# Algebraic Graph Transformations with Inheritance

Michael Löwe, Harald König, Christoph Schulz, and Marius Schultchen

FHDW Hannover University of Applied Sciences,
Freundallee 15, 30173 Hannover, Germany
{Michael.Loewe,Harald.Koenig,Christoph.Schulz}@fhdw.de,
Marius.Schultchen@web.de

**Abstract.** In this paper, we propose a new approach to inheritance in the context of algebraic graph transformation by providing a suitable categorial framework which reflects the semantics of class-based inheritance in software engineering. Inheritance is modelled by a type graph $T$ that comes equipped with a partial order. Typed graphs are arrows with codomain $T$ which preserve graph structures up to inheritance. Morphisms between typed graphs are "down typing" graph morphisms: An object of class $t$ can be mapped to an object of a subclass of $t$. We prove that this structure is an adhesive HLR category, i. e. pushouts along extremal monomorphisms are "well-behaved". This infers validity of classical results such as the Local Church-Rosser Theorem, the Parallelism Theorem, and the Concurrency Theorem.

**Keywords:** Graph transformation, Inheritance, Adhesive HLR category.

## 1 Introduction

Developing appropriate models to mimic reality has always been an important part of software engineering. However, the relation between coding and modelling has changed over time. Today, model-driven engineering focuses on generating code from appropriately detailed and formalised models, hoping that developing the model and using a mature and well-tested code generator is less error-prone than letting programmers write most of the code themselves. This reasoning, however, is only valid if model development is relatively easy. Typically, different graphical notations help people to structure the problem in various ways. Consequently, *graphs* or graph structures play an important role in software engineering today, compare e.g. the UML [13], a modelling language which is currently the de facto standard for modelling object-oriented systems.

If one looks more closely at object-oriented systems, one realises that it is impossible to analyse or build object-oriented software in an efficient way without making use of specialization or *inheritance*.[1] Therefore, it is sensible to require that the graphical notation supports aspects of inheritance well.

[1] In this paper, we do not differentiate between type specialization (subtyping) and class inheritance, because the differences are mostly relevant in the context of type theory, which we do not discuss, and because most mainstream OOP languages do not differentiate between these concepts.

J. Iyoda and L. de Moura (Eds.): SBMF 2013, LNCS 8195, pp. 211–226, 2013.

On the one side, graphs are well suited for modelling static aspects of software, e.g. the class and inheritance structure. On the other side, behavioural aspects of the system, e.g. state changes, can be modelled using *graph transformations* which formally describe when and how a graph (here: state of an object-oriented system) can change into another graph (here: another system state). Graph transformations, especially algebraic graph transformations based on adhesive HLR categories[2], have been studied for a long time and are a well-known tool in the context of software engineering.

However, if we want to combine a graphical notation supporting inheritance with graph transformations, which therefore have to operate on graphs with inheritance, there are relatively few approaches, which differ in flexibility and "readability". In this papier, we propose a new approach which binds the inheritance hierarchy to the type graph (only); objects are typed by providing a typing morphism into the type graph which preserves the graph structure up to inheritance. Morphisms between graphs are allowed to relate objects of different types as long as the target object is at least as specialised as the source object.[3] Upon this notion of inheritance, we build a suitable category and show that this category is an adhesive HLR category, such that many interesting results from the field of algebraic graph transformations can be applied immediately.

The paper is structured as follows: Section 2 develops some basic notions and defines the category $\mathcal{G}^T$ which is used in the subsequent sections. Sections 3, 4, and 5 analyse the properties of monomorphisms, pushouts, and pullbacks in $\mathcal{G}^T$. In section 6, we prove the main result of this paper, namely that $\mathcal{G}^T$ is an adhesive HLR category. Section 7 demonstrates the usefulness of our approach by means of a practical example. We discuss related approaches in section 8. Finally, section 9 summarises the results and discusses future work.

Due to space limitations, some of the proofs have been omitted. They can be found in [12].

## 2 Basic Definitions

$\mathcal{G}$ denotes the usual category of multi-graphs whose objects $G = (V_G, E_G, s_G : E \to V, t_G : E \to V)$ have vertices, edges, and the usual source and target mappings $s_G, t_G : E \to V$, resp.[4] Morphisms $f : G_1 \to G_2$ are pairs of mappings compatible with the graph structure, i.e. they obey the rules $f \circ s_{G_1} = s_{G_2} \circ f$

[2] Adhesive high-level replacement (HLR) categories introduced in [2, 5] combine high-level replacement systems[4] with the notion of adhesive categories[10] in order to be able to generalize the double pushout transformation approach from graphs to other high-level structures as e. g. Petri nets using a categorial framework. Generally, adhesiveness abstracts from exactness properties like compatibility of union and intersection of sets.

[3] We call this property "down-typing".

[4] These notations will remain fixed in that for any $X \in \mathcal{G}$ we will always write $V_X, E_X, s_X, t_X$ for the constituents of $X$ without defining them explicitly.

and $f \circ t_{G_1} = t_{G_2} \circ f$.[5] If a graph is used as a class diagram, its vertices represent the available classes and its edges model directed associations. We formalise class-inheritance by an additional partial order on the vertices of a type graph:

**Definition 1 (Type Graph).** *A* type graph *is a pair* $(T, \leq)$ *where* $T$ *is a graph and* $\leq \subseteq V \times V$ *is a partial order with a largest element* $O \in V_T$[6].

This definition reflects the basic nature of class models. It still lacks additional annotations like multiplicities, abstractness properties or other constraints. The forthcoming definition of object structures, however, shows that it is reasonable to interpret edges as associations with multiplicity "0..∗" on both ends.

