Abstract
In this paper, we propose a new approach to inheritance and abstraction 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 $t$. Abstract classes are modelled by a subset of
vertices of the type graph. 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, Abstraction, Adhesive HLR category

1. Introduction and Related Work
Modelling Object-oriented Systems

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 [1], a language which is currently the de facto
standard for modelling object-oriented systems.

If one looks more closely at object-oriented systems, which consist of a type level with
classes, associations, etc. accessible at design time and an instance level with objects,
links, etc. at run-time, one realises that it is impossible to analyse or build object-oriented software in an efficient way without making use of specialization or \textit{inheritance} \footnote{In this paper, we do not differentiate between interface inheritance (specialization or subtyping) and implementation inheritance (class inheritance of subclassing), 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; even in Java, subclassing always implies subtyping.}. Inheritance allows us to factor out common data and behaviour into separate classes which can be documented and tested separately. By differentiating between classes providing an interface (only) and classes implementing that interface, a separation between interface and implementation can be made explicit. Moreover, it allows to extend existing behaviour in a non-intruding way, which is sometimes called the “Open-Closed Principle”. When inheritance is applied top-down, it enables the use of subtype polymorphism and makes parallel development against common interfaces possible. When applied bottom-up, it permits refactoring and the cleanup of interfaces and code. All in all, inheritance is one of the most important features of the object-oriented paradigm \cite{2, 3}.

In addition, the notion of \textit{abstraction} allows to define abstract data types (ADT), which have to be specialised for concrete use cases. It is a natural consequence of the inheritance feature to be able to make the difference between abstract and concrete classes. Abstract classes are “unfinished” and need to be completed by subclasses. So no instances of abstract classes are desirable as such objects would not be able to understand and react to the complete set of messages their interface allows. In contrast, concrete classes guarantee that their instances are fully functional. In an object-oriented system, all objects are instances of concrete classes. When transforming such systems (see below), it is highly desirable that this property be preserved.

It should also be stressed that (interface) inheritance typically requires \textit{multiple inheritance}, i.e., that a type not only inherits from a single type but from a whole set of other types. This enables designers to formulate the key idea that a class implements behaviour specified by many types (interface inheritance) or extends the behaviour and/or state implemented by many classes (implementation inheritance). If multiple inheritance were not supported, the designer would be forced to create fat interfaces\footnote{A fat interface is an interface with many operations.} or to introduce complicated delegation patterns, which considerably defeats the goal of building an object-oriented system from small and coherent parts.

It follows that it is sensible to require that the graphical notation directly support aspects of inheritance and abstraction, without being forced to transform them to lower-level constructs. Keeping the models at a high level of abstraction helps in understanding what they are about and how they change. Lowering the abstraction level of the model necessarily leads to constructs that are more difficult to read and understand. Additionally, if such a lower-level model is transformed (see below), the resulting model somehow has to be translated back to a high-level representation, which may be difficult or even impossible in some cases\footnote{The second case may happen if the transformation does not preserve some of the properties required by the higher-level models.}.

\textit{Graph Transformations}

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). Typically, such a transformation is modelled either
by a graph transformation rule $L \xleftarrow{l} K \xrightarrow{r} R$ with source graph $L$, target graph $R$, and
total graph morphisms $l$ and $r$, as it is done in the double-pushout approach (DPO, [4]),
or as a single partial graph morphism $L \xrightarrow{l} R$, as it is done in the single-pushout approach
(SPO, [5]). In Fig. 1 you can see an example of the double-pushout approach where the
graph transformation rule is defined at the top and the graph to be transformed is found
at the bottom; a matching relation then defines which parts of the graph the rule shall
apply to. Both squares are so-called pushout squares. In the rule, $l$ specifies which graph
elements are to be deleted (namely those which are not in the image of $l$), whereas $r$
determines which graph elements are to be added (namely those which are not in the
image of $r$). When $G_I$ gets transformed along $l$ and $r$, the effects of $l$ and $r$ on $L_R$ are
mirrored on $G_I$, namely on the part of $G_I$ which is reached by $m$. So $f$ and $g$ do the
same on $m(L_R)$ in the context of $G_I$ as $l$ and $r$ do on $L_R$.

$$
\begin{array}{c}
L_R \\
\downarrow m \\
G_I
\end{array}
\xleftarrow{f}
\begin{array}{c}
K_R \\
\downarrow k \\
D_I
\end{array}
\xrightarrow{g}
\begin{array}{c}
R_R \\
\downarrow n \\
H_I
\end{array}
$$

Figure 1: Double-pushout transformation according to [4]

In this paper, we concentrate on the double-pushout approach as depicted in Fig. 1. To
be able to apply the double-pushout approach to a system that does not consist of standard
graphs and standard graph homomorphisms only, as e.g. typed graphs, we need to ensure
that the category we define is an adhesive HLR category. Adhesive high-level replacement
(HLR) categories as introduced in [6, 4] combine high-level replacement systems [7] with
the notion of adhesive categories [8] in order to be able to generalise the double-pushout
transformation approach from graphs to other high-level structures, as e.g. attributed
graphs [9] and Petri nets using a categorial framework. Generally, adhesiveness abstracts
from exactness properties like compatibility of union and intersection of sets. Due to
special properties that guarantee the preservation of pushouts and pullbacks in certain
situations, many useful results of the theory of graph transformation can be obtained,
for example the Local Church Rosser Theorem for pairwise analysis of sequential and
parallel independence [4, Thm. 5.12], the Parallelism Theorem for applying independent
rules and transformations in parallel [4, Thm. 5.18], or the Concurrency Theorem for
applying $E$-related dependent rules simultaneously [4, Thm. 5.23]. Algebraic graph
transformations based on adhesive HLR categories have been studied for a long time and
are a well-known tool in the context of software engineering [4].

Related Work

In the classical graph transformation approach, graphs are untyped. A common
extension is the transformation of typed graphs [4]. The type graph, however, does not
possess any inheritance structure. It follows that it is advisable to combine high-level
object-oriented models directly supporting the notions of inheritance and abstraction with high-level graph transformations that guarantee preservation of such notions and which allow to obtain useful results from the theory of graph transformation. However, there are relatively few approaches dealing with inheritance at all, and even less approaches that allow the use of multiple inheritance.

