Polymorphic Single-Pushout Graph Transformation

Michael Löwe, Harald König, and Christoph Schulz

FHDW Hannover, Freundallee 15, 30173 Hannover

Abstract. The paper extends single-pushout graph transformation by polymorphism, a key concept in object-oriented design. The notions *subrule* and *remainder*, well-known in single-pushout rewriting, are applied in order to model dynamic rule extension and type dependent rule application. This extension mechanism qualifies graph transformation as a modelling technique for extendable frameworks. Therefore, it contributes to the applicability of graph transformation in software engineering.

1 Introduction

Algebraic graph transformation has been extended by many object-oriented modelling concepts, for example types and attributes, compare [2]. However, the central structure of object-orientation, namely inheritance with polymorphism, has not been completely integrated yet. We propose a concept for polymorphism in the single pushout approach [13].

Object-oriented polymorphism is a concept that allows several methods for the same operation. The late-binding mechanism of the corresponding runtime system selects the "best" method dependent on the types of the involved objects. Typically the most special of all fitting methods is selected and executed. We transfer this concept to typed algebraic graph transformation systems without attributes. Here, transformation rules play the role of methods. In order to mimic the late binding mechanism of object-orientation, we need a specialisation hierarchy on types *and* on methods, i. e. on transformation rules.

In contrast to [2], we model the specialisation hierarchy on types by a partial order in the type graph and do not allow cycles in the specialisation relation. Again in contrast to [2], we do not get rid of the specialisation relation by a flattening process. Instead, we use it to design a category $\mathbb{G}^T$ of ordinary directed graphs typed in the type graph T where morphisms are allowed to map *up to specialisation*: A morphism can map a vertex v to any vertex the type of which is a specialisation of the type of v, compare Section 3 and [16].

The hierarchy on transformation rules is modelled by subrule relations, i. e. a rule t is more special than t', if t' is a subrule of t. The subrule concept is borrowed from the theory of single-pushout rewriting, compare Section 2. It perfectly models polymorphism, since general results guarantee that the behaviour of t extends the behaviour of t', if t' is a subrule of t, in the following sense:

S. Gnesi and A. Rensink (Eds.): FASE 2014, LNCS 8411, pp. 355–369, 2014.

Every transformation with t can be decomposed into a transformation with t' followed by a transformation with a uniquely determined remainder $t - t'$.

The paper is organised as follows. Section 2 recapitulates the theory of single-pushout graph transformation [9,11,13,14], especially the concepts sub-rule, remainder, and amalgamated rule. Section 3 summarises the results of [16]. It introduces the basic category $\mathbb{G}^T$ of graphs typed in a type graph T with inheritance hierarchy. In Section 4, Theorem 25 shows the sufficient conditions for $\mathbb{G}^T$ to admit single-pushout rewriting presented in Section 2. Section 4 also introduces the new concept of a polymorphic graph transformation system. It allows type dependent rule selection and application. The increase in the expressive power is demonstrated by some examples. Section 6 discusses topics of future research. We assume that the reader has basic knowledge of category theory.

2 Single-Pushout Transformation Framework

Single-pushout graph transformation simplifies the classical double-pushout approach [2,3] with the help of a category that represents double-pushout rules $(L \overset{l}{\hookleftarrow} K \overset{r}{\rightarrowtail} R)$ as partial morphisms.

A *span base* $(\mathcal{C}, \mathcal{M})$ consists of a category $\mathcal{C}$ and a subclass $\mathcal{M}$ of the morphisms of $\mathcal{C}$ such that:

1. $\mathcal{M}$ contains all isomorphisms of $\mathcal{C}$.
2. $\mathcal{M}$ is closed under composition.
3. $\mathcal{M}$ is prefix-closed: $q \circ p \in \mathcal{M}$ and $q \in \mathcal{M} \implies p \in \mathcal{M}$.
4. $\mathcal{C}$ has all pullbacks for all pairs of morphisms (p, q) with $p \in \mathcal{M}$.
5. Pullbacks in $\mathcal{C}$ are $\mathcal{M}$-closed: (p^*, q^*) pullback of (p, q), $p \in \mathcal{M} \implies p^* \in \mathcal{M}$.

Given an object $A \in \mathcal{C}$, $\mathcal{C} \downarrow_{\mathcal{M}} A$ denotes the restriction of the comma category $\mathcal{C} \downarrow A$ to $\mathcal{M}$-morphisms.[1] The conditions 1, 2, and 3 guarantee that $\mathcal{C} \downarrow_{\mathcal{M}} A$ is a category. The conditions 4 and 5 provide a pullback functor $h^* : \mathcal{C} \downarrow_{\mathcal{M}} B \to \mathcal{C} \downarrow_{\mathcal{M}} A$ for every morphism $h : A \to B$ in $\mathcal{C}$.

A concrete $\mathcal{M}$-span is a pair of $\mathcal{C}$-morphisms (p, q) such that $p \in \mathcal{M}$ and $\mathrm{domain}(p) = \mathrm{domain}(q)$. Two $\mathcal{M}$-spans (p_1, q_1) and (p_2, q_2) are equivalent and denote the same *abstract span* if there is an isomorphism i such that $p_1 \circ i = p_2$ and $q_1 \circ i = q_2$; in this case we write $(p_1, q_1) \equiv (p_2, q_2)$ and $[(p, q)]_{\equiv}$ for the class of spans that are equivalent to (p, q). The *category of abstract $\mathcal{M}$-spans* $\mathcal{M}(\mathcal{C})$ over $\mathcal{C}$ has the same objects as $\mathcal{C}$ and equivalence classes of spans wrt. $\equiv$ as arrows. The identities are defined by $\mathrm{id}_A^{\mathcal{M}(\mathcal{C})} = [(\mathrm{id}_A, \mathrm{id}_A)]_{\equiv}$ and composition of two spans $[(p, q)]_{\equiv}$ and $[(r, s)]_{\equiv}$ such that $\mathrm{codomain}(q) = \mathrm{codomain}(r)$ is given by $[(r, s)]_{\equiv} \circ_{\mathcal{M}(\mathcal{C})} [(p, q)]_{\equiv} = [(p \circ r', s \circ q')]_{\equiv}$ where (r', q') is a pullback of (q, r).