**Definition 2 (Typed Graph).** *Let* $I \in \mathcal{G}$ *and* $(T, \leq)$ *be a type graph. A mapping pair* $(i_V : V_I \to V_T, i_E : E_I \to E_T)$*, written* $i : I \to T$*, is called* $T$-typed graph *if the conditions* (1) *and* (2) *hold:*[7]

$$i \circ s_I \leq s_T \circ i \tag{1}$$

$$i \circ t_I \leq t_T \circ i \tag{2}$$

Condition (1) means that subtypes inherit all attributes of all their super-types. Condition (2) formalises the fact that referenced objects at run-time may appear polymorphically: They may possess any subtype of the corresponding association target, cf. Fig. 1. This concept coincides with the definition of "clan morphism" if the underlying relation $I$ in [9] is a partial order.

In the sequel, the type graph $T := (T, \leq)$ will be fixed, i.e. we speak of "typed graphs" instead of "$T$-typed graphs".

**Definition 3 (Type-Compatible Morphism).** *Given two typed graphs* $i : I \to T, j : J \to T$*, a graph morphism* $m : I \to J$ *is* type-compatible*, written* $m : i \to j$*, if*

$$j \circ m \leq i \tag{3}$$

*on* $V_I$ *and*

$$j \circ m = i \tag{4}$$

*on* $E_I$*. If in* (3) *"*$\leq$*" can be replaced by "*$=$*", m is called* strong*. A strong morphism f from i to j will be denoted* $i \xrightarrow{f} j$ .

It follows that type-compatible morphism can map an "object" of type $c$ to an "object" the type of which is a subtype of $c$. This is especially useful when

[5] Sometimes in the literature the two components $f_V$ and $f_E$ of $f$ are explicitly differentiated. We will not do that, because it will always become clear from the context which component is used.

[6] The letter "O" shall remind of the class "Object" in Java, which is a super class of all other classes, hence the inheritance order's largest object.

[7] If $f, g : X \to Y$ are two mappings into a partially ordered set $Y = (Y, \leq)$, we write $f \leq g$ if $f(x) \leq g(x)$ for all $x \in X$.

matching a graph transformation rule, since one can match many specialised objects by using an object of a general type.

Strong morphisms are closed under composition:

**Proposition 4.** *Let $i : I \to T$, $j : J \to T$, and $k : K \to T$ be some typed graphs, and let $m : i \to j$ and $n : j \to k$ be two strong morphisms. Then $n \circ m$ is also strong.*

*Proof.* $k \circ (n \circ m) = (k \circ n) \circ m = j \circ m = i$. □

**Definition 5 (Category $\mathcal{G}^T$).** *Let $T$ be a type graph. We define $\mathcal{G}^T$ to be the category which has typed graphs as objects and type-compatible morphisms between them as arrows.*

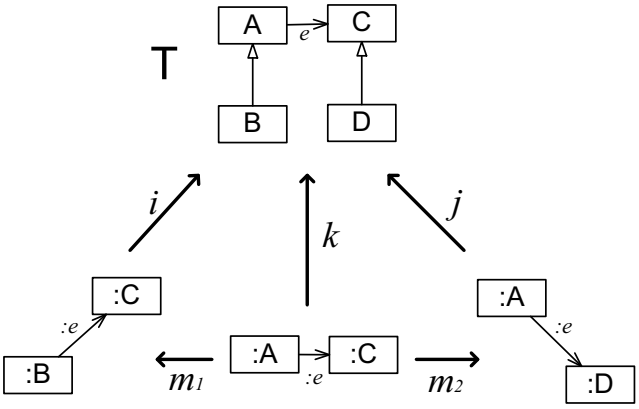



**Fig. 1.** Typed graphs and type-compatible morphisms

The main effects are shown in UML-styled Fig. 1: $T$ is the top graph in which nodes are rectangles and the partial order is depicted by arrows with end-triangles (reflexive elements and the largest element $O$ are not shown). There are three typed graphs $i, j, k$, their typing being highlighted by names $:X$ whenever they map to $X$. Since $B$ inherits association $e$, $i$ is a well-typed object structure. Since $A$-objects may polymorphically be linked to $C$- or $D$-objects $j$ is an admissible typing. Moreover, $m_1$ and $m_2$ are two type-compatible morphisms (e.g.: $m_1(:A) = :B$ yielding $i(m_1(:A)) = B < A = k(:A)$[8]).

In the sequel, we let

$$\tau : \begin{cases} \mathcal{G}^T \to \mathcal{G} \\ (g : G \to T) \xrightarrow{f} (h : H \to T) \;\mapsto\; G \xrightarrow{f} H \end{cases}$$

be the functor which forgets the typing structure.

[8] $<$ being short for: $\leq$ and $\neq$.

## 3 Monomorphisms and Epimorphisms

In order to investigate categorial properties of $\mathcal{G}^T$, we analyse the nature of monomorphisms and epimorphisms: First, a straightforward argument shows that any injective $m : i \to j$ is a monomorphism. The reverse statement is also true, but we need the existence of the largest element $O$ of $\leq$: If $m : g \to h$ is a monomorphisms then $m(v_1) = m(v_2)$ can be detected by mappings $k_1, k_2 : \{:O\} \to G$ with $k_i(:O) = v_i$ $(i \in \{1, 2\})$.[9] If we do not require the existence of a largest element, assume $T$ contains the three types $A$, $B$, and $C$ such that $C < A$ and $C < B$, then the non-injective $m\colon \{:A, :B\} \to \{:C\}$ with $m(:A) = m(:B) = :C$ is a monomorphism, as there do not exist any morphisms $p, q : X \to \{:A, :B\}$ which map some element $x \in X$ to $:A$ and $:B$, resp., due to the missing common supertype of $A$ and $B$.