H. Ehrig et al. [4] introduce inheritance as an additional set of inheritance edges between vertices in the type graph and abstract nodes as a subset of the vertices. Interestingly, 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, which is called “flattening the type graph”. By this reduction, they get rid of inheritance and are able to reestablish their theoretical results because the resulting category is an adhesive HLR category. E. Guerra and J. de Lara [10] extend this approach to inheritance between vertices and edges, J. de Lara et al. [11] extend it by productions operating on abstract nodes. However, all of these approaches do not fit our needs because the actual inheritance structure (i.e. its generating relation) is not preserved during graph transformation and has to be recreated somehow later.

F. Hermann et al. [12] avoid the flattening by establishing a different framework for dealing with inheritance structures. They define a category (called I-Graphs) the objects of which admit internal inheritance structures and the morphisms thus enable introduction or deletion of inheritance relations. They show that this category together with the class of injective and inheritance-reflecting morphisms is a weak adhesive HLR category. The main difference to our approach is that they operate on type graphs, not on instance graphs, so they have to deal with inheritance structures in the graphs to be transformed. In contrast, our approach transforms instance graphs without any inheritance structure; the inheritance relation occurs in the type graph only. The advantage of being able to condense many graph transformation rules to one abstract rule, however, does not vanish in our approach, because matches are allowed to be “down typing”.

The approach of U. Golas et al. [13] is similar to ours. On the one hand, it is more general since it includes NACs (negative application conditions) and attributed graphs. On the other hand, in contrast to our approach, it requires that every vertex has at most one supertype (single inheritance). This requirement admits certain “minimal contexts” which enable the definition of so-called “abstract critical pairs”. With multiple inheritance present, however, these minimal contexts seem to vanish.

A. P. L. Ferreira and L. Ribeiro [14] introduce a graph transformation framework for object-oriented programming based on single-pushout rewriting. They allow vertex and edge specializations 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. However, they do not address further categorial properties like adhesiveness. Also, their approach does not support multiple inheritance.

Outline of this Paper

In this paper we propose a new approach, which deals with the transformation of instance graphs that are typed in a type graph with an inheritance structure. 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. Multiple inheritance is explicitly allowed. 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. Additionally, we investigate abstraction modelled by a subset of abstract vertices of the type graph.

We address neither attributed graphs nor edge inheritance in this paper. The combination of our results with one or both of these extensions is left to future research.

The basic results were already presented in [15] for typed graphs. However, there was no integration of abstract classes into the framework. The main goal of this paper is to show that the very general results from [15] still hold if abstraction is added. Moreover, in this paper, we present more detailed case studies in order to show the benefit of the invented ideas.

The paper is structured as follows. Section 2 presents a practical example that demonstrates the goals of our approach. Some basic notions needed for understanding the mathematical theory developed in the later sections are introduced in section 3. Section 4 develops the basic concepts of our approach and defines the category $G^T$ which is used in subsequent sections. Sections 5 and 6 analyse the properties of monomorphisms, pushouts, and pullbacks in $G^T$. In section 8, we prove the main result of this paper, namely that $G^T$ is an adhesive HLR category. Section 9 demonstrates the usefulness of our approach by applying it to the example from section 2. Section 10 examines the integration of abstraction to our category. Finally, section 11 summarises the results and discusses future work.

2. Motivating Example

Consider a simple model of a file system (Fig. 2). In the sequel, type graphs (cf. Def. 1) are depicted using UML class diagrams where nodes represent the available classes and edges represent directed associations or attributes. Inheritance is represented by arrows with closed unfilled tips. Object graphs will be depicted using UML object diagrams where nodes represent properly typed objects and edges represent links.

On the one hand, we have the file system itself and subdirectories, which both can contain other file system objects and, thus, are called containers, which, in the sequel, will be abbreviated “C’er”. On the other hand, we have subdirectories and files, which are part of a container and, thus, are called containees (abbreviation: “C’ee”). Subdirectories can be created by the graph transformation rules in Fig. 3a and 3b (file creation is done by similar rules). The rules in Fig. 3c and 3d allow to delete a file by unlinking it from its container (deletion of subdirectories is done by similar rules).

One can immediately see that it would be very convenient to be able to specify one rule for the creation of files and subdirectories in any container, as well as one rule for

---

4 We call this property “down-typing”.
5 Note that we do not differentiate between associations and attributes as we do not differentiate between primitive types and class types.
6 Some sort of garbage collector is needed to physically delete all objects that are not part of any container. These rules are not shown in this example.
the deletion of files and directories from any container. This can obviously be achieved if the inheritance structure is exploited in such a way that concrete nodes in the graph to be transformed can be matched by abstract nodes in the rule’s left side.

Now we extend the file system model by links (see Fig. 4). Creating a link within a subdirectory that points to a file is handled by the rule in Fig. 5. Given a link pointing to a file, the rule in Fig. 6 allows to retarget it to a subdirectory.

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 would only need one rule to create the containee, instead of one rule for each concrete container. Only one rule would be needed to delete any file system object, i.e., no new rule must be added for the deletion of links. Retargeting a link could be specified by one single rule (independent

---

7 This cuts down the number of necessary rules for creating and deleting files and subdirectories from eight rules to three rules. Of course, we also need a rule for creating a file system, but this is not relevant for this discussion.

8 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.
of whether the old and new targets of the link are directories, files, or links), whereas without any abstraction, nine rules are necessary.

3. Preliminaries

In this section, we introduce some basic notions from the theory of graph transformation and from category theory. The inclined reader is referred to [4] and [16] for more details.

**Introduction to Category Theory**

A category $\mathcal{C} = (\text{Obj}^\mathcal{C}, \text{Mor}^\mathcal{C}, \circ^\mathcal{C}, id^\mathcal{C})$ consists of a class of objects $\text{Obj}^\mathcal{C}$, a class of morphisms $\text{Mor}^\mathcal{C} = (\text{Mor}^\mathcal{C}_{X,Y})_{X,Y \in \text{Obj}^\mathcal{C}}$, a class of compositions $\circ^\mathcal{C} = (\circ^\mathcal{C}_{X,Y,Z} : \text{Mor}^\mathcal{C}_{Y,Z} \times \text{Mor}^\mathcal{C}_{X,Y} \rightarrow \text{Mor}^\mathcal{C}_{X,Z})_{X,Y,Z \in \text{Obj}^\mathcal{C}}$ and a class of identities $id^\mathcal{C} = (id^\mathcal{C}_X \in \text{Mor}^\mathcal{C}_{X,X})_{X \in \text{Obj}^\mathcal{C}}$. 