[1] The $\mathcal{M}$-morphisms are the objects of $\mathcal{C} \downarrow_{\mathcal{M}} A$ and, due to condition 3, each $\mathcal{C} \downarrow_{\mathcal{M}} A$-morphism is an $\mathcal{M}$-morphism.

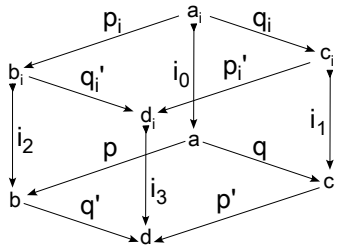

Fig. 1. Pushout/pullback cube

Note that there is the natural embedding faithful functor $\iota : \mathcal{C} \to \mathcal{M}(\mathcal{C})$ defined by identity on objects and $(f : A \to B) \mapsto [A \overset{\mathrm{id}_A}{\leftarrow} A \overset{f}{\to} B]_{\equiv}$ on morphisms. By a slight abuse of notation, we write $[A \overset{d}{\leftarrow} A' \overset{f}{\to} B]_{\equiv} \in \mathcal{C}$ if d is an isomorphism. From now on, we write $A \overset{f}{\leftarrow} B \overset{g}{\to} C$ for the abstract span $\left[A \overset{f}{\leftarrow} B \overset{g}{\to} C\right]_{\equiv}$.

$\mathcal{M}(\mathcal{C})$ is called a category of *partial morphisms* over $\mathcal{C}$, if $\mathcal{M}$ is a subclass of all monomorphisms of $\mathcal{C}$. The following property guarantees that pushouts in $\mathcal{C}$ are pushouts in a category of partial morphisms $\mathcal{M}(\mathcal{C})$.

Definition 1. ($\mathcal{M}$-Hereditary Pushout[2]) *A pushout (q', p') of (p, q) in $\mathcal{C}$ is* $\mathcal{M}$-hereditary *if for each commutative cube as in Figure 1, which has pullbacks (p_i, i_0) and (q_i, i_0) of (i_2, p) resp. (i_1, q) as back faces such that i_1 and i_2 are in $\mathcal{M}$, in the top square (q_i', p_i') is pushout of (p_i, q_i), if and only if in the front faces (p_i', i_1) and (q_i', i_2) are pullbacks of (i_3, p') resp. (i_3, q') and i_3 is in $\mathcal{M}$.*

The following fact reformulates the sufficient criterion of [14] for a category of partial morphisms to possess not only hereditary but all pushouts.

Fact 2. *(Pushout of Partial Morphisms) A category of partial morphisms $\mathcal{M}(\mathcal{C})$ has all pushouts, if* (i) *$\mathcal{C}$ has all pushouts and all small limits,* (ii) *pushouts in $\mathcal{C}$ are $\mathcal{M}$-hereditary,* (iii) *for every $h : A \to B$ in $\mathcal{C}$, the pullback functor $h^* : \mathcal{C} \downarrow_{\mathcal{M}} B \to \mathcal{C} \downarrow_{\mathcal{M}} A$ has a right adjoint $h_* : \mathcal{C} \downarrow_{\mathcal{M}} A \to \mathcal{C} \downarrow_{\mathcal{M}} B$.*

The theory of single-pushout transformation is built on a category $\mathcal{C}$ and a class $\mathcal{M}$ of monomorphisms such that $\mathcal{M}(\mathcal{C})$ has all pushouts. An example is the category $\mathcal{M}(\mathbb{G})$ of graphs where $\mathcal{M}$ is the class of all monomorphisms:

Definition 3. *(Category of Graphs $\mathbb{G}$) A* graph *$G = (V, E, src : E \to V, tgt : E \to V)$ consists of a set of* vertices *V, a set of* edges *E, and two mappings $src, tgt : E \to V$, which provide a* source *resp.* target *vertex for each edge. A* graph morphism *$f : G_1 \to G_2$ from a graph $G_1 = (V_1, E_1, src_1, tgt_1)$ to a graph $G_2 = (V_2, E_2, src_2, tgt_2)$ is a pair $(f_V : V_1 \to V_2, f_E : E_1 \to E_2)$ of mappings such that $f_V \circ src_1 = src_2 \circ f_E$ and $f_V \circ tgt_1 = tgt_2 \circ f_E$.*

[2] For details on hereditary pushouts see [9,11].

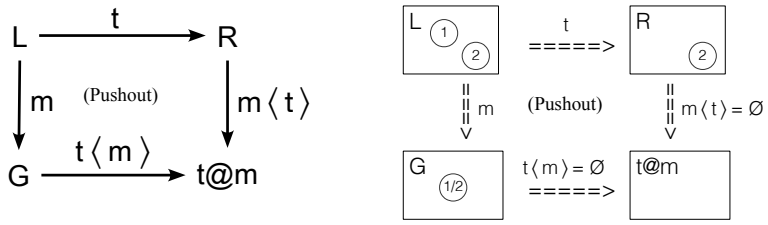

Fig. 2. Substitution and Partial Co-Match

Further examples are hyper-graphs or graph structures as in [13] where $\mathcal{M}$ is again the class of all monomorphisms.

