $u(n, t) = (n', t)$. Finally u preserves undefinedness by condition $(*)$. Thus u is an isomorphism, implying that diagram (1) is a pullback and hence a natural pushout. □

We want to prove the existence of a direct derivation given the existence of a match. The match however, is not sufficient for building the first pushout. Instead we can build the pushout-complement, for the existence of which we give a proof below.

Lemma 5.5 (Existence & Uniqueness of Natural Pushout-Complements)
Let $b : B \to C$ be an `AC`*-morphism that preserves values and $a : A \to B$ an inclusion that preserves values and B be complete. Then there exists an attribute collection D and morphisms $c : A \to D$, $d : D \to C$ such that (1) in the diagram below is a natural pushout:*



Moreover, in this case D is unique up to isomorphism.

Proof
(1) Construct attribute collection D:

$D = (N_D, t_D, v_D)$, *where* $N_D = N_C \setminus (N_B \setminus N_A)$,

$$t_D(n) = \begin{cases} t_A(n) & \text{if } n \in N_A \\ t_C(n) & \text{otherwise} \end{cases}$$

and

$$v_D(n,t) = \begin{cases} v_A(n',t) & \text{if } n = b(n') \text{ for some } n' \in N_A \\ v_C(n,t) & \text{otherwise} \end{cases}$$

D satisfies the conditions for attribute collections:

(1) $\forall (n,t) \in Dom(v_D) : t_D(n) = t$:
If $(n,t) \in Dom(v_D$ *then* $n \in N_A$ *or* $n \in N_C$ *for which (1) holds, then by construction of* t_D, $t_D(n) = t$.

(2) $\forall v \in Ran(v_D)$ *with* $v_D(n,t) = v$ *and* $v \in V_t$:
If $v \in Ran(v_D)$ *then there is a* $v' \in Ran(v_A)$ *or* $v' \in Ran(v_C)$ *for which (2) holds, then by definition of* v_D, $v' = v$ *and (2) holds for* v_D.

(2) Construct morphisms c, d:

Let $d(n) = n$ *and* $c(n) = \begin{cases} b(n) & \text{if } b(n) \in N_D \\ \perp & \text{otherwise} \end{cases}$.

For the preservation of types (i.e. c, d *are actually morphisms), refer to the proofs for totally labelled graph without relabelling [Ehr79] as outlined above.*

To show that c *and* d *preserve values:*

(c) For $(n,t) \in Dom(v_A)$, *we have* $v_D(c(n),t) = v_A(n,t)$, *by the definition of* c *and the definition of* v_D. *Thus* c *preserves values.*

(d) For $(n,t) \in Dom(v_D)$ there are two cases:

- *(n,t), where $n \in N_C$ and $n \notin b(N_B)$ and $t_C(n) = t$. Then $v_C(d(n),t) = v_C(n,t) = v_D(n,t)$ by definition of d and v_D.*
- *(n,t), where $n \in b(N_A)$ and $t_C(n) = t$. Let $n' \in N_A$ with $b(n') = n$. Then $v_C(d(n),t) = v_C(n,t) = v_C(b(n'),t) = v_B(n',t) = v_A(n',t) = v_D(n,t)$ by definition of d and the fact that b, a and c preserve values.*

Hence, c, d are `AC`*-morphisms that preserve values.*

(3) The square (1) commutes:

By construction in (1),(2).

(4) Universal property:

Let $b' : B \to E$ and $d' : D \to E$ be morphisms that preserve values with $b' \circ a = d' \circ c$. There is only one choice to define $u : C \to E$ such that $u \circ b = b'$ and $u \circ d = d'$:

$$u(n) = \begin{cases} d'(n) & \text{if } n \in N_D \\ b'(n) & \text{otherwise} \end{cases}$$

It remains to show that u is a morphism (i.e. u preserves values). To show that u preserves values, let $(n,t) \in Dom(v_C)$. We have two cases:

- *(n,t), where $n \in b(N_B)$ and $t_C(n) = t$ and $n' \in N_B$ with $b(n') = n$. Then $v_E(u(n),t) = v_E(u(b(n')),t) = v_E(b'(n'),t) = v_B(n',t) = v_C(b(n'),t) = v_C(n,t)$ by definition of u and the fact that b and b' are value-preserving.*
- *(n,t), where $n \in N_D$ and $n \in c(N_A)$ and $t_D(n) = t$. Then $v_E(u(n),t) = v_E(d'(n),t) = v_D(n,t) = v_C(n,t)$ by definition of u, the fact that d' preserves values and v_D.*

Thus u is a morphism that preserves values and the square (1) is a pushout.

Moreover, by definition of v_D, property $(*)$ of lemma 5.4 holds: for all (n,t), where $n \in N_A$ and $t_A(n) = t$, $v_A(n,t) = \bot$ implies $v_B(n,t) = \bot$ or $v_D(c(n),t) = \bot$. Thus, by lemma 5.4, diagram (1) is a natural pushout.