---

**Figure 4: File system model with links**

**Figure 5: Rule “Create Link to a File in a Subdirectory”**

**Figure 6: Rule “Retarget Link from a File to a Subdirectory”**
where the composition is associative and the identities obey the identity rules wrt. composition, i.e., we have
\[ h \circ^C (g \circ^C f) = (h \circ^C g) \circ^C f \]
and
\[ id^C_X \circ^C f = f \]
\[ f \circ^C id^C_W = f \]
for all \( W, X, Y, Z \in \text{Ob}^C \), \( f \in \text{Mor}^C_{W,X} \), \( g \in \text{Mor}^C_{X,Y} \), and \( h \in \text{Mor}^C_{Y,Z} \). In the sequel, we will simply write \( \circ \) instead of \( \circ^C \). The notations \( f: X \rightarrow Y \) and \( f \in \text{Mor}^C_{X,Y} \) are equivalent if the category in question can be deduced from the context.

A morphism \( f: X \rightarrow Y \) for some \( X, Y \in \text{Ob}^C \) is said to be a monomorphism iff it is left-cancellative, i.e., \( g \circ f = h \circ f \implies g = h \) for any two compatible morphisms \( g, h \in \text{Mor}^C_{W,X} \) and some object \( W \in \text{Ob}^C \). We sometimes say that \( f \) is monic iff it is a monomorphism.

A morphism \( f: X \rightarrow Y \) for some \( X, Y \in \text{Ob}^C \) is said to be an epimorphism iff it is right-cancellative, i.e., \( f \circ g = f \circ h \implies g = h \) for any two compatible morphisms \( g, h \in \text{Mor}^C_{Y,Z} \) and some object \( Z \in \text{Ob}^C \). We sometimes say that \( f \) is epic iff it is an epimorphism.

A morphism \( f: X \rightarrow Y \) for some \( X, Y \in \text{Ob}^C \) is said to be an isomorphism iff there exists another morphism \( f^{-1}: Y \rightarrow X \) such that \( f^{-1} \circ f = id^C_X \) and \( f \circ f^{-1} = id^C_Y \). Note that all identities are isomorphisms and all isomorphisms are monic and epic.

A commutative square as on the left side of Fig. 7 is called a pullback iff for each \( P' \) and \( p: P' \rightarrow X \), \( q: P' \rightarrow Y \) with \( f \circ p = g \circ q \) there is a unique \( u: P' \rightarrow P \) with \( g' \circ u = p \) and \( f' \circ u = q \). Similarly, a commutative square as on the right side of Fig. 7 is called a pushout iff for each \( P' \) and \( p: X \rightarrow P' \), \( q: Y \rightarrow P' \) with \( p \circ f = q \circ g \) there is a unique \( u: P \rightarrow P' \) with \( u \circ g' = p \) and \( u \circ f' = q \).

![Figure 7: Pullback and Pushout Diagrams](image)

Sometimes \( f \) and \( g' \) are given and one looks for \( Z \xrightarrow{g} Y \xrightarrow{f} P \) such that the resulting square becomes a pushout. If such an object \( Y \) exists, it is called pushout complement (of \( f \) and \( g' \)).

A functor \( F: C \rightarrow D \) between two categories \( C \) and \( D \) defines the mappings
\[
F_{\text{Ob}}: \text{Ob}_C \rightarrow \text{Ob}_D
\]
\[
(F_{\text{Mor}_{X,Y}}: \text{Mor}^C_{X,Y} \rightarrow \text{Mor}^D_{F_{\text{Ob}}(X),F_{\text{Ob}}(Y)})_{X,Y \in \text{Ob}^C}
\]
with
\[
F_{\text{Mor}_X,Y}(id_X) = id_{F_{\text{Ob}}(X)}
\]
\[
F_{\text{Mor}_X,Z}(g \circ f) = F_{\text{Mor}_Y,Z}(g) \circ F_{\text{Mor}_X,Y}(f)
\]
for all \(X, Y, Z \in \text{Ob}^C\), \(f \in \text{Mor}^C_{X,Y}\), and \(g \in \text{Mor}^C_{Y,Z}\).

Introduction to Graphs

A graph \(G = (V_G, E_G, s_G : E_G \to V_G, t_G : E_G \to V_G)\) consists of a set of vertices or nodes \(V_G\), edges \(E_G\), and two total mappings \(s_G, t_G : E_G \to V_G\) which assign to each edge a source or target node, respectively.\(^9\)

Morphisms \(f : G_1 \to G_2\) are pairs of mappings compatible with the graph structure, i.e. they obey the rules \(f \circ s_{G_1} = s_{G_2} \circ f\) and \(f \circ t_{G_1} = t_{G_2} \circ f\).\(^10\)

\(G\) denotes the category of graphs and graph morphisms as defined above, with the usual composition of graph morphisms and identity graph morphisms as pairs of identity mappings.

Pushouts in \(G\) can be interpreted as gluing both graphs \(X\) and \(Y\) along a common subgraph \(Z\) (as long as the morphisms \(f\) and \(g\) are injective). Pullbacks in \(G\) can be interpreted as the construction of a preimage of \(Y\) under \(f\), if \(g\) is injective and “cuts out” a subgraph from \(Z\). However, the pullback and pushout constructions are more general as they do not require these morphisms to be injective.