Definition 4. *(Rule, Pre-Match, and Substitution) A* rule $t : L \to R$ *is a morphism in* $\mathcal{M}(\mathcal{C})$*. A* pre-match *for* t *in a* host graph G *is a morphism* $m : L \to G \in \mathcal{C}$*. A* substitution $t@m$ *of a rule* t *along a pre-match* m *is given by the* $\mathcal{M}(\mathcal{C})$*-pushout* $(t\langle m\rangle : G \to t@m, m\langle t\rangle : R \to t@m)$ *of* (t, m)*. The object* $t@m$ *is the* substitution result*. The partial morphism* $t\langle m\rangle$ *is called the* trace*, the partial morphism* $m\langle t\rangle$ *the* co-match*, compare left part of Figure 2.*

Note that the co-match need not be total, i. e. need not be in $\mathcal{C}$. An example in $\mathcal{M}(\mathbb{G})$ is depicted in the right part of Figure 2. It shows the substitution of a rule $t : L \to R$ at a pre-match $m : L \to G$. The left-hand side L of t consists of two vertices, namely ① and ②. The rule deletes ① and preserves ②. The pre-match maps both vertices in L to the same and only vertex in the host graph G.

Definition 5. *(Conflict- and Confusion-Free Pre-Match) A pre-match* m *for rule* t *is* conflict-free,[3] *if its co-match is in* $\mathcal{C}$*. It is* confusion-free*, if it is conflict-free for every prefix of* t*, i. e.* $m\langle p\rangle \in \mathcal{C}$ *for each* $p \in \mathcal{M}(\mathcal{C})$ *such that* $x \circ p = t$*.*

In [13], conflict- and confusion-freeness have been characterised for $\mathcal{M}(\mathbb{G})$.

Fact 6. *(Conflict and Confusion in* $\mathcal{M}(\mathbb{G})$*) A pre-match* m *for a rule* $t : L \to R$*, which is a span* $t = L \overset{t^l}{\leftarrow} D^t \overset{t^r}{\to} R$*, in* $\mathcal{M}(\mathbb{G})$ *is* (i) *conflict-free, if and only if*

$$\forall x, y \in L : m(x) = m(y) \implies x, y \in t^l(D^t) \vee x, y \notin t^l(D^t),$$

and it is (ii) *confusion-free, if and only if*

$$\forall x, y \in L : m(x) = m(y) \implies x, y \in t^l(D^t) \vee x = y.$$

Confusion-free pre-matches induce decompositions of substitutions for each rule decomposition:

[3] Single-pushout derivations at conflict-free matches coincide with sesqui-pushout rewritings [1] with monic left-hand sides in rules.

Fact 7. *(Substitution at Confusion-Free Pre-Match) If $t@m$ is a substitution of rule t at confusion-free pre-match m and $t = t_2 \circ_{\mathcal{M}(\mathcal{C})} t_1$ is an arbitrary decomposition of the rule, then $t\langle m\rangle = t_2\langle m\langle t_1\rangle\rangle \circ t_1\langle m\rangle$ and $m\langle t\rangle = m\langle t_1\rangle\langle t_2\rangle$.*

Fact 7 shows that every transformation at a confusion-free pre-match can be decomposed in elementary actions, namely (i) the addition of a single object (vertex or edge), the deletion of a single object, and the identification of two objects. Due to these positive properties, it is reasonable to allow only confusion-free pre-matches in direct derivations:

Definition 8. *(Match and Direct Derivation) The* matches *for a rule are its confusion-free pre-matches.* Direct derivations *are substitutions along matches.*

The compact notion of direct derivation allows for a straightforward and simple theory of single-pushout rewriting.[4] We repeat some results of [13] which are used below when we add inheritance.

Definition 9. *(Parallel Independence) Direct derivations $t_1@m_1$ and $t_2@m_2$ starting from the same host graph are parallel independent if $t_2\langle m_2\rangle \circ m_1$ is a match for t_1 and $t_1\langle m_1\rangle \circ m_2$ is a match for t_2.*

Derivations at independent matches lead to the same trace in any application order:

Fact 10. *(Parallel Independence) If direct derivations $t_1@m_1$ and $t_2@m_2$ are parallel independent, then $t_2\langle t_1\langle m_1\rangle \circ m_2\rangle \circ t_1\langle m_1\rangle = t_1\langle t_2\langle m_2\rangle \circ m_1\rangle \circ t_2\langle m_2\rangle$.*

Definition 11. *(Sub-rule and Remainder) A rule $t : L \to R$ is an (i,j)-*sub-rule *of another rule $t' : L' \to R'$, written $t \subseteq_{i,j} t'$, if $i : L \to L', j : R \to R'$ are two* total *morphisms (i. e. $i, j \in \mathcal{C}$) such that* (i) $j \circ t = t' \circ i$ *and* (ii) *i is a match*[5] *for t. The (i,j)-*remainder *of t' wrt. t is the universal morphism $t' -_{i,j} t : t@i \to R'$ that satisfies* (a) $(t' -_{i,j} t) \circ t\langle i\rangle = t'$ *and* (b) $(t' -_{i,j} t) \circ i\langle t\rangle = j$.

Fact 12. *(Composition of Matches) If $t : L \to R \subseteq_{i,j} t' : L' \to R'$ and $m : L' \to G$ is match for t', then $m \circ i$ is match for t.*

This fact together with property (a) of Definition 11 immediately provides:

Corollary 13. *(Sub-Rule) Direct derivations with sub-rule-structured rules can be decomposed into a derivation with the sub-rule followed by a derivation with the remainder, i. e. if $t : L \to R \subseteq_{i,j} t' : L' \to R'$ and $m : L' \to G$ is a match for t', then $t'\langle m\rangle = t' -_{i,j} t\langle m\langle t\langle i\rangle\rangle \circ t\langle m \circ i\rangle$ and $m\langle t'\rangle = m\langle t\langle i\rangle\rangle\langle t' -_{i,j} t\rangle$.*

Definition 14. *(Amalgamation) If $t_0 : L_0 \to R_0$ is a (i_1, j_1)-*sub-rule *of $t_1 : L_1 \to R_1$ as well as a (i_2, j_2)-*sub-rule *of $t_2 : L_2 \to R_2$, the amalgamation of t_1*

[4] All necessary proofs can be performed just by using well-known general composition and decomposition results for pushouts.