It remains to show that D is unique up to isomorphism, that is: its values are uniquely determined. Consider any $(n,t) \in Dom(v_D)$. We have two cases:

- $(n,t) \in Dom(v_C)$ *and* $n \notin b(First(Dom(v_B)))$ [1]*. It can be shown that for every pushout of form (1), for each* $(n,t) \in Dom(v_C)$ *there is* $(n',t) \in Dom(v_D)$ *with* $b(n') = n$ *and* $v_B(n',t) = v_C(n,t)$ *or there is* $(n',t) \in Dom(v_D)$ *with* $d(n') = n$ *and* $v_D(n',t) = v_C(n,t)$*. Since in the present case* $(n,t) \notin Dom(v_B)$ *and* d *is an inclusion,* $v_D(n,t)$ *must be equal to* $v_C(n,t)$*.*
- (n,t)*, where* $n \in b(N_A)$ *and* $t_C(n) = t$*. Let* n' *be the unique element in* N_A *with* $b(n') = n$*: If* $v_A(n',t) \neq \perp$*, then* $v_D(n,t) = v_D(c(n'),t) = v_A(n',t)$ *because* c *preserves values. If* $v_A(n',t) = \perp$*, then by the characterisation of natural pushouts in lemma 5.4,* $v_D(n,t) = v_D(c(n'),t) = \perp$*.* □

Rules are defined analogously to rules for graphs. We require the two morphism of which the rules consist to be inclusions, since we want to change attribute values based on the names of attributes. In practice we will require the same of a match, but this requirement is not necessary for the proofs. We additionally require the left-hand side and the right-hand side of such a rule to be *complete*, i.e we require all of their names to have a value. In this way we can satisfy our requirement for nonempty attributes.

Definition 5.3 (AC-Rule)
An AC-rule $r = \langle L \leftarrow K \rightarrow R \rangle$ *consists of two inclusions* $K \rightarrow L$ *and* $K \rightarrow R$ *that preserve values, where* L, R *are complete.*

As can be seen in the following example, we also fulfill our requirement for the addition of attributes, since the morphisms are not required to be surjective. Removing attribute values is also possible, as can easily be seen from the example by swapping the left-hand side and the right-hand side.

Example 5.2
An example of an AC*-rule is shown in figure 5.3. We add a new attribute growthLY to the attribute collection from example 5.1 and change the value of population.*

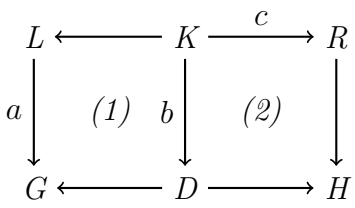
$$\begin{array}{c} name = \text{"}Generica\text{"} \\ population = 100000 \end{array} \longleftarrow \begin{array}{c} name = \text{"}Generica\text{"} \\ population = \perp \end{array} \longrightarrow \begin{array}{c} name = \text{"}Generica\text{"} \\ population = 120000 \\ growthLY = 20000 \end{array}$$

Figure 5.3: Example of an AC-Rule

[1] Where *First* selects the first element of $Dom(v_B)$

Given rules as defined above we prove the existence of derivations that preserve values and are unique up to isomorphism.

Theorem 5.6 (Exist. & Uniq. of Value-Preserving AC-Derivations)
Given an AC*-rule* $r = \langle L \leftarrow K \rightarrow R \rangle$ *and an* AC*-morphism* $a : L \rightarrow G$ *that preserves values there exists a value-preserving* AC*-derivation* $G \Rightarrow_r H$ *such as in the diagram below.*



Moreover D *and* H *are unique up to isomorphism.*

Example 5.3
An example of an AC*-derivation is shown in figure 5.4. The rule on the upper row is very similar to the rule from example 5.2. We could, in fact, use that rule instead without any changes to the match or the result of the derivation.*

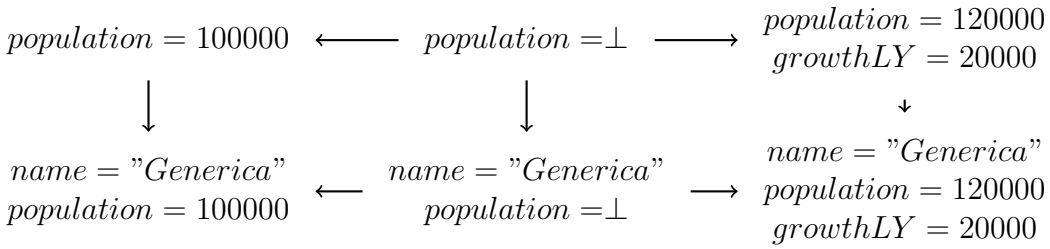


Figure 5.4: Example of an AC-Derivation

Proof
(1) Diagram (1) is a natural pushout:

By lemma 5.5 there exist an attribute collection D *that is unique up to isomorphism and* AC*-morphisms* $D \rightarrow G$ *and* $b : K \rightarrow D$ *that preserve values.*

(2) For all (n, t)*, where* $n \in N_R$ *and* $t_R(n) = t$*,* $\{v_R(n, t)\} \cup v_D(b(c^{-1}(n)), t)$ *contains at most one element:*

$v_R(n, t) \neq \perp$*, since* R *is complete. Then by the definition of a rule ,* $v_L(n', t) \neq \perp$ *with* $n' \in c^{-1}(n)$*. Consider* $n' \in c^{-1}(n)$*:*

- $v_K(n', t) \neq \perp$. *Then* $v_D(b(n'), t) = v_K(n', t) = v_R(c(n'), t)$ *because* b *and* c *preserve values.*
- $v_K(n', t) = \perp$. *Then by the characterisation of natural pushouts in lemma 5.4,* $v_D(b(n'), t) = \perp$.

Hence elements in $c^{-1}(n)$ *either have* $v_R(n, t)$ *as a value or have no value. Thus* $\{v_R(n, t)\} \cup v_D(b(c^{-1}(n)), t)$ *contains at most one value.*

Hence (2) is a pushout by lemma 5.3.

(3) We show that (2) is a natural pushout:

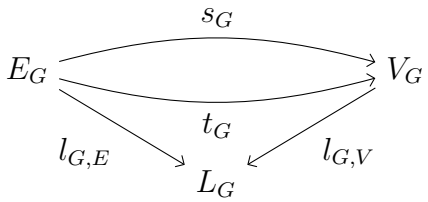
Consider any pair (n, t), *where* $n \in N_K$ *and* $t_K(n) = t$ *with* $v_K(n, t) = \perp$ *and* $v_R(c(n), t) \neq \perp$. *Then* $v_L(n, t) \neq \perp$ *by the definition of a rule. Hence* $v_D(b(n), t) = \perp$ *by the naturalness of pushout (1) and lemma 5.4. By lemma 5.4, diagram (2) is a natural pushout. Hence attribute collection* H *is unique up to isomorphism.* □

This category `AC` of attribute collections satisfies many of our requirements specified in section 5.1, but does not support attributing graphs.

5.4 Graphs with Complex Labels

We need to attach the attribute collections to elements of a graph to enable attribution for graphs. To this end we introduce a more generic category of graphs labelled with the objects of a category CL.

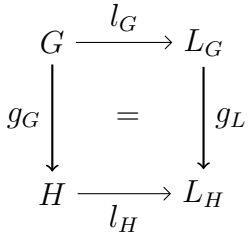
Definition 5.4 (Graphs with Complex Labels)
A graph with complex labels*, or* CL*-graph, over a category* CL*, is a system* $CLG = (G, L, l)$*, where* G *is a graph,* L_G *is a multiset of objects of the category* CL*, and* $l = (l_{G,V}, l_{G,E})$*, with* $l_{G,V} : V \to L$ *and* $l_{G,E} : E \to L$ *are labelling functions that attach elements of* L *to nodes and edges respectively.*



Remark
The labels L_G *are a multiset in a graph with complex labels to prevent the application of a rule from changing the attributes of more than one node or edge unless that is done explicitly.*

The morphisms for this category are graph morphisms together with morphisms for the individual labels.

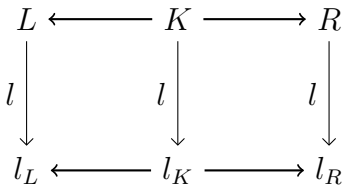
Definition 5.5 (CL-Morphism)
A CL-morphism $g : G \to H$ *from a* CL*-graph* G *to a* CL*-graph* H *consist of a graph morphism* g_G *together with a function* $g_L : L_G \to L_H$ *that consists of a set of morphisms in* CL*.*



Lemma 5.7
CL*-graphs and* CL*-morphisms form categories* CLG(CL)*, where composition of morphisms is componentwise composition of the morphisms of the underlying categories of graphs and* CL*.*

In the same vein rules are composed of graph-rules and rules in the category CL. Except for elements that are not in K, these elements are deleted or added along with the graph elements to which they are attached.

Definition 5.6 (CL-Rules)
A CL-rule $r = \langle L \leftarrow K \rightarrow R \rangle$ *consists of two* CL*-morphisms* $K \rightarrow L$ *and* $K \rightarrow R$*. The components of* a_L*,* b_L *for an element of* L_K *form a rule over the category* CL*.*



Remark
We assume that it is possible to construct the category of partially labelled graphs as a special case of graphs with complex labels by constructing a suitable category L *for the labels:*

- *Objects of the category are elements of the label alphabet* L *and one element to represent unlabelled nodes* ε*.*
- *Morphisms exists between equivalent elements of* L *or in the form* $\varepsilon \rightarrow l$ *for some* $l \in L$*.*

A rule over this category L *of labels is a pair of morphisms* $K \rightarrow L$ *and* $K \rightarrow R$ *as usual. For these rules there are direct derivations with a uniquely determined result. Proofs for this would be analogous to the proofs presented in the previous section. The resulting category* CLG(L) *would behave like the category* $\mathtt{Graphs}_{\mathtt{PLG}}$ *of partially labelled graphs. The proof for the existence of derivations in* CLG(L) *would be similar to those presented later in this chapter.*

5.5 Graphs with Attribute Collections

We use CL-graphs and the attribute collections defined previously to construct graphs with items labelled with collections of attributes.