Introduction to van Kampen Squares and Adhesive HLR Categories

Adhesive HLR categories provide a set of properties that are helpful for building a graph transformation framework. Generally, adhesiveness requires pushouts (unions) and pullbacks (intersections) to be “compatible” in a certain way:

A category \(\mathcal{C}\) with a morphism class \(\mathcal{M}\) is called an adhesive high-level replacement category, or adhesive HLR category in short, if:

1. \(\mathcal{M}\) is a class of monomorphisms closed under isomorphisms\(^11\), composition\(^12\), and decomposition\(^13\).
2. \(\mathcal{C}\) has pushouts and pullbacks along \(\mathcal{M}\)-morphisms, and \(\mathcal{M}\)-morphisms are preserved by pushouts and pullbacks\(^14\).
3. Pushouts in \(\mathcal{C}\) along \(\mathcal{M}\)-morphisms are van Kampen squares (see below).

A pushout as shown in the left part of Fig. \(\text{Fig. 8}\) is a van Kampen square if, for any commutative cube with the pushout at the bottom and where the back faces are pullbacks, the following statement holds: The top face is a pushout iff the front faces are pullbacks (right part of Fig. \(\text{Fig. 8}\)).

---

\(^9\) These notations will remain fixed in that for any graph \(X\) we will always write \(V_X, E_X, s_X, t_X\) for the constituents of \(X\) without defining them explicitly.

\(^10\) 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.

\(^11\) Let \(f : A \to B \in \mathcal{M}\) be given, then we have \(g \circ f \in \mathcal{M}\) and \(f \circ h \in \mathcal{M}\) for all \(\mathcal{C}\)-isomorphisms \(g : B \to C\) and \(h : D \to A\).

\(^12\) \(f : A \to B \in \mathcal{M} \land g : B \to C \in \mathcal{M} \Rightarrow g \circ f \in \mathcal{M}\)

\(^13\) \(g \circ f \in \mathcal{M} \land g \in \mathcal{M} \Rightarrow f \in \mathcal{M}\)

\(^14\) This means that \(f \in \mathcal{M} \Rightarrow f' \in \mathcal{M}\), cf. Fig. \(\text{Fig. 7}\).
4. Basic Definitions

We formalise class inheritance and abstraction by an additional partial order on the nodes of a type graph:

**Definition 1 (Type Graph).** A type graph is a tuple \((T, A, \leq)\) where \(T\) is a graph with vertex set \(V_T\), \(A \subseteq V_T\) denotes the abstract vertices, and \(\leq \subseteq V_T \times V_T\) is a partial order with a largest element \(O \in V_T\)\(^{15}\). The vertices in \(V_T \setminus A\) are called concrete.

This definition reflects the basic nature of class models, cf. Fig. 2 where \(A\) is represented by abstract classes and \(\leq\) is depicted by inheritance arrows. It still lacks additional annotations like multiplicities 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.

Now we define when an object structure (represented by some graph \(I\)) is properly typed in a type graph \(T\):

**Definition 2 (Typed Graph).** Let \(I \in \mathcal{G}\) and \((T, A, \leq)\) 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\(^{16}\)

\[
\text{(1)} \quad i \circ s_I \leq s_T \circ i \\
\text{(2)} \quad i \circ t_I \leq t_T \circ i
\]

A \(T\)-typed graph \(i: I \to T\) is called concrete if for all \(v \in V_I\), \(i(v)\) is concrete. Otherwise, \(i\) is called abstract.

Condition (1) means that subtypes inherit all associations of all their supertypes. Condition (2) formalises the fact that referenced objects at runtime may appear polymorphically: They may be of any subtype of the corresponding association target, cf. Fig. 9

\[^{15}\text{The letter “O” shall remind of the class “Object” in Java, which is a superclass of all other classes, hence the inheritance order’s largest object.}\]

\[^{16}\text{If } f, g: X \to Y \text{ are two mappings into a partially ordered set } Y = (Y, \leq), \text{ we write } f \leq g \text{ if } f(x) \leq g(x) \text{ for all } x \in X.\]
Note that typed graphs coincide with the definition of “clan morphism” if the underlying relation \( I \) in [12] is a partial order. Note also that in object-oriented systems, a system state (existing objects at runtime) is always a concrete typed graph.

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

Now we look at morphisms between typed graphs. If we recall the motivating example, we want to formulate graph transformation rules at an abstract level, i.e., where an object of a more general type \( c \) can match a specialized object of a type that is a subtype of \( c \). So we do not want morphisms between typed graphs to require type equality. Rather, we want something like type compatibility “up to inheritance”.

**Definition 3 (Type-Compatible and Strong Morphisms).**

Given two typed graphs \( i: I \to T \) and \( j: J \to T \), a graph morphism \( m: I \to J \) is **type-compatible**, written \( m: i \to j \), if the (in)equations

\[
\begin{align*}
 jV \circ mV & \leq iV \\
 jE \circ mE & = iE
\end{align*}
\]

hold.

If in (3) “\( \leq \)” can be replaced by “=”, \( m \) is called **strong** or **type-preserving**. A strong morphism \( f \) from \( i \) to \( j \) will be denoted \( i \xrightarrow{f} j \).

Note that a strong morphism is always type-compatible. Strong morphisms will be used when type equality is preferred to type compatibility “up to inheritance”. We will later see why this is sometimes necessary. Note also that there is no restriction wrt. the “abstraction level” of mapped nodes. We especially allow to map (more generalised) concrete nodes to (more specialised) abstract nodes.

Type-compatible morphisms compose, and strong morphisms are closed under composition:

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

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

Now we are able to build a category of typed graphs given a fixed type graph \( T \):

**Definition 5 (Category \( \mathcal{G}^T \)).** Let \( T \) be a type graph. We define \( \mathcal{G}^T \) to be the category which has \( T \)-typed graphs as objects and type-compatible morphisms between them as arrows.
The main consequences of this definition are shown in Fig. 9. The type graph $T$ is at the top (reflexive elements $\{\}\$ and the largest element $O$ are not shown). There are three typed graphs $i$, $j$, and $k$. Since $B$ inherits the association $e$, $i$ is a well-typed object structure. Since $A$-objects may be linked polymorphically to $C$- or $D$-objects, $j$ is an admissible typing. Moreover, $m_1$ and $m_2$ are two type-compatible morphisms (e.g. $m_1(\cdot A) = :B$ yields $i(m_1(\cdot A)) = B \leq A = k(\cdot A)$).

In the sequel, we let

$$\tau : \{ (g : G \to T) \xrightarrow{f} (h : H \to T) \mapsto (G \xrightarrow{f} H) \}$$

be the functor which forgets the typing structure, i.e., it maps a typed graph to the underlying graph without typing.