Surjective morphisms coincide with the class of epimorphisms. In contrast to monomorphisms, however, the proof of this fact does not make use of the largest element and is proven in the same way as the corresponding fact in $\mathcal{G}$.

**Proposition 6.** *Epimorphisms of $\mathcal{G}^T$ are exactly the surjective morphisms. Monomorphisms of $\mathcal{G}^T$ are exactly the injective morphisms.*

Conventionally, in category theory, *extremal* monomorphisms are often the right choice if (ordinary) monomorphisms do not represent embeddings: A monomorphism $m$ is said to be *extremal*, if any decomposition $m = m' \circ f$ with an epimorphism $f$ already forces $f$ to be an isomorphism. In $\mathcal{G}^T$ a morphism $m : \{:B\} \to \{:A\}$ with $A < B$ is monic and epic (cf. Proposition 6) but no isomorphism, because a hypothetical inverse $n$ would have to "upcast" $(n(:A) =: B)$, which is not possible. Thus $m$ is not extremal, because $m = id \circ m$.[10]

**Proposition 7 (Strong Monos and Extremal Monos coincide).** *A monomorphism in $\mathcal{G}^T$ is extremal if and only if it is strong.*

Because of this result, it is reasonable to denote an extremal mono $m$ from $i$ to $j$ by $i \rightarrowtail^{m} j$ .

## 4 Pushouts

In order to define and apply double-pushout graph transformation rules in the category $\mathcal{G}^T$, we need to analyse how pushouts can be constructed. The first observation is that pushouts do not always exist: Let $T$ be the discrete graph[11] with $V_T = \{O, B, C\}$ and $\leq = \{(B, O), (C, O)\}$ together with reflexive pairs. Then

$$\{:C\} \longleftarrow \{:O\} \longrightarrow \{:B\}$$

[9] The notation $\{x\}$ is short for the graph $(\{x\}, \emptyset, \emptyset, \emptyset)$.

[10] In topoi, an epic monomorphisms necessarily becomes an isomorphism. Hence this example shows that $\mathcal{G}^T$ is not a topos. In the next sections there will be many other aspects detecting this property (e.g. the fact that some limits and some more co-limits do not exist).

[11] A graph with empty edge set.

obviously possesses no pushout. Even if one restricts down-typing to at most one of the given morphisms, pushouts along monomorphisms need not exist, because

$$\{1{:}B, 2{:}C\} \longleftarrow \{1{:}O, 2{:}O\} \longrightarrow \{12{:}O\}\ ,$$

where the left leg maps according to the numbers (and hence is monic) and where the right leg identifies the objects $1{:}O$ and $2{:}O$ by mapping them to $12{:}O$, does not admit a pushout. This behaviour has its roots in the fact that $B$ and $C$ are incomparable and do not possess a common subtype.

Our goal is to find a feasible criterion for a span

$$j \xleftarrow{\beta} g \xrightarrow{\alpha} h \tag{5}$$

to admit a pushout. For this we denote with $\bigwedge A$ the *greatest lower bound* of a subset $A \subseteq V_T$ if it exists[12]. Let furthermore $G := \tau(g)$, $H := \tau(h)$, and $J := \tau(j)$ with the above introduced forgetful functor. We denote with $[h, j] : H + J \to T$ the disjoint union of $h$ and $j$ and we need the usual relation

$$\sim := \{(\alpha(x), \beta(x)) \mid x \in G\} \tag{6}$$

on $H+J$, for which $\equiv$ denotes the smallest (sortwise) equivalence on $H+J$ which contains $\sim$. An equivalence class of $\equiv$ will be written $[v]_\equiv$ or $[v]$. Let $H +_G J := (H + J)/_\equiv$ together with the canonical graph morphisms $\overline{\alpha} : J \to H +_G J$ and $\overline{\beta} : H \to H +_G J$ (which map $v$ to $[v]_\equiv$) that make up the $\mathcal{G}$-pushout of $\alpha$ and $\beta$.

**Theorem 8 (Characterisation of Pushouts).** *The span* (5) *admits a pushout in* $\mathcal{G}^T$ *if and only if*

$$\forall v \in V_{H+J} : \bigwedge \{[h, j](x) \mid x \in [v]_\equiv\}$$

*exists. If this condition is met, the square*

$$\begin{array}{ccc} g & \xrightarrow{\alpha} & h \\ {\scriptstyle\beta}\downarrow & & \downarrow{\scriptstyle\overline{\beta}} \\ j & \xrightarrow[\overline{\alpha}]{} & p \end{array} \tag{7}$$

*is a pushout in* $\mathcal{G}^T$*, where* $p : H +_G J \to T$ *is defined by*

$$p([v]) = \bigwedge \{[h, j](x) \mid x \in [v]\}$$

*on vertices and* $p([e]) = [h, j](e)$ *on edges.*

**Corollary 9.** $\mathcal{G}^T$ *has all pushouts along extremal monomorphisms. In such a pushout the extremal mono is preserved under the pushout.*

[12] The notation $\bigwedge$ shall remind of "intersection" (of sets): For any set $X$, any indexed set $(Y_i)_{i\in I}$ with $Y_i \in (\wp(X), \subseteq)$ always has a greatest lower bound, namely $\bigcap_{i\in I} Y_i$.