[5] Remember, that, due to Definition 8, all matches are confusion-free!

and t_2 along[6] *t_0 is the universal morphism $t_3 : L_3 \to R_3$ from the pushout $(i_1^* : L_2 \to L_3, i_2^* : L_1 \to L_3)$ of (i_1, i_2) to the pushout $(j_1^* : R_2 \to R_3, j_2^* : R_1 \to R_3)$ of (j_1, j_2) that satisfies $t_3 \circ i_1^* = j_1^* \circ t_2$ and $t_3 \circ i_2^* = j_2^* \circ t_1$.*

Fact 15. *(Induced Matches) The morphisms i_1^* and i_2^* constructed in Definition 14 are matches*[7] *for t_2 and t_1 resp.*

Lemma 16. *If m is match for rule t, then it is match for $j \circ t$, if $j \in \mathcal{C}$.*

The derivation with an amalgamated rule results in the same trace as applying the common sub-rule followed by the two remainders in any order.[8] This is an immediate consequence of the following fact:

Proposition 17. *(Amalgamation) Let $t_1 +_{t_0} t_2$ be the amalgamation of t_1 and t_2 along t_0, where t_0 is a (i_1, j_1)-subrule of t_1* and a *(i_2, j_2)-subrule of t_2. Let $i_0 = i_1^* \circ i_2 = i_2^* \circ i_1$, $m_1^R = i_2^* \langle t_0 \langle i_1 \rangle \rangle$, $t_1^R = (t_1 - t_0) \langle m_1^R \rangle$, $m_2^R = i_1^* \langle t_0 \langle i_2 \rangle \rangle$, and $t_2^R = (t_2 - t_0) \langle m_2^R \rangle$, then we obtain the following two properties:*

$$t_1 +_{t_0} t_2 = (t_2 - t_0) \langle t_1^R \circ m_2^R \rangle \circ (t_1 - t_0) \langle m_1^R \rangle \circ t_0 \langle i_0 \rangle$$
$$t_1 +_{t_0} t_2 = (t_1 - t_0) \langle t_2^R \circ m_1^R \rangle \circ (t_2 - t_0) \langle m_2^R \rangle \circ t_0 \langle i_0 \rangle .$$

Proof. Direct consequence of Facts 12 and 15, Lemma 16, and the observation that all quadrangles in Figure 3 are pushouts due to general pushout properties.

3 The Category of Typed Graphs with Inheritance

In this section, we recapitulate definitions and results from [16].

Definition 18. *(Type Graph) A* type graph *$T = (G_T, \leq)$ consists of a graph $G_T = (V, E, src, tgt)$ and a partial order $\leq \subseteq V \times V$, which has* least upper bounds *$\bigvee S$ and* greatest lower bounds *$\bigwedge S$ for every subset $S \subseteq V$.* □

The interpretation of *type graphs* from a software engineering perspective is the following: Vertices stand for *types* and edges model *associations* between types. The partial order $\leq$ on types represents the *inheritance* relation, i. e. $x \leq y$ means that x is a *sub-type* of y.

Note that the vertex set of a type graph cannot be empty, since *the least element* $\bigvee \emptyset$ and *the greatest element* $\bigwedge \emptyset$ must be vertices. Therefore, the simplest type graph consists of a single type vertex and no edges.

From a practical point of view, the existence of all greatest lower bounds and all least upper bounds seems to be a very strong and restrictive requirement. But it can easily be satisfied:

[6] More precisely, along (i_1, j_1) and (i_2, j_2).
[7] Confusion-free.
[8] I. e. the remainders are parallel independent.

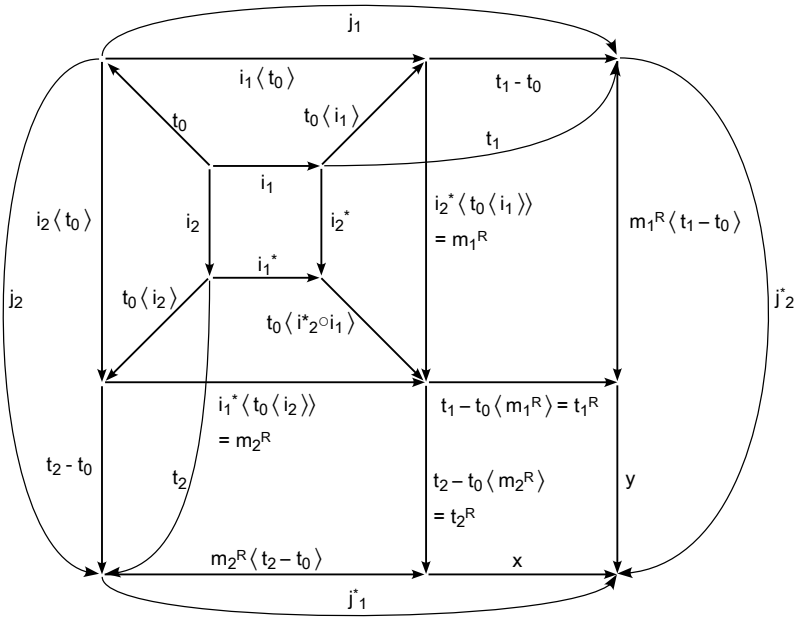

Fig. 3. Decomposition of Amalgamated Rule

For example any graph without inheritance relation can be turned into a type graph by adding the diagonal (reflexive vertex pairs), the greatest type $\bigwedge \emptyset$, an abstraction of all types, and the least type $\bigvee \emptyset$, a type that specialises all types, with the induced ordering. Any single-inheritance type hierarchy H can be turned into a type graph in our sense the same way: Add $\bigwedge \emptyset \mathrel{\widehat{=}} \texttt{Anything}$ if H has more than one root and add $\bigvee \emptyset \mathrel{\widehat{=}} \texttt{Everything}$ as a type for *objects of every shape*.

For an arbitrary type hierarchy H, there is the Dedekind/MacNeille-completion [17], which provides the smallest partial order closed under least upper and greatest lower bounds containing the original H. In the completion, any added element is a missing bound. In the finite case, the bounds in the completion coincide with the original bounds if they already existed in H.

