Derivations in object-oriented graph grammars

Ana Paula Lüdtke Ferreira[1] and Leila Ribeiro[2]

[1] Universidade do Vale do Rio dos Sinos – UNISINOS
[2] Universidade Federal do Rio Grande do Sul – UFRGS
anapaula@exatas.unisinos.br, leila@inf.ufrgs.br

Abstract This work aims to extend the algebraic single-pushout approach to graph transformation to model object-oriented systems structures and computations. Graphs whose sets of nodes and edges are partially ordered are used to model the class hierarchy of an object-oriented system, making the inheritance and overriding relations explicit. Graphs typed over such structures are then used to model systems of objects, whose behaviour is given by an object-oriented graph grammar. It will be shown how the derivations which arise from those grammars are compatible with the way object-oriented programs are defined and executed.

1 Introduction

The principles behind the object-oriented paradigm — *data and code encapsulation*, *information hiding*, *inheritance* and *polymorphism* — fit very well into the needs of modular system development, distributed testing, and reuse of software faced by system designers and engineers, making it perhaps the most popular paradigm of system development in use nowadays. The most distinguished features of object-oriented systems are inheritance and polymorphism, which make them considerably different from other systems in both their architecture and model of execution.

Inheritance is the construction which permits a class (in the class-based approach [6]) or an object (in the object-based approach [27]) to be specialized from an already existing one. This newly created object carries (or "inherits") all the data and the actions belonging to its primitive object, in addition to its own data and actions. If this new object is further extended using inheritance, then all the new information will also be carried along. The relation "inherits from" induces a hierarchical relationship among the defined classes of a system, which can be viewed as a set of trees (single inheritance) or as an acyclic graph (multiple inheritance). The subclass polymorphism [4] relation follows exactly the same structure.

Subclass polymorphism specifies that an instance of a subclass can appear wherever an instance of a superclass is required. Hence, in order to apply it, the inheritance relationship must be made explicit by any formalism aiming to model object-oriented systems. One of the most useful ways inheritance and polymorphism can be used is through method redefinition. The purpose of method redefinition is to take advantage of the polymorphism concept through the mechanism

known as *dynamic binding*, which is an execution time routine to determine what piece of code should be called when a message is sent to an object.

Object-oriented programs usually make use of inheritance (which is the most common way of code reuse) and method redefinition to achieve their goals. Therefore, it should be expected that formalisms for the specification of object-oriented architectures or programs reflect those concepts, otherwise the use of such formalisms will neglect concepts that have a major influence in their organization and model of execution.

Graph grammars have been used to specify various kinds of software systems, where the graphs correspond to the states and the graph productions to the operations or transformations of such systems [16]. System specifications through graphs often rely on labelled graphs or typed graphs to represent different system entities [1], [2], [9], [11], [19], [21], [23], [25]. However, labelling and typing do not reflect the inheritance relation among objects, and polymorphism cannot be applied if it is not made explicit. To do so, a class, if represented by a node or edge in a graph, should have a multiplicity of labels or types assigned to it, which is not compatible with the usual way labelling or typing are portrayed (as functions).

This work aims to extend the algebraic single-pushout approach to graph grammars [13], [20] to model object-oriented systems structures and computations. Graphs whose sets of nodes and edges are partially ordered [10] are used to model the class hierarchy of an object-oriented specification, making the inheritance and overriding [5], [26] relations explicit. Typing morphisms and graph morphisms are compatible with the order structure, adequately reflecting inherited attributes and methods. Some preliminary results were presented in [17], and here we go further into the theory. The main result of this paper is to show that a direct derivation in an object-oriented graph grammar is a pushout on the category of object-oriented graphs and their morphisms. It is an important result in the sense that the algebraic approach to graph grammars rely on categorical constructs to express most of its results. Having proven that those constructs exist in the new setting, all results derived from them can be automatically used.