*Proof.* If $\alpha$ is an extremal mono, it is a strong monomorphism by Proposition 7. This means, that for any $v \in V_{H+J}$ the set $[v]$ is a singleton (if $v \in V_H$ is not in the image of $\alpha$), or it is of the form $\{\alpha(y) \mid y \in \beta^{-1}(\beta(x))\} \cup \{\beta(x)\}$ for some $x \in V_G$. In the first case, the greatest lower bound is $h(v)$, in the latter case, by strongness, it is $j(\beta(x))$. Thus, by Theorem 8, the pushout can be constructed with the usual construction in $\mathcal{G}$ such that $\overline{\alpha}$ becomes an embedding, hence a strong (thus extremal) mono. □

Theorem 8 can be alternatively formulated as

**Corollary 10.** *A commutative diagram* $\mathcal{D} =$

$$\begin{array}{ccc} g & \xrightarrow{\alpha} & h \\ {\scriptstyle\beta}\downarrow & & \downarrow{\scriptstyle\delta} \\ j & \xrightarrow[\gamma]{} & q \end{array}$$

*s.t.* $\tau(\mathcal{D})$ *is a pushout in* $\mathcal{G}$*, is a pushout in* $\mathcal{G}^T \iff \forall v \in V_{\tau(q)} : q(v) = \bigwedge \{[h,j](x) \mid [\gamma,\delta](x) = v\}$.

*Proof.* Let $i\colon \tau(q) \to \tau(p)$ be the canonical $\mathcal{G}$-isomorphism between the given pushout $\tau(\mathcal{D})$ and the canonical pushout in $\mathcal{G}$ ($\tau$ applied to (7)). Then for all $x \in H + J$, $v \in V_{\tau(q)}$ we obtain $[\gamma,\delta](x) = v \Leftrightarrow i([\gamma,\delta](x)) = i(v) \Leftrightarrow [\overline{\alpha},\overline{\beta}](x) = i(v)$, thus

$$[\gamma,\delta](x) = v \Leftrightarrow x \in i(v) \tag{8}$$

"⇒": By Theorem 8, $\bigwedge S_v$ exists and (7) is pushout. Thus, $i$ is a $\mathcal{G}^T$-isomorphism. Then $q = p \circ i$ and (8) yield $q(v) = p(i(v)) = \bigwedge \{[h,j](x) \mid x \in i(v)\} = \bigwedge \{[h,j](x) \mid [\gamma,\delta](x) = v\}$.

"⇐": The definition of $q$ and (8) yield the characterising condition of Theorem 8. Hence (7) is a $\mathcal{G}^T$-pushout. Moreover, by (8) and the definition of $p$ we have $p(i(v)) = \bigwedge \{[h,j](x) \mid x \in i(v)\} = \bigwedge \{[h,j](x) \mid [\gamma,\delta](x) = v\} = q(v)$, such that $i$ is a $\mathcal{G}^T$-isomorphism and the given square is a $\mathcal{G}^T$-pushout. □

Note that the results of this section remain true even if we do not claim the existence of a largest element $O$.

## 5 Pullbacks

In this section we characterise those co-spans of $\mathcal{G}^T$ which admit pullbacks. The situation is *not* dual to the situation in Section 4 because of the existence of the largest element: If $T$ consists of nodes $\{A, B, C, O\}$ with no edges where $\leq$ is generated from $\{(A,B),(A,C),(B,O),(C,O)\}$, the co-span

$$\{:C\} \longrightarrow \{:A\} \longleftarrow \{:B\}$$

possesses the pullback

$$\{:C\} \longleftarrow \{:O\} \longrightarrow \{:B\} \ .$$

But pullback construction fails in more complex situations: Let a type graph be given by the class diagram in Fig. 2, in which the partial order is generated by the depicted arrows.

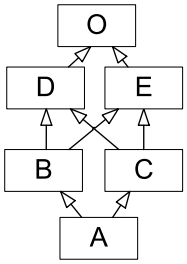



**Fig. 2.** A type graph

Then the co-span

$$\{:C\} \longrightarrow \{:A\} \longleftarrow \{:B\}$$

admits no pullback, because there are two incompatible candidates, namely the spans

$$\{:C\} \longleftarrow \{:D\} \longrightarrow \{:B\} \quad \text{and} \quad \{:C\} \longleftarrow \{:E\} \longrightarrow \{:B\} \ ,$$

and a minimal candidate

$$\{:C\} \longleftarrow \{:D,:E\} \longrightarrow \{:B\} \ , \tag{9}$$

for which, however, two different mediators exist from

$$\{:C\} \longleftarrow \{:O\} \longrightarrow \{:B\} \ .$$

This example shows that it seems to be difficult to find a feasible criterion for a pullback to exist without claiming the existence of a largest element: If we omitted $O$ in Figure 2, there would indeed be a pullback, namely the span (9) (which seems to be weird because the middle graph possesses two vertices – note that monos are still preserved by pullbacks because both morphisms in (9) are now monos, see the example in the first paragraph of Section 3).