If we are given an arbitrary type hierarchy H, which does not satisfy the type graph requirements of Definition 18, we calculate the type graph $T(H)$ by the Dedekind/MacNeille-completion. If, in any rewriting computation, added types, i. e. types in $T(H) - H$, occur, they can be interpreted as follows: `Everything` almost always indicates an error and all other added types indicate "uncertainty" in the sense that the concrete type in H cannot be computed on the basis of the given information.

Definition 19. *(Typed Graph) Given a type graph T, a graph G becomes a T-*typed graph *by a* typing *$i : G \to T$ which is a pair $(i_V : G_V \to T_V, i_E : G_E \to T_E)$ of mappings such that*[9]

$$i_V \circ src_G \leq src_T \circ i_E \tag{1}$$

$$i_V \circ tgt_G \leq tgt_T \circ i_E \tag{2}$$

The interpretation of *typed graphs* from a software engineering perspective is the following: Vertices of G stand for *objects*, which are assigned a type by the *instance-of* mapping i_V. Edges og G are *links* between objects. The type of a link is an association: The *instance-of* mapping i_E provides the corresponding assignment.

Conditions (1) and (2) specify that the source and target assignments of a link must be consistent with the source and target prescriptions of its type $i_E(l)$: The pair (i_V, i_E) is only required to be a homomorphism *up to inheritance*. Condition (1) means that sub-types inherit all associations of all their super-types. Condition (2) formalises the fact that associations may appear polymorphic at run-time in the type of their target.

Definition 20. *(Type-Compatible Morphism) If $i : G \to T$ and $j : H \to T$ are two typings into the same type graph T, a graph morphism $m : G \to H$ is* type-compatible*, written $m : i \to j$, if*

$$j_V \circ m_V \leq i_V \tag{3}$$

$$j_E \circ m_E = i_E \tag{4}$$

A morphism is called strong*, if $\leq$ in (3) can be replaced by $=$, i. e. $j_V \circ m_V = i_V$.*

*The typings in T together with the type-compatible graph morphisms between them constitute the category of T-*typed graphs *$\mathbb{G}^T$.*

There is a functor $\tau : \mathbb{G}^T \to \mathbb{G}$ which forgets the typing, i. e. maps a $\mathbb{G}^T$-morphism $m : (i : G \to T) \to (j : H \to T)$ to the $\mathbb{G}$-morphism $m : G \to H$.

A type-compatible morphism can map an object of type c to an object the type of which is a sub-type of c. *Strong* morphisms do not use this flexibility.

Fact 21. *(Strong Morphisms)* (a) *Isomorphisms are strong.* (b) *The composition of two strong morphisms is strong.* (c) *Strongness is prefix-closed, i. e. if $f \circ g$ is strong, then g is strong.*

Proposition 22. *(Limits and Co-Limits) For every small diagram $\delta : D \to \mathbb{G}^T$, there is a limit $(l_o : \mathbf{L} \to \delta(o))_{o \in D}$ and co-limit $(c_o : \delta(o) \to \mathbf{C})_{o \in D}$, such that $\tau(l_o)_{o \in D}$ and $\tau(c_o)_{o \in D}$ are the limit and co-limit of the diagram $\tau \circ \delta : D \to \mathbb{G}$ resp. The typings $l : \tau(\mathbf{L}) \to T$ and $c : \tau(\mathbf{C}) \to T$ map $x \in \tau(\mathbf{L})_V$ to $\bigvee\{\delta(o)(y) : y = l_o(x), o \in D\}$ and $x \in \tau(\mathbf{C})_V$ to $\bigwedge\{\delta(o)(y) : x = c_o(y), o \in D\}$ resp.*[10]

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

[10] The notation $o \in D$ stands here for $o \in Object_D$.

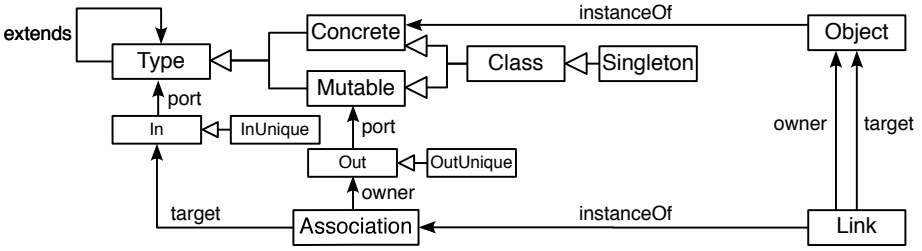

Fig. 4. Type Graph for Typed Object-Oriented Systems

Proof. Direct consequence of the results in [16] and the fact that all least upper bounds and greatest lower bounds exist in T.

To apply the single pushout transformation framework of Section 4, strong monomorphisms are of special interest:

Fact 23. *Pushouts and pullbacks preserve strong morphisms in $\mathbb{G}^T$, i. e. if (p^*, q^*) is pushout or pullback of (p, q) and p is strong, then p^* is strong.*

4 Single Pushout Transformation with Polymorphism

In this section, we instantiate the single-pushout framework of Section 2 for the category $\mathbb{G}^T$ of typed graphs presented in the last section. The span base category is $(\mathbb{G}^T, \mathcal{S})$ where $\mathcal{S}$ is the class of all strong monomorphisms. Thus, $\mathcal{S}(\mathbb{G}^T)$ is a category of (strong) partial morphisms. $(\mathbb{G}^T, \mathcal{S})$ satisfies the requirements (1) – (5) on page 356 of a span base, compare section 3.

Since $\mathbb{G}^T$ has all small limits and co-limits,[11] it remains to show the requirements (ii) and (iii) of Fact 2 for $\mathcal{S}(\mathbb{G}^T)$ to be a suitable category for single-pushout rewriting.[12]

Proposition 24. *(Pushout Conditions)* (a) *Pushouts in $\mathbb{G}^T$ are $\mathcal{S}$-hereditary.* (b) *Given a morphism $b : (h : H \to T) \to (k : K \to T)$ in $\mathbb{G}^T$, the pullback functor $b^* : \mathbb{G}^T \downarrow_{\mathcal{S}} k \to \mathbb{G}^T \downarrow_{\mathcal{S}} h$ has a right-adjoint.*

Theorem 25. *$\mathcal{S}(\mathbb{G}^T)$ has all pushouts.*

Proof. Direct consequence of Propositions 22 and 24 and Fact 2.