Having defined the underlying category of typed graphs $\mathcal{G}^T$, we are now able to analyse the transformation of typed graphs. Ideally, we simply would like to use double-pushout graph transformations, as the DPO approach leads to a rich set of properties and theorems that are both interesting and helpful for users of our model. As type changes of objects at runtime must not occur in an object-oriented system, graph transformation rules contain only strong morphisms. As we will see later, this decision enables a unique pushout complement.

**Definition 6 (Graph Transformation Rule).** In $\mathcal{G}^T$, a graph transformation rule $l \xrightarrow{v} k \xrightarrow{w} r$ consists of typed graphs $l$, $k$, and $r$, called the left-hand side, gluing graph, and the right-hand side, respectively, and two injective and strong graph morphisms $v$ and $w$. 

\footnote{Pairs $(x, x) \in V_T \times V_T$ for all $x \in V_T$}
Note that the span \((m_1, m_2)\) in Fig. 9 is not a rule, as the morphisms are not strong.

**Definition 7 (Graph Transformation).** Given an arbitrary graph transformation rule \(p = l \xrightarrow{\mu} k \xrightarrow{\nu} r\) and a typed graph \(g\) with a type-compatible graph morphism \(m: l \rightarrow g\), called the match, a graph transformation \(g \xrightarrow{\mu m} h\) from \(g\) to a typed graph \(h\) is given by the following double-pushout (DPO) diagram, where (1) and (2) are pushouts in the category \(\mathcal{G}^T\).

\[
\begin{array}{c}
l \xrightarrow{\nu} k \xrightarrow{\mu} r \\
\downarrow m \quad \downarrow n \\
g \xrightarrow{\mu} \quad d \xrightarrow{s} h
\end{array}
\]

As for transformation rules requiring strong graph morphisms, consider the situation in Fig. 10. The graph in Fig. 10a allows two non-isomorphic pushout complements (cf. Fig. 10b and Fig. 10c) due to the possible variety caused by \(l\) being not strong. Allowing non-unique pushout complements would make the result applying a rule to a match non-deterministic, a situation that is usually not desired.

\[
\begin{array}{c}
:aSubdir \leftarrow l :C'er \\
\downarrow m \\
:aSubdir
\end{array}
\quad
\begin{array}{c}
:aSubdir \leftarrow l :C'er \\
\downarrow m' \\
:aSubdir
\end{array}
\quad
\begin{array}{c}
:aSubdir \leftarrow l :C'er \\
\downarrow m' \\
:aSubdir
\end{array}
\]

(a) Original situation \quad \hspace{1cm} (b) First pushout complement \quad \hspace{1cm} (c) Second pushout complement

Figure 10: Two possible pushout complements if rule morphism is not strong

In order to benefit from the various theoretical results of the double-pushout approach, we have to prove that our category \(\mathcal{G}^T\) is an adhesive HLR category. For this to show, we need to investigate the properties of (strong) monomorphisms, pullbacks, and pushouts in \(\mathcal{G}^T\) as well as the interplay between them. The next sections deal with monomorphisms, pullbacks, and pushouts in \(\mathcal{G}^T\), respectively. Finally, we prove that \((\mathcal{G}^T, M)\), with \(M\) being the class of all strong monomorphisms, is an adhesive HLR category.

5. 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 \(m: i \rightarrow j\) is a monomorphism. The reverse statement is also true, but we need the existence of the largest element \(O\) of \(\leq\): If \(m: g \rightarrow h\) is a monomorphism then \(m(v_1) = m(v_2)\)
can be detected by mappings \( k_1, k_2: \{\cdot O\} \to G \) with \( k_i(\cdot O) = v_i \ (i \in \{1, 2\}) \). 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 \( m: \{1: A; 2: B\} \to \{1: C\} \) with \( m(\cdot A) = m(\cdot B) = : C \) is a monomorphism, as there do not exist any morphisms \( p, q: X \to \{1: A; 2: B\} \) 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 \( G \).

**Proposition 8.** Epimorphisms of \( G^T \) are exactly the surjective morphisms. Monomorphisms of \( G^T \) are exactly the injective morphisms.

Conventionally, in category theory, extremal monomorphisms are often the right choice for representing substructures (subsets, subgraphs, and the like). 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 \( G^T \) a morphism \( m: \{1: B\} \to \{1: A\} \) with \( A < B \) is monic (cf. Proposition 8), but not an isomorphism, because a hypothetical inverse \( n \) would have to “upcast” \( n(\cdot A) = : B \), which is not possible. Thus \( m \) is not extremal, because \( m = id \circ m \).

It is worth noting that this coincides with the concept of strong monomorphisms:

**Proposition 9 (Strong and Extremal Monos coincide).** A monomorphism in \( G^T \) is extremal if and only if it is strong.

**Proof.** See [17, Prop. 7].

Because of this result, it is reasonable to denote an extremal monomorphism \( m \) from \( i \) to \( j \) by \( i \xrightarrow{m} j \).

6. **Pushouts**

In order to define and apply double-pushout graph transformation rules in the category \( G^T \), 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 \( T \) with \( V_T = \{O, B, C\} \) and \( \leq = \{(B, O); (C, O)\} \) plus all reflexive pairs. Then

\[
\xymatrix{C \ar[r] & O \ar[r] & B}
\]

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

\[
\xymatrix{1: B \ar[r] & 1: O \ar[r] & 2: O \ar[r] & 12: O}
\]
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

\[
\begin{array}{c}
 j \\
\downarrow \beta \\
g \downarrow \alpha \\
\hline
 h
\end{array}
\]

(5)

to admit a pushout. For this we denote with \(\bigwedge X\) the greatest lower bound of a subset \(X \subseteq V_T\) if it exist.\(^{21}\) Let furthermore \(G := \tau(g)\), \(H := \tau(h)\), and \(J := \tau(j)\) with the forgetful functor \(\tau\) introduced in Sect. 4. We denote with \([h, j]: H + J \to T\) the disjoint union of \(h\) and \(j\) and we need the usual relation

\[
\sim := \{(\alpha(x), \beta(x)) \mid x \in G\}
\]

(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 \(H + G \cdot J := (H + J)/\equiv\) together with the canonical graph morphisms \(\pi : J \to H + G \cdot J\) and \(\beta : H \to H + G \cdot J\) (which map \(v\) to \([v]\equiv\)) that make up the \(G\)-pushout of \(\alpha\) and \(\beta\).