In order to avoid these degenerate limits we return to the original situation in which $O$ exists. We want to find a necessary and sufficient criterion for a co-span

$$j \xrightarrow{\beta} g \xleftarrow{\alpha} h \tag{10}$$

to admit a pullback which is feasible enough to be used in practical contexts. It turns out that the existence of pullbacks heavily depends on the existence of *least upper bounds* of two nodes of $T$. We use the notation $B \vee C$ to denote the least upper bound if it exists.[13]

We abbreviate $J := \tau(j)$, $G := \tau(g)$, and $H := \tau(h)$. $H \times_G J$ is the pullback object of $\alpha$ and $\beta$ in $\mathcal{G}$ together with projections $\pi_1 : H \times_G J \to H$ and $\pi_2 : H \times_G J \to J$. It turns out that the two above examples fully characterise the limitations for the existence of pullbacks:

**Theorem 11 (Characterisation of Pullbacks).** *The co-span* (10) *admits a pullback if and only if*

$$\forall (v_1, v_2) \in V_{H \times_G J} : h(v_1) \vee j(v_2) \text{ exists.}$$

*If this condition is met, the square*

$$\begin{array}{ccc} g & \xleftarrow{\alpha} & h \\ {\scriptstyle\beta}\uparrow & & \uparrow{\scriptstyle\pi_1} \\ j & \xleftarrow[\pi_2]{} & p \end{array} \tag{11}$$

*is a pullback in $\mathcal{G}^T$, where $p : H \times_G J \to T$ is defined by*

$$p(v_1, v_2) = h(v_1) \vee j(v_2)$$

*on vertices and*

$$p(e_1, e_2) = h(e_1)(= j(e_2))$$

*on edges.*

We obtain the following consequences:

**Corollary 12.**

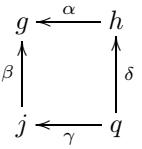
*(1) If in* (10) *at least one morphism is strong, the pullback exists.*
*(2) If in $(T, \leq)$ all pairs have a least upper bound, all pullbacks exist.*
*(3) If $T$ is finite and $\leq$ represents a hierarchy, i.e. if each node in $V_T - \{O\}$ has exactly one direct super node*[14]*, all pullbacks exist.*
*(4) Extremal monomorphisms as well as strong morphisms are preserved under pullbacks.*

*Proof.* 12(1), 12(2), and 12(4) are immediate consequences of Theorem 11 and the fact that pullbacks preserve monos in $\mathcal{G}$. 12(3) can be easily proved by induction over path lengths from $h(v_1)$ to $O$ and $j(v_2)$ to O, respectively. □

Theorem 11 can be alternatively formulated:

[13] The notation $\vee$ shall remind of "union" (of sets): For any set $X$, any two elements $Y_1, Y_2 \in (\wp(X), \subseteq)$ have always a least upper bound, namely $Y_1 \cup Y_2$.
[14] As is the case in each programming language that prohibits multiple inheritance.

**Corollary 13.** *A commutative diagram* $\mathcal{D} =$




*s.t.* $\tau(\mathcal{D})$ *is a pullback in* $\mathcal{G}$*, is a pullback in* $\mathcal{G}^T$ *if and only if* $\forall z \in V_{\tau(q)} : q(z) = h(\delta(z)) \vee j(\gamma(z))$.

*Proof.* "⇒": $\mathcal{D}$ and (11) yield a canonical $\mathcal{G}^T$-isomorphism $i\colon q \to p$, such that for $z \in V_{\tau(q)}$, $q(z) = p(i(z)) = p(\pi_1(i(z)), \pi_2(i(z))) = p(\delta(z), \gamma(z)) = h(\delta(z)) \vee j(\gamma(z))$.

"⇐": Let $i\colon \tau(q) \to \tau(p)$ be the canonical $\mathcal{G}$-isomorphism between $\tau(\mathcal{D})$ and $\tau$ applied to (11). Then $q(z) = h(\pi_1(i(z))) \vee j(\pi_2(i(z))) = p(\pi_1(i(z)), \pi_2(i(z))) = p(i(z))$, thus $i$ is $\mathcal{G}^T$-isomorphism, hence $\mathcal{D}$ is $\mathcal{G}^T$-pullback. □

## 6 Adhesiveness

In this section, we intend to show that $\mathcal{G}^T$ is an adhesive HLR category for the class $\mathcal{M}$ of all extremal monomorphisms.

**Theorem 14.** $\mathcal{G}^T$ *is an adhesive HLR category for the class* $\mathcal{M}$ *of all extremal monomorphisms.*

*Proof.* Due to Prop. 4 and [1, Prop. 7.62(2)], $\mathcal{M}$ is closed under composition (also with isomorphisms) and decomposition[15], resp. Moreover, $\mathcal{G}^T$ has all pushouts and pullbacks along $\mathcal{M}$, and $\mathcal{M}$-morphisms are preserved under pushouts and pullbacks (cf. Corollaries 9 and 12). It remains to show that pushouts along $\mathcal{M}$-morphisms are VK squares, cf. [2, Def. 4.9]. Let therefore a commutative cube be given with a pushout along the extremal mono $\alpha$ at the bottom and two rear pullbacks (Fig. 3). From Corollaries 9 and 12(4) we can deduce that $\overline{\alpha}$ and $\alpha'$ are extremal monos, too (which is already indicated in Fig. 3).

We now show that the top face in Fig. 3 is a pushout $\iff$ the two front faces are pullbacks.

"⇒": By Corollary 9 and Proposition 7 $\overline{\alpha}'$ is strong. Applying $\tau$ to the cube shows that front and right faces are pullbacks in $\mathcal{G}$ (by adhesiveness of $\mathcal{G}$). By Corollary 13 it suffices to show that $c = d \circ \overline{\alpha}' \vee h \circ i_1$ and $b = d \circ \overline{\beta}' \vee i \circ i_2$ on vertices. The first statement follows immediately, because $\overline{\alpha}'$ is strong and thus for any $x \in V_{\tau(c)}$

$$c(x) = d(\overline{\alpha}'(x)) \text{ and } h(i_1(x)) \leq c(x)$$

s.t. $c(x) = d(\overline{\alpha}'(x)) \vee h(i_1(x))$.