These results provide the fundament on which a concept of inheritance for single-pushout graph transformation can be built. We are already able to write *generic* rules. This is illustrated by the example depicted in Figures 4, 5, and 6.

[11] Compare Proposition 22.
[12] A detailed proof for the following proposition can be found in [15].

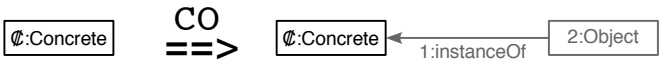

Fig. 5. Object Creation

Figure 4 depicts a type graph in the sense of Definition 18 for a small model for typed object-oriented systems. The partial order on vertices is generated by the given inheritance relations in UML notation.[13] There are only two types missing for the vertex order to possess all limits, namely $\bigvee \emptyset$ (`Everything`) and $\bigwedge \emptyset$ (`Anything`). We assume that these types are always implicitly added. The type graph specifies a type level and an instance level connected by the `instanceOf`-edges.

The most prominent type on the type level is `Type`. Types are orthogonally classified two times, namely in *concrete* versus not concrete (i. e. abstract) types on the one hand and in *mutable* versus immutable types on the other hand. `Type`-objects represent abstract *and* immutable types. `Concrete`-objects stand for *concrete* (i. e. not abstract) and immutable types, i. e. these objects can be target of `instanceOf`-edges. `Mutable`-objects model *mutable* and abstract types, i. e. these objects can be target of `port`-edges from `Out`-Objects. The type `Class` is derived from `Concrete` *and* `Mutable`. Therefore classes *inherit* the properties of *both* direct super-classes, i. e. objects of type `Class` (or more special) can be target of `instanceOf`-edges *and* can be owner of `Associations`. `Singleton`-types [4] are modelled as a specialisation of `Class`. The edges of type `extends` model specialisation. We assume that the set of these edges represents a hierarchy, i. e. includes edges for all paths (reflexive and transitive), and does not contain cyclic paths of length greater than zero (anti-symmetric). `Association`-objects connect `Out`- with `In`-ports. The specialisations of these port classes, i. e. `OutUnique` and `InUnique`, will be used later to model multiplicity specifications for associations.

The instance level is very simple. There are `Object`-objects which obtain a type (`Concrete`-object) on the type level by an `instanceOf`-edge. And there are `Link`-objects representing instances of `Association`-objects. `Link`-objects can connect `Object`-objects the `instanceOf`-target of which has type[14] `Class` (`owner`)[15] with `Object`-objects the `instanceOf`-target of which has type `Concrete` (`target`).[16]

Figure 5 depicts the method `CO` for the operation `createObject(¢:Concrete)`.[17] The method is generic because it can be applied to objects of all sub-types of `Concrete`, namely `Concrete`, `Class`, and `Singleton`.

[13] `http://www.uml.org/`

[14] A t'-object is of type t, if t' is equal to a direct or indirect sub-type of t.

[15] Note that the owner of a link must be concrete *and* mutable.

[16] The `owner`- and `target`-relations on the instance level must be consistent with the corresponding relations on the model level. We model this constraint by the link creation rule, compare Figure 6.

[17] We use the UML notation for object diagrams.

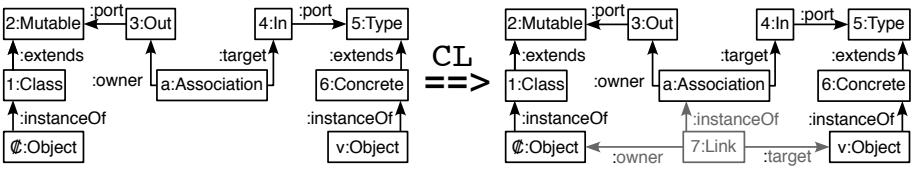

Fig. 6. Link Creation

The operation `createLink(¢ : Object, a : Association, v : Object)` is implemented by the method `CL` which is depicted in Figure 6. It allows the creation of a link only if the types of the receiver (`¢`) and the given value (`v`) are specialisations of the `owner`- and `target`-type of the given association parameter (`a`) resp. This method is also generic, since the object `1:Class` for example can be matched with `Class`- or `Singleton`-objects and there are 5 type choices for the object `5:Type`. Without specialisation, we would have to write 90 concrete rules for the type variations of the objects `1`, `2`, `5`, and `6`.

But *generic methods* are not the end of the game. Now, we introduce a mechanism that allows to extend rules. This is equivalent to method redefinition or polymorphism in object-oriented programming. A good example is the object creation rule `CO` in Figure 5. It does not always work right: The rule can create several instances for a `Singleton`-object.

Figure 7 shows the redefinition of `createObject(¢:Concrete)` in Figure 5 by a more special method $\texttt{CO}_\texttt{S}$, namely `createObject(¢:Singleton)`. The redefinition does not create an `Object`, if there is already one instance for the `Singleton`-class.[18] Note that `CO` has been made a sub-rule of $\texttt{CO}_\texttt{S}$ by the morphism pair (i, j).

Figure 7 also shows the application $\texttt{CO}@i$ which provides the remainder $\texttt{CO}_\texttt{S} - \texttt{CO}$, compare Definition 11. Note, that every direct derivation $\texttt{CO}_\texttt{S}@m$ with the redefinition coincides with the derivation sequence that applies the sub-rule `CO` at the match $m \circ i$ followed by the remainder application $(\texttt{CO}_\texttt{S} - \texttt{CO})\,@m\,\langle \texttt{CO}\,\langle \texttt{i} \rangle \rangle$, compare Corollary 13. Thus, the application of a sub-rule-structured rule corresponds to a `super`-call in object-oriented programming. We can think of the sub-rule as "shared code" that is always executed, in the example "adding an instance", and the remainder as the set of *additional* actions specified by the redefinition, in the example "identification of the new and the old instance".