**Theorem 10 (Characterisation of Pushouts).** The span (5) admits a pushout in \(G^T\) 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

\[
\begin{array}{ccc}
g & \xrightarrow{\alpha} & h \\
\downarrow \beta & & \downarrow \pi \\
j & \xrightarrow{\beta} & j \\
\end{array}
\]

(7)
is a pushout in \(G^T\), where \(p : H + G \cdot J \to T\) is defined by

\[
p([v]) = \bigwedge \{[h, j](x) \mid x \in [v]\}
\]
on vertices and \(p([e]) = [h, j](e)\) on edges.

**Proof.** A detailed proof of this theorem is given in [17] for type graphs without abstraction. It is, however, easy to see that it directly carries over to type graphs with node type abstraction. \(\square\)

The following corollaries are later needed for proving that our category is an adhesive HLR category:

**Corollary 11.** \(G^T\) has all pushouts along extremal monomorphisms. In such a pushout the extremal monomorphism is preserved under the pushout.

---

\(^{21}\)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 \(\bigcap_{i \in I} Y_i\).
Theorem 10 can be alternatively formulated as

Corollary 12. A commutative diagram $D =$

$$
\begin{array}{ccc}
\alpha & \Rightarrow & h \\
\downarrow s & & \downarrow s \\
\beta & \Rightarrow & q
\end{array}
$$

s.t. $\tau(D)$ 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. See [15, p. 9]. □

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

7. 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 6 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

$$\{C\} \overset{\cdot}{\rightarrow} \{A\} \overset{\cdot}{\leftarrow} \{B\}$$

possesses the pullback

$$\{C\} \overset{\cdot}{\leftarrow} \{O\} \overset{\cdot}{\rightarrow} \{B\}.$$

But pullback construction fails in more complex situations: Let a type graph be given by the class diagram in Fig. 11 in which the partial order is generated by the depicted arrows.

![Figure 11: A type graph](image-url)
Then the co-span
\[ \{ :C \} \longrightarrow \{ :A \} \longleftarrow \{ :B \} \]
admits no pullback, because there are two incompatible candidates, namely the spans
\[ \{ :C \} \longrightarrow \{ :D \} \longrightarrow \{ :B \} \quad \text{and} \quad \{ :C \} \longleftarrow \{ :E \} \longrightarrow \{ :B \} , \]
and a minimal candidate
\[ (8) \quad \{ :C \} \longleftarrow \{ :D \} \longrightarrow \{ :B \} , \]
for which, however, two different mediators exist from
\[ \{ :C \} \longleftarrow \{ :O \} \longrightarrow \{ :B \} . \]

The example justifies the introduction of the largest element \( O \), since otherwise it seems to be difficult to find a feasible criterion for a pullback to exist: If we omitted \( O \) in Figure 11 there would indeed be a pullback, namely the span \( (8) \) (which seems to be weird because the middle graph possesses two vertices).

In order to avoid these degenerate pullbacks we return to the original situation in which \( O \) exists. We want to find a necessary and sufficient criterion for a co-span
\[ (9) \quad j \xrightarrow{\beta} g \xleftarrow{\alpha} h \]
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 \( B \lor C \) to denote the least upper bound if it exists.

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 \( G \) 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 13 (Characterisation of Pullbacks).** The co-span \( (9) \) admits a pullback if and only if
\[ \forall (v_1, v_2) \in V_{H \times_G J} : h(v_1) \lor j(v_2) \exists. \]
If this condition is met, the square
\[ (10) \]
\[ \begin{array}{ccc}
g \xrightarrow{\alpha} h \\
\beta \downarrow & & \downarrow \pi_1 \\
j \xleftarrow{\pi_2} p \end{array} \]

\( \lor \) 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 \( Y_1 \cup Y_2 \).
is a pullback in $G^T$, where $p : H \times_G J \to T$ is defined by
\[ p(v_1, v_2) = h(v_1) \lor j(v_2) \]
on vertices and
\[ p(e_1, e_2) = h(e_1) = j(e_2) \]
on edges.

**Proof.** Again, the proof of [17] smoothly carries over to the present case with node type abstraction. \hfill \square

We obtain the following consequences:

**Corollary 14.**

1. If in (9) 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 type in $V_T - \{O\}$ has exactly one direct supertype\(^{23}\) all pullbacks exist.
4. Extremal monomorphisms as well as strong morphisms are preserved under pullbacks.

**Proof.** See [15, p. 10]. \hfill \square

Theorem 13 can be alternatively formulated:

**Corollary 15.** A commutative diagram $D = \begin{align*}
\begin{array}{ccc}
g & \xrightarrow{\alpha} & h \\
\downarrow{\delta} & & \downarrow{\delta} \\
\gamma & \xrightarrow{j} & q
\end{array}
\end{align*}

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

**Proof.** See [15, p. 10]. \hfill \square

**8. Adhesiveness**

In this section, we intend to show that $G^T$ is an adhesive HLR category\(^ {24}\) for the class $\mathcal{M}$ of all extremal monomorphisms. This ensures the results discussed in Sec. [1] We explicitly list main results that we obtain in Cor. [18]

---

\(^{23}\) As is the case in any programming language that prohibits multiple inheritance.

\(^{24}\)cf. [4]
Theorem 16. $\mathcal{G}^T$ is an adhesive HLR category for the class $\mathcal{M}$ of all extremal monomorphisms.

Proof. Since this result is very relevant for this paper, we include a proof sketch. Due to Prop. 4, Prop. 9, and [16, Prop. 7.62(2)], $\mathcal{M}$ is closed under composition (also with isomorphisms) and decomposition, 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 11 and 14). It remains to show that pushouts along $\mathcal{M}$-morphisms are VK squares, cf. [4, Def. 4.9]. Let therefore a commutative cube be given with a pushout along the extremal monomorphism $\alpha$ at the bottom and two rear pullbacks (Fig. 12). From Corollaries 11 and 14(4) we can deduce that $\alpha$ and $\alpha'$ are also extremal monomorphisms (which is already indicated in Fig. 12).