[15] "Decomposition" means: $g \circ m$ an extremal mono $\Rightarrow$ $m$ an extremal mono.

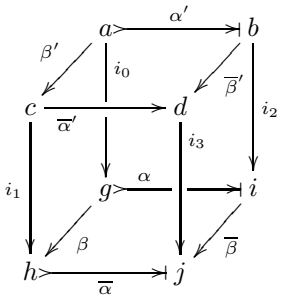



**Fig. 3.** Commutative cube

In order to show the second statement, we let $z \in V_{\tau(b)}$ be arbitrary. If $z$ is in the image of $\alpha'$, i.e. $z = \alpha'(z')$ for some $z' \in V_{\tau(a)}$, we obtain

$$\begin{aligned} b(z) &= a(z') && \text{Strongness of } \alpha' \\ &= c(\beta'(z')) \vee g(i_0(z')) && \text{Cor. 13 applied to left rear pullback} \\ &= d(\overline{\alpha}'(\beta'(z'))) \vee i(\alpha(i_0(z'))) && \text{Strongness of } \overline{\alpha}' \text{ and } \alpha \\ &= d(\overline{\beta}'(z)) \vee i(i_2(z)) && \text{Top and right rear faces commute.} \end{aligned}$$

If $z$ is not in the image of $\alpha'$, the pushout construction of Theorem 8 shows that $\overline{\beta}'(z)$ is not in the image of $\overline{\alpha}'$, such that by Corollary 10

$$b(z) = d(\overline{\beta}'(z))$$

which yields $b(z) = d(\overline{\beta}'(z)) \vee i(i_2(z))$, because $i(i_2(z)) \leq b(z)$.

"$\Leftarrow$": Assume all four side faces are pullbacks. By adhesiveness of $\mathcal{G}$ the top face is a pushout in $\mathcal{G}$ such that by Corollary 10 it suffices to show that $d \circ \overline{\alpha}' = c$ and $d \circ \overline{\beta}' = b$ on $\tau(b) - \alpha'(\tau(a))$. The first statement is immediate because $\overline{\alpha}'$ is strong by Corollary 12(4).

Let $z \in \tau(b) - \alpha'(\tau(a))$. Because the rear face is a pullback, $i_2(z) \in \tau(i) - \alpha(\tau(g))$. By the pushout property of the bottom face, Corollary 10 yields $j(\overline{\beta}(i_2(z))) = i(i_2(z))$. Thus by Corollary 13

$$b(z) = d(\overline{\beta}'(z)) \vee i(i_2(z)) = d(\overline{\beta}'(z)) \vee j(\overline{\beta}(i_2(z))) = d(\overline{\beta}'(z)) \vee j(i_3(\overline{\beta}'(z)))$$

But $j \circ i_3 \leq d$, such that

$$b(z) = d(\overline{\beta}'(z))$$

as desired. □

**Proposition 15.** *In $\mathcal{G}^T$, binary coproducts are compatible with $\mathcal{M}$.*

We conclude this section with the main result of this paper: If all graph transformation rules in $\mathcal{G}^T$ are spans $L \rightarrowtail\!\!\!\!\leftarrow K \rightarrowtail R$ of two extremal monomorphisms, we obtain the well-known concurrency theorems for the DPO-approach:

**Corollary 16.** *The following results for graph transformation based on $\mathcal{G}^T$ and the class $M$ of all extremal monomorphisms are valid due to Theorem 14 and Proposition 15:*

- *Local Church Rosser Theorem for pairwise analysis of sequential and parallel independence [2, Thm. 5.12]*
- *Parallelism Theorem for applying independent rules and transformations in parallel [2, Thm. 5.18]*
- *Concurrency Theorem for applying edge-related dependent rules simultaneously [2, Thm. 5.23]*

## 7 Example

Consider a simple model of a file system (Fig. 4). On the one hand, we have the file system itself and directories, which both can contain other file system objects and, thus, are called *containers*. On the other hand, we have directories and files, which are part of a (unique) container and, thus, are called *containees*.[16] Directories can be created by the rule in Fig. 5a (file creation is done by a similar rule). Fig. 5b allows to delete a file system object by unlinking it from its container.[17]

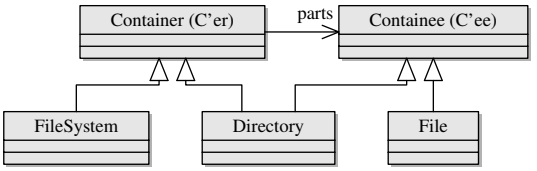
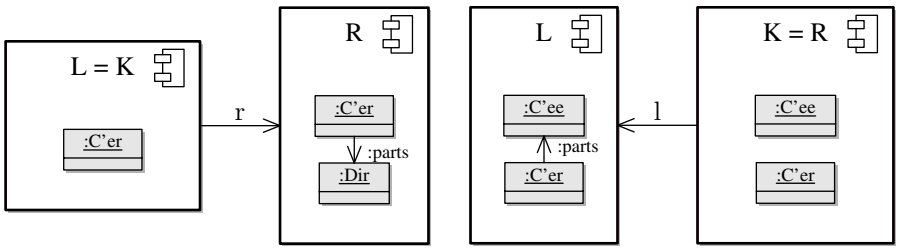



**Fig. 4.** File system model




(a) Rule "Create Directory" (b) Rule "Delete Object"

**Fig. 5.** Example rules

[16] We do not cover container uniqueness in this example.