Definition 5.7 (Graphs with Attribute Collections)
Graphs with attribute collections, *short* AC*-graphs, are the category* CLG(AC) *of* CL*-graphs, where the category of labels* CL *is the category* AC *of attribute collections.*

The definition of rules follows from the definition given in section 5.4.

Example 5.4
The city example from section 3.1 as a AC*-graph is shown in figure 5.5.*

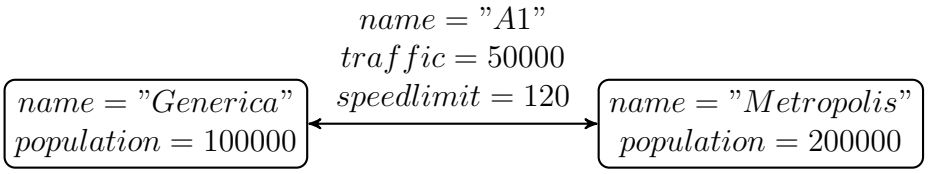


Figure 5.5: Example of an AC-Graph

We are able to show the existence and uniqueness of derivations in the category.

Theorem 5.8 (Existence & Uniqueness of AC-Graph Derivations)
Given a rule $r = \langle L \leftarrow K \rightarrow R \rangle$ *and a morphism* $g : L \rightarrow G$*, over* CLG(AC) *there exists a direct derivation as in the following diagram if and only if the graph component of* g *satisfies the dangling condition.*

$$\begin{array}{ccccc} L & \longleftarrow & K & \longrightarrow & R \\ {\scriptstyle g}\downarrow & & \downarrow & & \downarrow \\ G & \longleftarrow & D & \longrightarrow & H \end{array}$$

Moreover D *and* H *are unique up to isomorphism.*

Proof
(G) For the graph component:

If the graph component g_G *of* g *satisfies the dangling condition there exists a unique direct derivation in the underlying category of graphs.*

(A) For the attribute collections:

For each of the attribute collections in L that is also in K there exists a unique direct derivation, since the components of g_L in the match g are matches for rules in `AC` *and there is such a rule for each element in L_K. Attribute collections that are in L but not in K are deleted from L_G when creating L_D, conversely attribute collections that are not in K but are in R are added to L_H.*

(l) This leaves the labelling functions l:

For attribute collections that are in both L_L and L_K no changes to the labelling function are necessary, since the target of l_K is transformed using the aforementioned rule. The labelling function is extended to any new graph elements in R. □

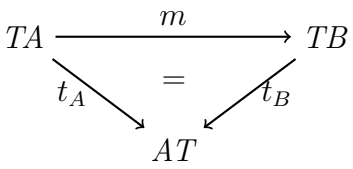
Consequently, we have graph transformation that supports attribution. One of our requirements has not been fulfilled however, we have no way to restrict the addition or removal of attributes yet.

5.6 Typed Graphs with Attribute Collections

We introduce typing for the attribute collections (and consequently the AC-graphs) to satisfy this requirement.

Definition 5.8 (Typed Attribute Collection)
A typed attribute collection *is a pair $TA = (A, t_A)$ over AT consisting of an attribute collection A and a surjective* typing morphism $t_A : TA \to AT$.

Given typed attribute collections $TA = (A, t_A)$ and $TB = (B, t_B$ over AT, a typed AC-morphism $m : TA \to TB$ *is an* AC*-morphism $m : A \to B$ such that $t_A \circ m = t_B$.*



Rules for typed attribute collections are defined analogously to rules for attribute collections.

We show an example of an AC-morphism that is not a typed AC-morphism in figure 5.6 below. If we assume that the attribute collection on the left is the interface of a rule and the attribute collection on the right the right-hand side, the rule would add an attribute. The AC-morphism m is not typed though, since t_H is not a morphism.

If we instead assume that the two attribute collections are interface and left-hand side respectively and the attribute collection below would include an attribute $population : Nat$, this would be a rule for removing an attribute. In that case however, t_G would not be surjective, hence the AC-morphism m would not be a valid typed AC-morphism.