Before we look at more complex examples, we formalise the presented feature of rule-extension and "application of the most specific rule".

Definition 26. *(Polymorphic Graph Transformation System) A* polymorphic graph transformation system $(T, P, \leq_P, M_P)$ *consists of a* type graph T*, a finite set of partial morphisms* $P \subseteq \mathbb{G}^T$*, representing the* rules*, a* partial rule order $\leq_P \subseteq P \times P$*, representing the specialisation relation on rules, and a family* M_P

[18] Note that the redefinition is the identity morphism.

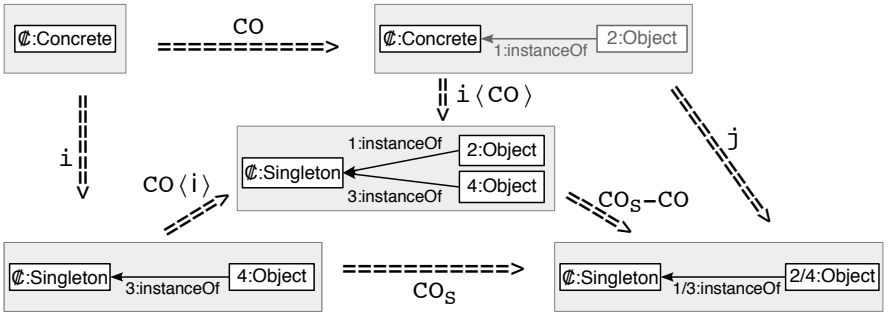

Fig. 7. Singleton Creation

of sub-rule specifications[19] $(l_{t,t'} : L \to L', r_{t,t'} : R \to R')$ *for every pair* $t' : L' \to R' \leq_P t : L \to R$ *satisfying:*

(a) *Every rule has a unique most general rule, i. e.*

$$t_3 \leq_P t_1 \wedge t_3 \leq_P t_2 \implies \exists t : t_1 \leq_P t \wedge t_2 \leq_P t \tag{5}$$

(b) *The sub-rule specifications are consistent with the sub-rule order, i. e.*

$$(l_{t,t}, r_{t,t}) = (\mathrm{id}_L, \mathrm{id}_R) \text{ for each } t : L \to R \in P \tag{6}$$

$$(l_{t,t''}, r_{t,t''}) = (l_{t',t''} \circ l_{t,t'}, r_{t',t''} \circ r_{t,t'}) \text{ for each triple } t'' \leq_P t' \leq_P t \tag{7}$$

Note that t' is a sub-rule of t, if t is a specialisation of t', since specialisation on rules means extension.[20] The analogy to Condition (a) in object-orientated programming is the fact that each method implements a unique operation.

From object orientation, we inherit the idea that the most specific method has to be chosen, if several methods for the same operation are applicable. This concept translates to our approach as follows:

Definition 27. *(Match and Derivation in Polymorphic System) A match* m *for a rule* t *in a* system $PGT = (T, P, \leq_P, M_P)$ *is a* most specific match*, if for all rules* $t', \widehat{t} \in P$ *with* $t' \leq_P \widehat{t}$*,* $t \leq_P \widehat{t}$ *and all matches* m' *for* t' *we have:* $m' \circ l_{\widehat{t},t'} = m \circ l_{\widehat{t},t} \implies t \leq_P t' \wedge m \circ l_{t't} = m'$*. A* direct derivation in PGT *is a direct derivation at a most specific match.*[21]

The specialisation `OutUnique` of `Out`-ports stands for the multiplicity `0...1`. Figure 8 depicts an extension $\mathtt{CL}_0$ of the rule `CL` in Figure 6 which guarantees the

[19] Compare Definition 11.

[20] Thus, the partial orders on types and rules are consistent: "less than" means "more special".

[21] Note that rule redefinition formulates a negative application condition in the sense of [8]: A rule is *not* applicable if a more special rule is.

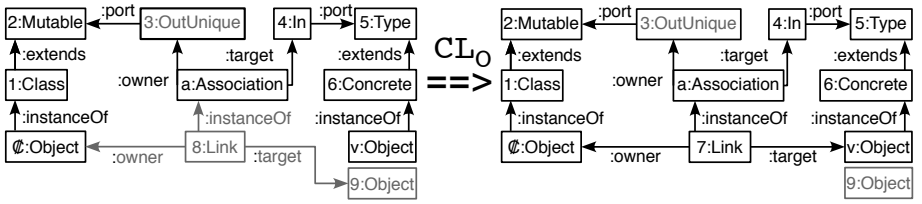

Fig. 8. Extension of Link Creation

corresponding uniqueness property for outgoing edges. The rule CL is redrawn in Figure 8 in black, the additional parts of $\mathtt{CL_0} - \mathtt{CL}$ are drawn in grey: $\mathtt{CL_0}$ removes an existing link, when a new link is added.

Symmetrically, we can extend CL by a rule $\mathtt{CL_I}$ that removes an existing In-link of an object *o*, when a new In-link of the same association with a InUnique-port is added to *o*. Now, we have the situation $\mathtt{CL_0} \leq \mathtt{CL} \geq \mathtt{CL_I}$ which makes it possible that there is no most specific match, namely if a link shall be added the Association of which is unique at the In- and the Out-end between two objects that both possess such a link on the owner resp. target side. We need a special rule for this situation. It can be automatically generated, namely by the computation of the amalgamated rule $\mathtt{CL_0} +_{\mathtt{CL}} \mathtt{CL_I}$, compare Definition 14. Due to Proposition 17, it provides the correct effect. Thus, the automatic addition of all amalgamated rule extensions can help to solve the "method selection problem" in multiple redefinition situations.