[17] Some sort of a garbage collector is needed to physically delete all objects that are not part of any container. These rules are not shown in this example but can be modelled by using NACs (negative application conditions) [2].

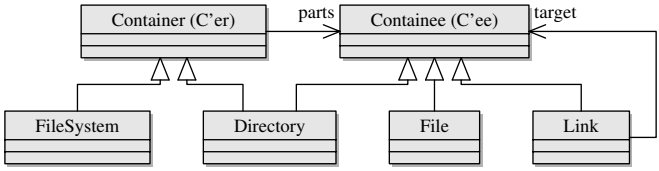



**Fig. 6.** File system model with links

Now we extend the file system model by links (see Fig. 6).[18] Creating a link is handled by the rule in Fig. 7. The rule in Fig. 8 allows to retarget a link; the figure also demonstrates how the rule can be applied to a concrete instance $G$.

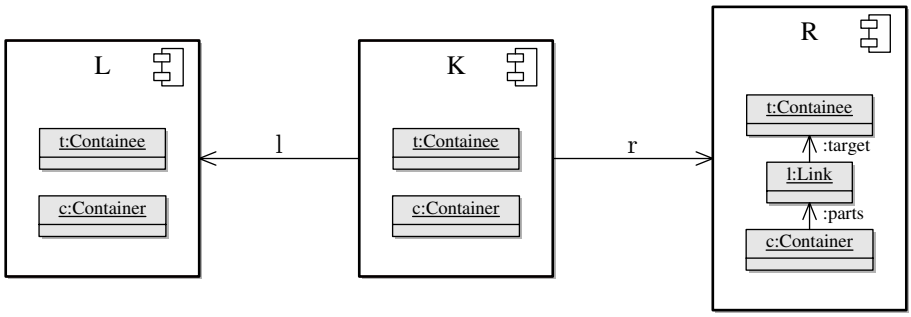



**Fig. 7.** Rule "Create Link"

In this example, the advantage of being able to define a graph transformation rule on an abstract level should have become clear. For each containee, we only need *one* rule to create the containee, instead of one rule for each concrete container. It is not necessary to change or extend the rule to delete a file system object. Retargeting a link can be specified by *one* single rule (independent of whether the old and new targets of the link are directories, files, or links), whereas without any abstraction, nine rules would be necessary.

## 8 Related Work

There are relatively few approaches that integrate inheritance or inheritance-like features into graph transformation. Most of these research lines are based on algebraic graph transformation, either on the double pushout approach [2] or on the single-pushout approach [11].

[18] A (symbolic) *link* is a reference to another file system object, which can be a link itself. Typically, operating systems confine the link depth in order to sort out circular references.

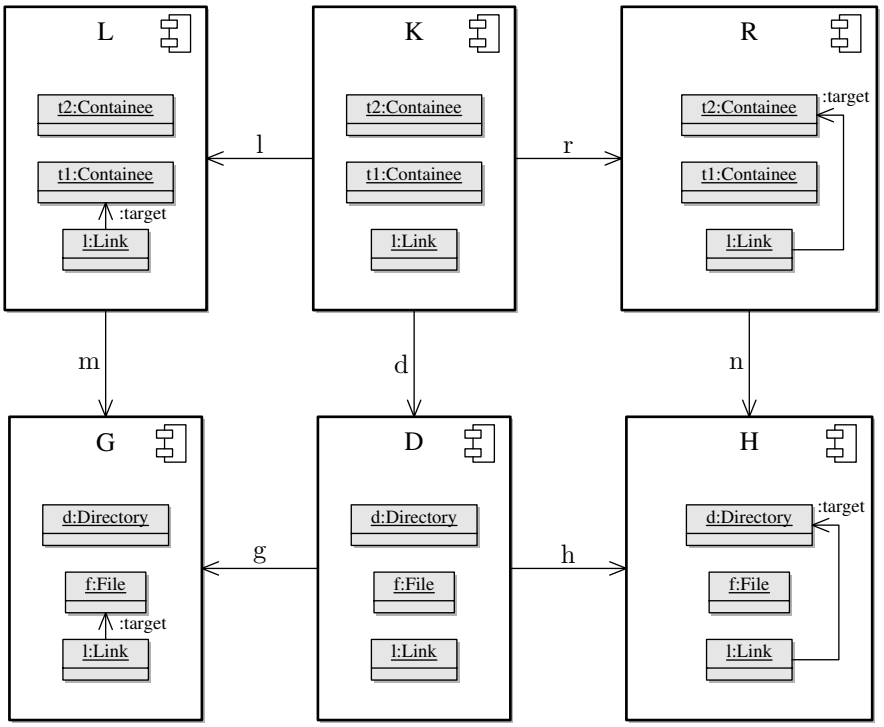



**Fig. 8.** Rule "Retarget Link" and a sample application

H. Ehrig et al. [2] introduce inheritance as an additional set of inheritance edges between vertices in the type graph. This structure is not required to be hierarchical. Cycle-freeness is not necessary, since they do not work with the original type graph. Instead they use a canonically flattened type structure, in which inheritance edges are removed and some of the other edges are copied to the "more special" vertices. By this reduction, they get rid of inheritance and are able to reestablish their theoretical results. E. Guerra and J. de Lara [8] extend this approach to inheritance between vertices *and* edges.