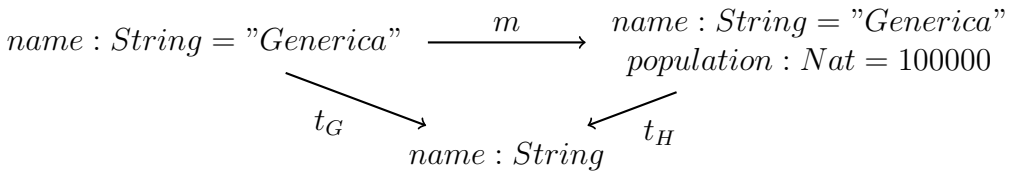


Figure 5.6: AC-Morphism that is *not* a Typed AC-Morphism

Fact 5.9
Typed attribute collections and their morphisms form a category TAC*, where composition of morphisms is componentwise function composition.*

Lemma 5.10 (Existence & Uniqueness of TAC-Derivations)
Given a rule $r = \langle L \leftarrow K \rightarrow R \rangle$ and a typed AC-morphism g that preserves values there exists a value-preserving TAC-derivation $G \Rightarrow_r H$. Moreover D and H are unique up to isomorphism.

Proof
As can be seen from the examples above, rules over TAC are the subset of rules over AC that do not add or remove attributes. The proofs in section 5.3 hold for this subset. □

Remark
Some simplifications would be possible here, since each attribute is now typed twice. We could instead construct attribute collections that only have values and assign the sorts of the algebra as values in the type collection.

This construction of typed attribute collections can be lifted to graphs in the same way as the untyped attribute collections.

Definition 5.9 (Typed Graphs with Complex Labels)
A typed graph with complex labels*, or typed CL-graph, is a pair $G^T = (G, t_G)$ over TG, consisting of a graph with complex labels G and a CL-morphism $t_G : G \rightarrow TG$.*

Given typed graphs with complex labels $G^T = (G, t_G$ and $H^T = (H, t_H$ over TG, a typed CL-morphism *$m : G^T \rightarrow H^T$ is a CL-morphism $m : G \rightarrow H$ such that $t_G \circ m = t_H$.*

Definition 5.10 (Typed Graph with Attribute Collections)
Typed graphs with attribute collections*, short TAC-graphs, are the category of typed CL-graphs, where the category of labels CL is the category AC of attribute collections.*

The definition of rules is analogous to the definition given in section 5.4.

Theorem 5.11 (Existence of TAC-Graph Derivations)
Given a rule $r = \langle L \leftarrow K \rightarrow R \rangle$ and a morphism $g : L \rightarrow G$, over TAC-graphs there exists a direct derivation if and only if the graph component of g satisfies the dangling condition. Moreover D and H are unique up to isomorphism.

Proof
Refer to the proof for untyped graphs with attribute collections in section 5.5 and substitute typed graphs for graphs. □

We thus have an attribution concept for graph transformation that allows typing but does not require it. It satisfies all but one of the requirements listed in the beginning of this chapter: The ability to reuse existing theory. In the next chapter we will consider the ability to reuse existing theory for this new concept.

6 Analysis of the New Concept

In this chapter we analyse the concept introduced in the previous chapter. We start by investigating whether or not graphs with attribute collections belong to the classes of either $\mathcal{M}$-adhesive [EGH10] or $\mathcal{M},\mathcal{N}$-adhesive [HP12] categories. We conclude with a look at related work and compare graphs with attribute collections to the concepts introduced in chapter 3.

6.1 $\texttt{Graphs}_{\texttt{AC}}$ is not $\mathcal{M}$-Adhesive

For a category to be $\mathcal{M}$-adhesive, it must have pushouts along a class $\mathcal{M}$ of morphisms.

Theorem 6.1 ($\texttt{Graphs}_{\texttt{AC}}$ is not $\mathcal{M}$-Adhesive)
The category $\texttt{Graphs}_{\texttt{AC}}$ together with AC-graph morphisms is not $\mathcal{M}$-adhesive, where $\mathcal{M}$ is the class of inclusions.

Proof
It is sufficient to give a counterexample for the category AC of attribute collections alone, we give such a counterexample in figure 6.1 below. □

$$\begin{array}{ccc} population = \perp & \xrightarrow{\ a\ } & population = 120000 \\ {\scriptstyle b}\downarrow & & \downarrow{\scriptstyle c} \\ population = 100000 & \xrightarrow[\ d\ ]{} & population = ? \end{array}$$

Figure 6.1: AC is not $\mathcal{M}$-Adhesive

The proofs in chapter 5 do not assume a single class of morphisms, specifically the vertical morphisms must preserve undefinedness, as it is otherwise not possible to obtain a pushout. If we do not choose morphisms that preserve undefinedness as

the class $\mathcal{M}$, we would have to find a value for *population* such that both c and d above preserve values. On the other hand, the horizontal morphisms can not also preserve undefinedness, since changing attribute values would then be impossible.

6.2 Is $\texttt{Graphs}_{\texttt{AC}}$ $\mathcal{M},\mathcal{N}$-Adhesive?

Conversely it is very likely that both the category AC of attribute collections and $\texttt{Graphs}_{\texttt{AC}}$ are $\mathcal{M},\mathcal{N}$-adhesive [HP12].