This paper is structured as follows: Section 2 presents class-model graphs, which are graphs whose node and edge sets are special partially ordered sets, and whose structure fits the way object-oriented programming specifications are built. Class-model graphs are meant to portray the structure of object-oriented systems, and they will provide a typing structure to the graphs and grammars presented in the rest of the text. Section 3 presents the graphs and grammars used to represent object-oriented systems and programs. It is also shown how the semantics of dynamic binding can be given in terms of pushouts in a suitable category. Finally, Section 4 presents some final remarks and directions for future work.

2 Class-model graphs

Inheritance, in the context of the object-orientation paradigm, is the construction which permits an object[1] to be specialized from an already existing one. When only single inheritance is allowed, the hierarchical relationship among the classes of a system can be formalized as a set of trees. A *strict relation*, defined below, formalizes what should be the fundamental object-oriented hierarchical structure of classes.

Definition 1 (Strict relation). *A finite binary relation $R \subseteq A \times A$ is said a* strict relation *if and only if it has the following properties: if $(a, a') \in R$ then $a \neq a'$ (R has no reflexive pairs); if $(a, a_1), (a_1, a_2), \ldots, (a_{n-1}, a_n), (a_n, a') \in R$, $n \geqslant 0$, then $(a', a) \notin R$ (R has no cycles); for any $a, a', a'' \in A$, if $(a, a'), (a, a'') \in R$ then $a' = a''$ (R is a function).*

Notice that the requirement concerning the absence of cycles and reflexive pairs on strict relations is consistent with both the creation of classes and redefinition of methods (overriding). A class is defined as a specialization of at most one class (single inheritance), which must exist prior to the creation of the new one. A method can only redefine another method (with the same signature) if it exists in an already existing primitive class. Hence, neither a class can ever be created nor a method can be redefined in a circular or reflexive way. It is easy to prove that the reflexive and transitive closure of a strict relation R is a partial order, so in the rest of this paper we will assume that result.

Definition 2 (Strict ordered set). *A* strict ordered set *is a pair $\langle P, \sqsubseteq_P^* \rangle$ where P is a set, $\sqsubseteq_P$ is a strict relation, and $\sqsubseteq_P^*$ is its reflexive and transitive closure.*

Object-oriented systems consist of instances of previously defined classes which have an internal structure defined by attributes and communicate among themselves solely through message passing. That approach underlies the structure of the graphs used to model those systems, as defined next.

Definition 3 (Class-model graph). *A class-model graph is a tuple $\langle V_\sqsubseteq, E_\sqsubseteq, L, src, tar, lab \rangle$ where $V_\sqsubseteq = \langle V, \sqsubseteq_V^* \rangle$ is a finite strict ordered set of vertices, $E_\sqsubseteq = \langle E, \sqsubseteq_E^* \rangle$ is a finite strict ordered set of (hyper)edges, $L = \{\text{attr}, \text{msg}\}$ is an unordered set of two edge labels, $src, tar : E \to V^*$ are monotone order-preserving functions, called respectively* source *and* target *functions, $lab : E \to L$ is the edge* labelling *function, such that the following constraints hold:*

Structural constraints: *for all $e \in E$, the following holds:*
- *if $lab(e) = \text{attr}$ then $src(e) \in V$ and $tar(e) \in V^*$, and*
- *if $lab(e) = \text{msg}$ then $src(e) \in V^*$ and $tar(e) \in V$.*

[1] Object-based and class-based models are actually equivalent in terms of what they can represent and compute [27], so we will sometimes use the terms *object* and *class* interchangeably. We hope this will not cause any confusion to the reader.