F. Hermann et al. [9] avoid the flattening and define a weak adhesive category based on the original type graph *with* inheritance structure. The morphisms in the rules are restricted to those which reflect the subtype structure: if an image of a morphism possesses subtypes, all these subtypes have pre-images under the morphism. This feature considerably restricts applicability to examples as in section 7.

U. Golas et al. [7] also avoid the flattening process. They, however, require that the paths along inheritance edges are cycle-free (hierarchy) and that every vertex has at most one abstraction (single inheritance). For this set-up, they devise an adhesive categorial framework comparable to our approach which is, however, restricted to single inheritance.

A. P. L. Ferreira and L. Ribeiro [6] introduced a graph transformation framework for object-oriented programming based on single-pushout rewriting. They allow vertex and edge specialisations in the type graph and show that suitably restricted situations admit pushouts of partial morphisms. Their framework is shown adequate as a model for object-oriented systems. They do not address further categorial properties like adhesiveness.

## 9 Conclusion

Since our introduced formal foundation is enriched with *inheritance*, it is better capable of modelling static structures of object-oriented systems. Although there have been similar approaches (see Section 8), the innovation of our work is the proof that our framework is well-behaved w.r.t. the interplay of pushouts and pullbacks (adhesiveness). Consequently, important theorems on concurrent applications of graph transformation rules are valid. This enables controlled manipulation and evolution of object graphs with inheritance based on the general theory of algebraic graph transformations.

The presented inheritance concept increases the value of graph transformation techniques for applications. But beside the specification of associations (i.e. admissible object linkings) and inheritance (property transfer between classes), (UML-)class diagrams also specify attributes, object containment relations (composition), instantiation restrictions (abstract classes), arbitrary mulitiplicities, and other limiting constraints. Hence, there is one important direction for future research: Is adhesiveness invariant under enlargements of $\mathcal{G}^T$ such as introduction of attributes [3], addition of abstractness predicate, or sketched OCL[19] constraints [14]?

It is also a goal of forthcoming research to define *single pushout rewriting* [11] with inheritance: For this, transformation rules $r : L \to R$ with $r$ a *partial* type-compatible morphism have to be introduced, conflict freeness and more generally "deletion injectivity" have to be made precise. In addition to static inheritance features introduced above, we conjecture that simple inclusion relations of rules lead to a better formal understanding of *overwriting* (a rule by a larger rule). Consequently, the effect of replacing an application of a rule $r$ by a super rule $r'$ could also be interpreted as a negative application condition [2], if $r'$ is the identity.

Finally, the overall research goal must be to integrate all important object-orientation concepts to graph transformations, which will result in a comprehensive visual formal framework to be applied to object-oriented modelling and meta-modelling.

## References

[1] Adámek, J., Herrlich, H., Strecker, G.E.: Abstract and Concrete Categories: The Joy of Cats. Free Software Foundation (2004)

[19] Object Constraint Language.

[2] Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Springer (2006)
[3] Ehrig, H., Prange, U., Taentzer, G.: Fundamental theory for typed attributed graph transformation. In: Ehrig, H., Engels, G., Parisi-Presicce, F., Rozenberg, G. (eds.) ICGT 2004. LNCS, vol. 3256, pp. 161–177. Springer, Heidelberg (2004)
[4] Ehrig, H., Habel, A., Kreowski, H.J., Parisi-Presicce, F.: Parallelism and concurrency in high-level replacement systems. Mathematical Structures in Computer Science 1, 361–404 (1991)
[5] Ehrig, H., Padberg, J., Prange, U., Habel, A.: Adhesive high-level replacement systems: A new categorical framework for graph transformation. Fundam. Inf. 74(1), 1–29 (2006)
[6] Lüdtke Ferreira, A.P., Ribeiro, L.: Derivations in object-oriented graph grammars. In: Ehrig, H., Engels, G., Parisi-Presicce, F., Rozenberg, G. (eds.) ICGT 2004. LNCS, vol. 3256, pp. 416–430. Springer, Heidelberg (2004)
[7] Golas, U., Lambers, L., Ehrig, H., Orejas, F.: Attributed graph transformation with inheritance: Efficient conflict detection and local confluence analysis using abstract critical pairs. Theoretical Computer Science 424, 46–68 (2012)
[8] Guerra, E., de Lara, J.: Attributed typed triple graph transformation with inheritance in the Double Pushout approach. Tech. Rep. UC3M-TR-CS-06- 01, Universidad Carlos III de Madrid (2006)
[9] Hermann, F., Ehrig, H., Ermel, C.: Transformation of type graphs with inheritance for ensuring security in e-government networks. In: Chechik, M., Wirsing, M. (eds.) FASE 2009. LNCS, vol. 5503, pp. 325–339. Springer, Heidelberg (2009)
[10] Lack, S., Sobociński, P.: Adhesive categories. In: Walukiewicz, I. (ed.) FOSSACS 2004. LNCS, vol. 2987, pp. 273–288. Springer, Heidelberg (2004)
[11] Löwe, M.: Algebraic approach to single-pushout graph transformation. Theoret. Comput. Sci. 109, 181–224 (1993)
[12] Löwe, M., König, H., Schulz, C., Schultchen, M.: Algebraic graph transformations with inheritance. Tech. Rep. 02013/03, University of Applied Sciences, FHDW Hannover (2013)
[13] Pilone, D.: UML 2.0 in a Nutshell. O'Reilly (2006)
[14] Rutle, A., Wolter, U., Lamo, Y.: A diagrammatic approach to model transformations. In: Proceedings of the 2008 Euro American Conference on Telematics and Information Systems (EATIS 2008), pp. 1–8. ACM (2008)