Conjecture 6.2 ($\texttt{Graphs}_{\texttt{AC}}$, AC are $\mathcal{M},\mathcal{N}$-Adhesive)
The category AC *of attribute collections and the category* $\texttt{Graphs}_{\texttt{AC}}$ *of graphs labelled with these attribute collections are both $\mathcal{M},\mathcal{N}$-adhesive.*

For AC*, the class $\mathcal{M}$ would consist of all inclusions and $\mathcal{N}$ would consist of all inclusions that preserve undefinedness. For* $\texttt{Graphs}_{\texttt{AC}}$*, the same requirements would apply to the attribute components of morphisms.*

We believe the similarity between attribute collections and partially labelled graphs as noted in chapter 5 makes it very likely that AC, and consequently also $\texttt{Graphs}_{\texttt{AC}}$, is $\mathcal{M},\mathcal{N}$-adhesive.

For $\mathcal{M},\mathcal{N}$-adhesiveness, we would need to show the following:

- Closure properties: $\mathcal{M}$ and $\mathcal{N}$ are closed under composition and decomposition. $\mathcal{N}$ is closed under $\mathcal{M}$-decomposition.
- The category has pushouts along $\mathcal{M},\mathcal{N}$-morphisms. (This is already proven in chapter 5.)
- The category has all pullbacks. (Proof in chapter 5.)
- $\mathcal{M}$ and $\mathcal{N}$ are stable under $\mathcal{M},\mathcal{N}$-pushouts and $\mathcal{M}$-pullbacks.
- Pushouts along $\mathcal{M},\mathcal{N}$-morphisms are $\mathcal{M},\mathcal{N}$-van Kampen squares.

Given the similarities between the proofs in chapter 5 and those for partially labelled graphs, the above mentioned proofs would likely be similar to the proofs in [HP12], where partially labelled graphs are proven to be $\mathcal{M},\mathcal{N}$-adhesive.

6.3 Related Work

In this section we compare AC-graphs with the concepts introduced in chapter 3.

Typed Attributed Graphs Like Typed Attributed Graphs [EEPT06], AC-graphs allow the attribution of both nodes and edges. Attributes are referred to by name and have types and values determined by an algebraic specification. The attributes of a Typed Attributed Graph are implicitly multisets, while the attributes of AC-graphs can only have a single value. If sets or multisets are desired as an attribute type, they have to be explicitly specified in the underlying algebra. While Typed Attributed Graphs must have typed nodes and edges and the attributes of a node or edge are therefore fixed, AC-graphs allows the use of either typed or untyped graphs.

Graphs in GP Both the graphs used in GP [Plu09] and AC-graphs are closely related to partially labelled graphs with relabelling. Both concepts specify attribute types and values through an algebraic specification and both concepts allow the attribution of an untyped graph while still retaining types for attribute values. While a rule in GP must always reference *all* attributes of a node or edge, AC-graphs allow matches even if only a subset of the attributes is specified in the left-hand side of a rule. In both concepts attributes can be added to or removed from a node or edge.

The relationships of AC-graphs, Typed Attributed Graphs and the graphs used in GP is illustrated in figure 6.2 below. The dashed edges represent a likely existence of transformations between our new concept and the other two.

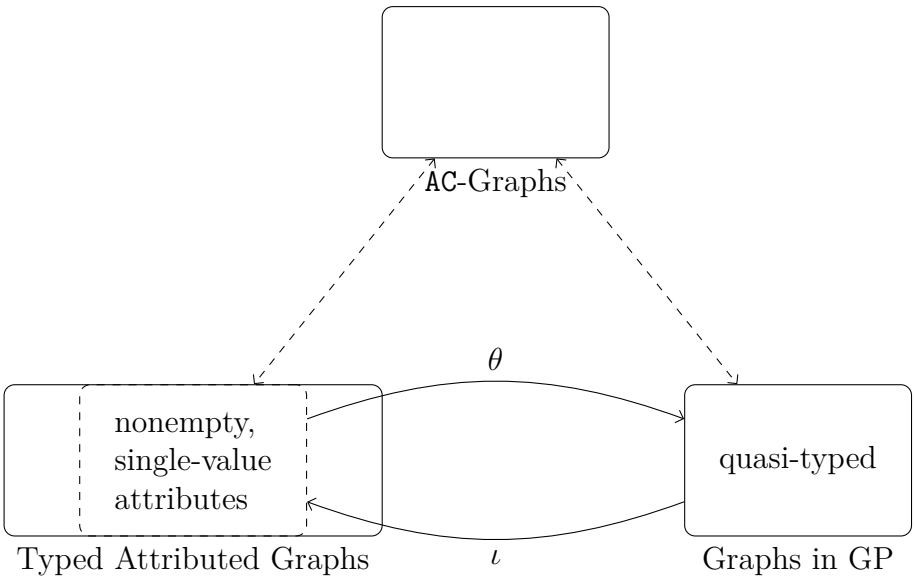


Figure 6.2: AC-Graphs, Typed Attributed Graphs, and Graphs as Used in GP

For the meaning of the transformations ι and θ, as well as nonempty, single-value attributes and quasi-typing refer to section 4.4.