Sets $\{e \in E \mid lab(e) = \text{attr}\}$ *and* $\{e \in E \mid lab(e) = \text{msg}\}$ *will be denoted by* $E|_{\text{attr}}$ *and* $E|_{\text{msg}}$*, respectively.*

Order relations constraints: *for all* $e \in E$*, the following holds:*

1. *if* $(e, e') \in \sqsubseteq_E$ *then* $lab(e) = lab(e') = \text{msg}$*,*
2. *if* $(e, e') \in \sqsubseteq_E$ *then* $src(e) = src(e')$*,*
3. *if* $(e, e') \in \sqsubseteq_E$ *then* $(tar(e), tar(e')) \in \sqsubseteq_V^+$*, and*
4. *if* $(e', e) \in \sqsubseteq_E$ *and* $(e'', e) \in \sqsubseteq_E$ *, with* $e' \neq e''$*, then* $(tar(e'), tar(e'')) \notin \sqsubseteq_V^*$ *and* $(tar(e''), tar(e')) \notin \sqsubseteq_V^*$*.*

That approach underlies the structure of the graphs used to model those systems. Each graph node is a class identifier, hyperarcs departing from it correspond to its internal attributes, and messages addressed to it consist of the services it provides to the exterior (i.e., its methods). Notice that the restrictions put to the structure of the hyperarcs assure, as expected, that messages target and attributes belong to a single object.

Inheritance and overriding hierarchies are made explicit by imposing that graph nodes (i.e., the objects) and message edges (i.e., methods) are strict ordered sets (Definition 2). Notice that only single inheritance is allowed, since $\sqsubseteq_V$ is required to be a function. The relation between message arcs, $\sqsubseteq_E$, establishes which methods will be redefined within the derived object, by mapping them. The restrictions applied to $\sqsubseteq_E$ ensure that methods are redefined consistently, i.e., only two message arcs can be mapped (i), their parameters are the same (ii), the method being redefined is located somewhere (strictly) above in the class-model graph (under $\sqsubseteq_V^+$) (iii), and only the closest message with respect to relations $\sqsubseteq_V$ and $\sqsubseteq_E$ can be redefined (iv).

Since so many structures in computer science are usually represented as graphs, and a number of other structures in the same field are adequately represented by order relations, the idea of combining the two formalisms is appealing. However, this combination does not appear often in the literature. In [3], for instance, "partially ordered graphs" are defined, which consist of ordinary labelled graphs together with a tree structure on their nodes. Partially ordered graph grammars are also defined, which consist of graph productions and tree productions, which must assure that the rewriting process maintains the tree structure. They are applied on lowering the complexity of the membership problem of context sensitive string grammars. Graphs labelled with alphabets equipped with preorders (i.e., reflexive and transitive binary relations) appear in [22] to deal with variables within graph productions. Unification of terms can be achieved (by the rule morphism) if the terms are related by the order relation, which means that the ordering is actually a sort of variable typing mechanism. The concluding remarks of this work present some ideas on using the framework to describe inheritance, but this direction seems not having been pursuit.

It can be shown that class-model graphs and suitable morphisms between them form a cocomplete category, and the two most common operations of object-oriented system extension, namely *object extension by inheritance* and *object creation by aggregation*, can be expressed in terms of colimits in it [18]. Our focus, however, lies on object-oriented *systems*, i.e., a collection of instances

from the classes defined in a class-model graph. Such structure will be characterized as a graph typed over a class-model graph.

Remark 1. For any partially ordered set $\langle P, \sqsubseteq_P \rangle$, an induced partially ordered set $\langle P^*, \sqsubseteq_{P^*} \rangle$ can be constructed, such that for any two strings $u = u_1 \dots u_n, v = v_1 \dots v_n \in P^*$, we have that $u \sqsubseteq_{P^*} v$ if and only if $|u| = |v|$ and $u_i \sqsubseteq_P v_i$, $i = 1, \dots, |u|$. If $P_\sqsubseteq$ and $Q_\sqsubseteq$ are partially ordered sets, any monotone function $f : P \to Q$ can be extended to a monotone function $f^* : P^* \to Q^*$, with $f^*(p_1 \dots p_n) = f(p_1)f(p_2) \dots f(p_n) = q_1 \dots q_n$ where $p_1 \dots p_n \in P^*$ and $q_1 \dots q_n \in Q^*$.