We now show that the top face in Fig. 12 is a pushout $\iff$ the two front faces are pullbacks.

$\Rightarrow$: By Corollary 11 and Proposition 9, $\alpha'$ 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 15 it suffices to show that $c = d \circ \alpha'$ \lor h \circ i_1$ and $b = d \circ \beta' \lor i \circ i_2$ on vertices. The first statement follows immediately as $\alpha'$ is strong. Let therefore $z \in \tau(b) - \alpha'(\tau(a))$. 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 12 yields $j(\beta(i_2(z))) = i(i_2(z))$. Thus, by applying Corollary 15 and by recalling that $j \circ i_3 \leq d$, we finally obtain $b(z) = d(\beta(z))$ as desired.

$\Leftarrow$: 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 12 it suffices to show that $d \circ \alpha' = c$ and $d \circ \beta' = b$ on $\tau(b) - \alpha'(\tau(a))$. The first statement is immediate because $\alpha'$ is strong by Corollary 14(4). Let therefore $z \in \tau(b) - \alpha'(\tau(a))$. 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 12 yields $j(\beta(i_2(z))) = i(i_2(z))$. Thus, by applying Corollary 15 and by recalling that $j \circ i_3 \leq d$, we finally obtain $b(z) = d(\beta(z))$ as desired.

The following technical fact prepares the concluding main result of this section:

---

25For a detailed proof, see [15, pp. 10–12].
Proposition 17. In $G^T$, binary coproducts are compatible with $M$.

Proof. See [18, Fact 7].

Since all graph transformation rules (cf. definition [1] in $G^T$ are spans of two extremal monomorphisms, we obtain the well-known concurrency theorems for the DPO approach:

Corollary 18. The following results for graph transformation based on $G^T$ and the class $M$ of all extremal monomorphisms are valid due to Theorem 16 and Proposition 17:

- Local Church Rosser Theorem for pairwise analysis of sequential and parallel independence [4, Thm. 5.12]
- Parallelism Theorem for applying independent rules and transformations in parallel [4, Thm. 5.18]
- Concurrency Theorem for applying E-related dependent rules simultaneously [4, Thm. 5.23]

9. Motivating Example Revisited

Together with the results proved in the previous sections, we can now revisit our example from Section 2 and specify transformation rules using inheritance and abstraction. The universal “Delete Object” rule is depicted in Fig. 13a. The rule for creating a subdirectory in any container is shown in Fig. 13b. The universal “Retarget Link” rule is shown in Fig. 14: the figure also demonstrates how the rule can be applied to a concrete instance $G$.

Figure 13: Rules for creating/deleting

Now consider applying rule in Fig. 13b multiple times. All these creation operations are obviously pairwise parallel independent. Due to Corollary 18, all these transformations can be done in parallel due to the Parallelism Theorem, or sequentially in arbitrary order due to the Local Church-Rosser Theorem, always yielding the same result (up to isomorphism).
10. Abstraction

Until now, we did not care about whether a node of a graph to be transformed is typed in an abstract or a concrete node in the type graph. However, in order to be able to properly simulate a running object-oriented system, we need to find a property of a graph transformation rule which ensures that the graph transformation preserves concreteness, i.e., which guarantees that every “valid” graph transformation rule applied to a concrete typed graph \( g \) yields another concrete typed graph \( h \). This is important as the object-oriented paradigm demands that objects be members of concrete classes.

This desired property is based on the following definition (recall that \( A \subseteq V_T \) denotes the set of abstract vertices of the type graph \( T \), cf. Def. 1):

**Definition 19 (Concreteness Preserving Type-Compatible Morphism).**

A type compatible morphism \( f: g \to h \) between the \( T \)-typed graphs \( g: G \to T \) and \( h: H \to T \) is said to preserve concreteness if the conditions (11) and (12) hold:

1. \( \forall v \in V_H \setminus f_V(V_G) : h_V(v) \in V_T \setminus A \)
2. \( \forall v \in V_G : g_V(v) \in V_T \setminus A \implies h_V(f_V(v)) \in V_T \setminus A \)

Condition (11) ensures that new nodes being added by a concreteness preserving morphism are concrete. Condition (12) prevents concrete nodes to be mapped to abstract nodes.
nodes. Together, both conditions guarantee that
\[ g \text{ is concrete } \Rightarrow f(g) \text{ is concrete} \]
for any concreteness preserving morphism \( f : g \to h \). Note that Condition (12) holds for all rule morphisms in the previous examples. Moreover, Condition (11) would be violated if a containee object was added in Fig. 13b instead of a concrete subdirectory.

Based on this definition, we are now able to define transformation rules that preserve concreteness of typed graphs:

**Definition 20 (Concreteness Preserving Graph Transformation Rule).**
A graph transformation rule \( p = (l \leftarrow k \rightarrow r) \) is said to preserve concreteness if \( j \) preserves the concreteness of \( k \).

Note that due to strict type compatibility, Cond. (12) is always true for strong morphisms. Hence, for a strong morphism \( f \), it suffices to show the validity of Cond. (11) in order to prove that \( f \) preserves concreteness.

In the sequel we assume that for every typed graph \( g : G \to T \), the underlying graph \( G \) is defined, without explicitly naming it.

For the following propositions, we assume a graph transformation as depicted in Fig. 15.

**Proposition 21.** For a type-compatible morphism \( m' : r \to h \) in a pushout \((h : H \to T, j' : d \to h, m' : r \to h)\) of the two type-compatible morphisms \( j : k \to r \) and \( m^* : k \to d \) with \( j \in \mathcal{M} \) we have:
\[
\forall v \in V_R \setminus j_V(V_K) : r_V(v) = h_V(m'_V(v))
\]

**Proof.** This follows directly from Theorem 10. \( \square \)

The following two propositions show that \( h \) is concrete if we start the transformation with a concrete graph \( g \) and the graph transformation rule preserves concreteness:

**Proposition 22.** The type-compatible morphisms \( j' : d \to h \) and \( m' : r \to h \) in a pushout \((h : H \to T, j', m')\) of the two type-compatible morphisms \( j : k \to r \) and \( m^* : k \to d \) with \( j \in \mathcal{M} \) preserve concreteness, if \( d \) is concrete and \( j \) preserves concreteness.