Graphs as Special Algebras The attribution concept in [LKW93] is quite distant from the concept we presented in chapter 5. Due to this tenuous connection, we have not evaluated similarities between the concepts further.

$\mathcal{W}$-adhesive Transformation Systems There are some similarities between the $\mathcal{W}$-adhesive categories defined in [Gol12] and the complex labelled graphs in chapter 5. In both cases types of attributes and their values are determined by the sorts of an algebra and its base sets respectively. The approaches differ in the way attributes are attached to the items of a graph. In $\mathcal{W}$-adhesive categories attributes are mapped to items of a graph individually, according to an attribution type. In complex labelled graphs the attributes of an item form a single object, which is then attached to the item. Due to being determined by the attribution type, attributes can not be added to or removed from an item in $\mathcal{W}$-adhesive categories. On the other hand, $\mathcal{W}$-adhesive categories are defined over an arbitrary $\mathcal{M}$-adhesive category and can therefore be used to attribute other structures besides graphs.

7 Conclusion and Future Work

Attribution is important for many modeling problems that are otherwise well suited for graph models. In this thesis we have briefly explored the existing concepts for attribution in graph transformation. We compared two of these concepts, Typed Attributed Graphs by Ehrig et al. and the graphs used in GP by Plump. We were able to show that these concepts, despite their significantly different approaches to the problem of attribution, can nevertheless handle similar classes of attributed graphs. To this end we introduced transformations between the two approaches.

We further introduced a new approach to attribution based on a generalization of labelling that allows for more complex labels. We constructed a category for these labels and a category of graphs with attribute collections labelled with them. By building a separate category for the attributes we reach a clear separation between graph and attributes. The approach is flexible enough to be used in both typed and untyped graphs and allows for the addition and removal of attributes in the untyped case.

It seems to be obvious that both Typed Attributed Graphs and the graphs used in GP can be expressed in graphs with attribute collections and vice versa.

Attributes in Typed Attributed Graphs can implicitly have multiple values, even multiples of the same value. In graphs with attribute collections multisets can be specified as an attribute type, this must be done explicitly though. While careful construction of graphs and rules in Typed Attributed Graphs can prevent attributes from unexpectedly having multiple values, graphs with attribute collections ensure that an attribute has only a single value. Sets or other data types that contain multiple values must be specified as attribute types and can still be used in graphs with attribute collections.

Graphs with attribute collections can similarly be expressed in the graphs used in GP. Since graphs with attribute collections allow matches even if only a part of a node's or edge's attributes have been specified, the number of rules needed in this approach can be significantly smaller. Additionally its easier to avoid accidental interference between otherwise unrelated rules.

We have presented a useful new concept for attribution in graph transformation. Although it doesn't expand on the expressiveness of Typed Attributed Graphs or the graphs used in GP, it does offer a more comfortable approach that makers it harder to make mistakes in specifying attributed graphs or rules for these graphs. The concept offers:

- The flexibility to be used with either typed or untyped graphs.
- A clear separation between graph and attributes.
- Attributes that behave in an intuitive manner.

In future work the following problems could be explored further:

- **Investigation of $\mathcal{M},\mathcal{N}$-adhesiveness:** The conjecture that graphs with attribute collections are $\mathcal{M},\mathcal{N}$-adhesive could be proven. This would allow the approach to use existing and future results for this class of categories instead of forcing us to revisit all of these questions.
- **Computations over attribute values:** Currently there is no way to include computations over attribute values in rules over graphs with attribute collections. Here an approach similar to the rule schemata in GP could be useful.
- **Constraints over graphs with attribute collections:** Constraints over graphs with attribute collections could be formulated as nested graph conditions. It would not be possible however, to include constraints over attribute values in that setting unless these conditions are expanded in this direction.

Bibliography

[AHS09] Jirí Adámek, Horst Herrlich, and George E. Strecker. *Abstract and Concrete Categories - The Joy of Cats*. Dover Publications, 2009.

[EEKR99] Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 2: Applications, Languages and Tools. World Scientific, 1999.

[EEPT06] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. *Fundamentals of Algebraic Graph Transformation*. Springer, 2006.

[EGH10] Hartmut Ehrig, Ulrike Golas, and Frank Hermann. Categorical frameworks for graph transformation and HLR systems based on the DPO approach. *Bulletin of the EATCS*, 112:111–121, 2010.

[EGH+12] Hartmut Ehrig, Ulrike Golas, Annegret Habel, Leen Lambers, and Fernando Orejas. $\mathcal{M}$-adhesive transformation systems with nested application conditions. Part 2: Embedding, critical pairs and local confluence. *Fundamenta Informaticae*, 118:35–63, 2012.