Definition 4 (Hierarchical graph). *A hierarchical graph $G^\mathcal{C}$ is a tuple $\langle G, t, \mathcal{C} \rangle$, where $\mathcal{C} = \langle V_\sqsubseteq, E_\sqsubseteq, L, src, tar, lab \rangle$ is a class-model graph, G is a hypergraph, and t is a pair of total functions $\langle t_V : V_G \to V, t_E : E_G \to E \rangle$ such that $(t_V \circ src_G) \sqsubseteq_{V^*} (src \circ t_E)$, and $(t_V \circ tar_G) \sqsubseteq_{V^*} (tar \circ t_E)$.*

Hierarchical graphs reflect the inheritance of attributes and methods within the object-oriented paradigm. Notice that they are hypergraphs typed over a class-model graph. However, the typing morphism is more flexible than the traditional one [23]: an edge can be incident to any node which is connected to its typing edge in the underlying order relation. This definition reflects the idea that an object can use any attribute one of its primitive classes have, since it was inherited when the class was specialized.

Remark 2. For all diagrams presented in the rest of this text, $\mapsto$-arrows denote total morphisms whereas $\to$-arrows denote arbitrary morphisms (possibly partial). For a partial function f, $dom(f)$ represents its domain of definition, $f?$ and $f!$ denote the corresponding domain inclusion and domain restriction. Each morphism f within category **SetP** can be factorized into components $f?$ and $f!$.

Definition 5 (Hierarchical graph morphism). *Let $G_1^\mathcal{C} = \langle G_1, t_1, \mathcal{C} \rangle$ and $G_2^\mathcal{C} = \langle G_2, t_2, \mathcal{C} \rangle$ be two hierarchical graphs typed over the same class-model graph $\mathcal{C}$. A* hierarchical graph morphism *$h : G_1^\mathcal{C} \to G_2^\mathcal{C}$ between $G_1^\mathcal{C}$ and $G_2^\mathcal{C}$, is a pair of partial functions $h = \langle h_V : V_{G_1} \to V_{G_2}, h_E : E_{G_1} \to E_{G_2} \rangle$ such that the diagram (in* **SetP***)*

$$\begin{array}{ccccc} E_{G_1} & \xleftarrow{h_E?} & dom(h_E) & \xmapsto{h_E!} & E_{G_2} \\ {\scriptstyle src_{G_1}, tar_{G_1}}\downarrow & & & & \downarrow{\scriptstyle src_{G_2}, tar_{G_2}} \\ V_{G_1}^* & & \xrightarrow{\quad h_V^* \quad} & & V_{G_2}^* \end{array}$$

commutes, for all elements $v \in dom(h_V)$, $(t_{2V} \circ h_V)(v) \sqsubseteq^*_{V^*} t_{1V}(v)$, and for all elements $e \in dom(h_E)$, $(t_{2E} \circ h_E)(e) \sqsubseteq^*_E t_{1E}(e)$. If $(t_{2E} \circ h_E)(e) = t_{1E}(e)$ for all elements $e \in dom(h_E)$, the morphism is said to be strict.

A graph morphism is a mapping which preserves hyperarcs origins and targets. Ordinary typed graph morphisms, however, cannot describe correctly

morphisms on object-oriented systems because the existing inheritance relation among objects causes that manipulations defined for objects of a certain kind are valid to all objects derived from it. So, an object can be viewed as not being uniquely typed, but having a type *set* (the set of all types it is connected via the relation on nodes). Defining a graph morphism compatible with the underlying order relations assures that polymorphism can be applied consistently.