22
Assume that the abstraction predicate (cf. Definition 20) aggravates or even prevents verification of been applied preserve concreteness according to Def. 20. Theorem 24 states that all graph transformation rules that have tionally, if a chain of graph transformations concreteness, then transforming a concrete graph always yields concrete graphs. Addi-

Theorem 24. Proofs.

Proof. Assume that \( j' \) does not preserve concreteness. By the remark after Def. 20, \( j' \) fulfills condition \([12] \). So assume that condition \([11] \) is not satisfied. Then we obtain

\[
\exists v \in \mathcal{V}_H \setminus j'_V(V_G) : h_V(v) \in A
\]

\[
\Rightarrow \exists v \in \mathcal{V}_R \setminus j_V(V_K) : h_V(m'_V(v)) \in A
\]

(pushout property) 

\[
\Rightarrow \exists v \in \mathcal{V}_R : h_V(v) \in A
\]

(Proposition 21) 

\[
\Rightarrow j \text{ does not preserve concreteness. } \]

Assume that \( m' \) does not preserve concreteness. Then there are two possibilities:

1. Condition \([11] \) is not satisfied. Then we obtain

\[
\exists v \in \mathcal{V}_H \setminus m'_V(V_G) : h_V(v) \in A
\]

\[
\Rightarrow \exists v \in \mathcal{V}_D : h_V(j'_V(v)) \in A
\]

(pushout property) 

\[
\Rightarrow \exists v \in \mathcal{V}_D : d_V(v) \in A
\]

\( (j' \text{ is extremal}) \)

\[
\Rightarrow d \text{ is not concrete. } \]

2. Condition \([12] \) is not satisfied. Then there is a concrete node \( v \in \mathcal{V}_R \) which is mapped to an abstract node \( v_2 \in \mathcal{V}_H \). Assume that \( v \in \mathcal{V}_R \setminus j_V(V_K) \), then \( v \) has to be abstract, since \( v \) will not be “down-typed”. This is a contradiction, because \( j \) preserves concreteness. Assume that \( v \in j_V(V_K) \), than there is \( v_1 \in \mathcal{V}_D \) with the same pre-image in \( V_K \). \( v_1 \) is concrete since \( d \) is concrete. \( v_1 \) will be mapped to the abstract node \( v_2 \) which is a contradiction to the concreteness preserving property of \( j' \).

Proposition 23. A typed graph \( h : \mathcal{H} \to \mathcal{T} \) is concrete, if there is a concreteness preserving type-compatible morphism \( j' : d \to h \) for a concrete typed graph \( d : D \to T \).

Proof. Since \( j' \) preserves concreteness, condition \([12] \) yields \( \forall v \in \mathcal{V}_D : d(v) \in \mathcal{V}_T - A \implies \forall v \in \mathcal{V}_D : h_V(j'_V(v)) \in \mathcal{V}_T - A \), i.e., \( h_V(v) \in \mathcal{V}_T - A \) for all \( v \in j'_V(V_D) \). Moreover, \( h_V(v) \in \mathcal{V}_T - A \) for all \( v \in \mathcal{V}_H \setminus j'_V(V_D) \) by \([11] \). Thus \( h_V(v) \in \mathcal{V}_T - A \) for all \( v \in \mathcal{V}_H \).

Combining the propositions \([21] \), \([22] \), and \([23] \) we can state the main result of this section:

Theorem 24. The graph transformation \( g \overset{m}{\Rightarrow} h \) in \( G^T \) with the match \( m : l \to g \) preserves concreteness of \( g \) if the graph transformation rule \( p \) preserves concreteness.

Proof. “\( \Leftarrow \)”: Concreteness of \( h \) follows from Corollary \([11] \) and Proposition \([22] \) and \([23] \). 

“\( \Rightarrow \)”: Concreteness of the nodes in \( \mathcal{V}_R \setminus j_V(V_K) \) follows from Proposition \([21] \). 

Theorem \([24] \) guarantees that if one uses only graph transformation rules preserving concreteness, then transforming a concrete graph always yields concrete graphs. Additionally, if a chain of graph transformations \( g \Rightarrow g' \Rightarrow g'' \Rightarrow \ldots \Rightarrow h \) is given where all graphs are concrete, Theorem \([24] \) states that all graph transformation rules that have been applied preserve concreteness according to Def. \([20] \).

One might reason that asymmetry of graph transformation rules in the presence of the abstraction predicate (cf. Definition \([20] \) aggravates or even prevents verification of
Corollary 18 because the proof of the Local Church Rosser Theorem in [6] relies on the construction of inverse graph transformation rules. However, the proof can be carried out in the adhesive HLR category $\mathcal{G}T, \mathcal{M}$ (by Theorem 16), in which graph transformation rules are still symmetric (cf. Definition 6) and inverse rules do not appear in the statements of Local Church Rosser. Thus Corollary 18 remains valid in the presence of node type abstraction.

11. 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 4), the innovation of our work is the proof that our framework is well-behaved w.r.t. the interplay of pushouts and pullbacks (adhesiveness) in the presence of node type abstraction and multiple inheritance. 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), arbitrary multiplicities, and other limiting constraints. The proof of Theorem 16 infers adhesiveness of $\mathcal{G}T$ from adhesiveness of $\mathcal{G}$. It is evident that (weak) adhesiveness of $\mathcal{G}T$ can be derived from (weak) adhesiveness of $\mathcal{G}$ in the same way, if $\mathcal{G}$ is an extension of ordinary graphs. Thus there is a direction for future research: Is (weak) adhesiveness invariant under enlargements of $\mathcal{G}T$ such as attributed graphs [9], place/transition nets [3], or sketched OCL [26] constraints [19]?

It is also a goal of forthcoming research to define single pushout rewriting [5] with inheritance: For this, transformation rules $r: L \rightarrow R$ with $r$ being 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 superrule $r'$ could also be interpreted as a negative application condition [4], if $r'$ is the identity.

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

References


Object Constraint Language