The example demonstrates a maximum of code sharing: $\mathtt{CL_0}$ and $\mathtt{CL_I}$ reuse CL and $\mathtt{CL_0} +_{\mathtt{CL}} \mathtt{CL_I}$ reuses $\mathtt{CL_0}$ and $\mathtt{CL_I}$ and, indirectly, also CL. Every extension adds behaviour and *does not change* behaviour of the more general rules. These properties are guaranteed by Corollary 13 and Proposition 17. Therefore, we obtain a predictable system behaviour although we admit specialisation of rules.

5 Related Work

Most related theoretical research lines do not admit polymorphism. H. Ehrig et al. [2] introduce inheritance as an additional set of inheritance edges between vertices in the type graph. It is not required that this structure is 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 [7] extend this approach to inheritance between vertices and edges.

F. Hermann et al. [10] avoid this flattening and define a weak adhesive category based on the original type graph with inheritance structure. The rule morphisms are required to reflect the sub-type structure: If an image of a morphism possesses

sub-types, all these sub-types have pre-images under the morphism. This feature considerably restricts the applicability of the approach in situations like those in Section 4.

U. Golas et al. [6] also avoid flattening. They require that the paths along inheritance edges are cycle-free (hierarchy) and that every vertex has at most one abstraction. For this set-up, they devise an adhesive category comparable to our approach in [16] but restricted to single-inheritance.

The above mentioned concepts do not address redefinition of rules and "code sharing" by using rule specialisation and polymorphism. One approach in this direction is the model of object-oriented programming by A. P. Lüdtke Ferreira and L. Ribeiro [5], which is 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 to be adequate as a model for object-oriented systems. They do not address further categorical properties. The work in [5] aims at modelling object-oriented concepts like inheritance and polymorphism by single-pushout graph grammars. It does not equip general single-pushout rewriting with polymorphism.

There are some practical approaches that allow rule extension. One example is [12] which is based on triple graph grammars. The operational effects are comparable to ours, but the devised mechanisms are described informally only.

6 Conclusions and Future Research

In this paper, we extended single-pushout graph transformation by inheritance and polymorphism. The introduced polymorphism is controlled, since it allows to add behaviour by rule extension but forbids changes of behaviour. This extension mechanism qualifies graph transformation as a modelling technique for extendable frameworks. Since non-monic rules are possible, effects of negative application conditions [8] can be modelled, compare for example Figure 7.

There are two directions for future theoretical research. After having handled inheritance for the double- and the single-pushout approach in [16] resp. in this paper, the concepts have to be generalised to the sesqui-pushout approach [1]. And theoretical results of the algebraic approach, e. g. the critical pair analysis, have to be generalised to polymorphic systems.

From the practical point of view, future research has to investigate the gained increase in expressiveness. Besides addition, deletion and identification of objects, the application of a single-pushout rule $L \leftarrow_l D \rightarrow_r R$ with inheritance at match m can also specialise the types of objects, namely for those $x \in D$ for which (i) r is not strong, i. e. $R(r(x)) \leqq D(x)$, or (ii) which are identified by r with an item $y \neq x$ such that the types of $m(l(x))$ and $m(l(y))$ are different.

Another interesting practical issue is the invention of a methodology for the development of *message-based* object-oriented systems, for example in the sense of [5], starting from arbitrary polymorphic graph transformation systems.

References

1. Corradini, A., Heindel, T., Hermann, F., König, B.: Sesqui-pushout rewriting. In: Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozenberg, G. (eds.) ICGT 2006. LNCS, vol. 4178, pp. 30–45. Springer, Heidelberg (2006)
2. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Springer (2006)
3. Ehrig, H., Pfender, M., Schneider, H.J.: Graph-grammars: An algebraic approach. In: FOCS, pp. 167–180. IEEE (1973)
4. Gamma, E., et al.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley (1994)
5. 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)
6. Golas, U., Lambers, L., Ehrig, H., Orejas, F.: Attributed graph transformation with inheritance: Efficient conflict detection and local confluence analysis using abstract critical pairs. Theor. Comput. Sci. 424, 46–68 (2012)
7. Guerra, E., de Lara, J.: Attributed typed triple graph transformation with inheritance in the double pushout approach. Technical Report UC3M-TR-CS-06-01. Technical Report Universidad Carlos III de Madrid (2006)
8. Habel, A., Heckel, R., Taentzer, G.: Graph grammars with negative application conditions. Fundam. Inform. 26(3/4), 287–313 (1996)
9. Heindel, T.: Hereditary pushouts reconsidered. In: Ehrig, H., Rensink, A., Rozenberg, G., Schürr, A. (eds.) ICGT 2010. LNCS, vol. 6372, pp. 250–265. Springer, Heidelberg (2010)
10. 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)
11. Kennaway, R.: Graph rewriting in some categories of partial morphisms. In: Ehrig, H., Kreowski, H.-J., Rozenberg, G. (eds.) Graph Grammars 1990. LNCS, vol. 532, pp. 490–504. Springer, Heidelberg (1991)
12. Klar, F., Königs, A., Schürr, A.: Model transformation in the large. In: Crnkovic, I., Bertolino, A. (eds.) ESEC/SIGSOFT FSE, pp. 285–294. ACM (2007)
13. Löwe, M.: Algebraic approach to single-pushout graph transformation. Theor. Comput. Sci. 109(1&2), 181–224 (1993)
14. Löwe, M.: A unifying framework for algebraic graph transformation. Technical Report 2012/03, FHDW-Hannover (2012)
15. Löwe, M., König, H., Schulz, C.: Polymorphic single-pushout graph transformation. Technical Report 2013/04, FHDW-Hannover (2013)
16. Löwe, M., König, H., Schulz, C., Schultchen, M.: Algebraic graph transformations with inheritance. In: Iyoda, J., de Moura, L. (eds.) SBMF 2013. LNCS, vol. 8195, pp. 211–226. Springer, Heidelberg (2013)
17. MacNeille, H.M.: Partially ordered sets. Trans. Amer. Math. Soc. 42(3), 416–460 (1937)