[EGH+13] Hartmut Ehrig, Ulrike Golas, Annegret Habel, Leen Lambers, and Fernando Orejas. $\mathcal{M}$-adhesive transformation systems with nested application conditions. Part 1: Parallelism, concurrency and amalgamation. *Mathematical Structures in Computer Science*, 23, 2013.

[Ehr79] Hartmut Ehrig. Introduction to the algebraic theory of graph grammars (a survey). In *Proceedings of the International Workshop on Graph-Grammars and Their Application to Computer Science and Biology*, volume 73 of *Lecture Notes in Computer Science*, pages 1–69. Springer, 1979.

[EK79] Hartmut Ehrig and Hans-Jörg Kreowski. Pushout-properties: An analysis of gluing constructions for graphs. *Mathematische Nachrichten*, 91:135–149, 1979.

[EKMR99] Hartmut Ehrig, Hans-Jörg Kreowski, Ugo Montanari, and Grzegorz Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 3: Concurrency, Parallelism, and Distribution. World Scientific, 1999.

[EM85] Hartmut Ehrig and Bernd Mahr. *Fundamentals of Algebraic Specification 1: Equations und Initial Semantics*, volume 6 of *Monographs in Theoretical Computer Science. An EATCS Series*. Springer, 1985.

[Gol12] Ulrike Golas. A general attribution concept for models in $\mathcal{M}$-adhesive transformation systems. In *Proceedings of the 6th International Conference on Graph Transformations (ICGT'12)*, volume 7562 of *Lecture Notes in Computer Science*, pages 187–202. Springer, 2012.

[HHT96] Annegret Habel, Reiko Heckel, and Gabriele Taentzer. Graph grammars with negative application conditions. *Fundamenta Informaticae*, 26:287–313, 1996.

[HKT02] Reiko Heckel, Jochen Malte Küster, and Gabriele Taentzer. Confluence of typed attributed graph transformation systems. In *Proceedings of the First International Conference on Graph Transformation (ICGT '02)*, volume 2505 of *Lecture Notes in Computer Science*, pages 161–176. Springer, 2002.

[HP02] Annegret Habel and Detlef Plump. Relabelling in graph transformation. In *Proceedings of the First International Conference on Graph Transformation (ICGT '02)*, volume 2505 of *Lecture Notes in Computer Science*, pages 135–147. Springer, 2002.

[HP09] Annegret Habel and Karl-Heinz Pennemann. Correctness of high-level transformation systems relative to nested conditions. *Mathematical Structures in Computer Science*, 19(2):245–296, 2009.

[HP12] Annegret Habel and Detlef Plump. $\mathcal{M},\mathcal{N}$-adhesive transformation systems. In *Graph Transformations (ICGT 2012)*, volume 7562 of *Lecture Notes in Computer Science*, pages 218–233, 2012.

[KR12] Harmen Kastenberg and Arend Rensink. Graph attribution through sub-graphs. Technical Report TR-CTIT-12-27, Centre for Telematics and Information Technology, University of Twente, Enschede, 2012.

[LKW93] Michael Löwe, Martin Korff, and Annika Wagner. An algebraic framework for the transformation of attributed graphs. In *Term graph rewriting*, pages 185–199. John Wiley and Sons Ltd., 1993.

[Ore11] Fernando Orejas. Symbolic graphs for attributed graph constraints. *Journal of Symbolic Computation*, 46(3):294–315, 2011.

[Plu99] Detlef Plump. Computing by graph rewriting. Habilitation thesis, University of Bremen, 1999.

[Plu09] Detlef Plump. The graph programming language GP. In *Proceedings of the Third International Conference on Algebraic Informatics (CAI 2009)*, volume 5725 of *Lecture Notes in Computer Science*, pages 99–122. Springer, 2009.

[Plu11] Detlef Plump. The design of GP 2. In *Proceedings of the 10th International Workshop on Reduction Strategies in Rewriting and Programming*, volume 82 of *Electronic Proceedings in Theoretical Computer Science*, pages 1–16, 2011.

[PS04] Detlef Plump and Sandra Steinert. Towards graph programs for graph algorithms. In *Proceedings of the Second International Conference on Graph Transformation (ICGT '04)*, volume 3256 of *Lecture Notes in Computer Science*, pages 128–143. Springer, 2004.

[Roz97] Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 1: Foundations. World Scientific, 1997.

[Sim11] Harold Simmons. *An Introduction to Category Theory*. Cambridge University Press, 2011.

Hiermit versichere ich, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Außerdem versichere ich, dass ich die allgemeinen Prinzipien wissenschaftlicher Arbeit und Veröffentlichung, wie sie in den Leitlinien guter wissenschaftlicher Praxis der Carl von Ossietzky Universität Oldenburg festgelegt sind, befolgt habe.

Oldenburg, den 10.06.2013

_____