The following theorem has been proven in [17], and will be used to prove the remaining results of this paper:

Theorem 1 (Category GraphP($\mathcal{C}$)). *There is a category* **GraphP**($\mathcal{C}$) *which has hierarchical graphs over a class-model graph $\mathcal{C}$ as objects and hierarchical graph morphisms as arrows.*

Remark 3. Some usual notation from order theory is used in the rest of this paper. More specifically, given a partially ordered set $\langle P, \sqsubseteq_P \rangle$, a subset $A \subseteq P$ is an *upper set* if whenever $x \in A$, $y \in P$ and $x \sqsubseteq_P y$ we have $y \in A$. The upper set of $\{x\}$, with $x \in P$ (also called the set of all elements *above* x) is denoted by $\uparrow x$. A *lower set* and the set of all elements below some element $x \in A$, denoted by $\downarrow x$, is defined dually. An element $x \in P$ is called an *upper bound* for a subset $A \subseteq P$, written $A \sqsubseteq x$, if and only if $a \sqsubseteq_P x$ for all $a \in A$. The set of all upper bounds of A is denoted by $\mathrm{ub}(A)$, and if is has a least element (i.e., an element which is below all others), that element is denoted by $lub(A)$ or $\sqcup A$. *Lower bounds*, the set of all lower bounds $lb(A)$, and the greatest lower bound $glb(A)$ or $\sqcap A$, can be defined dually.

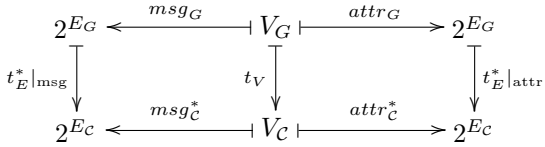
Definition 6 (Attribute and message sets). *Let $\langle G, t, \mathcal{C} \rangle$ be a hierarchical graph. Then the following functions are defined:*

- *the* attribute set function $attr_G : V_G \to 2^{E_G}$ *return for each vertex $v \in V_G$ the set $\{e \in E_G \mid src_G(e) = v \wedge lab(t_E(e)) = \mathrm{attr}\}$;*
- *the* message set function $msg_G : V_G \to 2^{E_G}$ *returns for each vertex $v \in V_G$ the set $\{e \in E_G \mid tar_G(e) = v \wedge lab(t_E(e)) = \mathrm{msg}\}$;*
- *the* extended attribute set function*, $attr^*_{\mathcal{C}} : V \to 2^E$, where $attr^*(v) = \{e \in E \mid lab(e) = \mathrm{attr} \wedge src(e) \in \uparrow v\}$;*
- *the* extended message set function*, $msg^*_{\mathcal{C}} : V \to 2^E$, where $msg^*(v) = \{e \in E|_{\mathrm{msg}} \mid tar(e) \in \uparrow v \wedge \neg \exists e' \in E|_{\mathrm{msg}} : tar(e') \in \uparrow v \wedge e' \sqsubseteq_E e\}$.*

For any hierarchical graph $\langle G, t, \mathcal{C} \rangle$ there is a total function $t^*_E : 2^{E_G} \to 2^{E_{\mathcal{M}}}$, which can be viewed as the extension of the typing function to edge (or node) sets. The function t^*_E, when applied to a set $E \in 2^{E_G}$, returns the set $\{t_E(e) \in E_{\mathcal{M}} | e \in E\} \in 2^{E_{\mathcal{M}}}$. Notation $t^*_E|_{\mathrm{msg}}$ and $t^*_E|_{\mathrm{attr}}$ will be used to denote the application of t^*_E to sets containing exclusively message and attribute (respectively) hyperarcs. Now, given the functions already defined, we can present a definition of the kind of graph which represents an object-oriented system.

Definition 7 (Object-oriented graph). *Let $\mathcal{C}$ be a class-model graph. A hierarchical hypergraph $\langle G, t, \mathcal{C} \rangle$ is an* object-oriented graph *if and only if the*

following diagram (in **Set***)*



can be constructed, and all squares commute. If, for each $v \in V_G$, the function $t_E^*|_{\text{attr}}(attr_G(v))$ is injective, $G^{\mathcal{C}}$ is said a strict *object-oriented graph. If $t_E^*|_{\text{attr}}(attr_G(v))$ is also surjective, $G^{\mathcal{C}}$ is called a* complete *object-oriented graph.*

Theorem 2. *There is a category* $\mathbf{OOGraphP}(\mathcal{C})$ *with object-oriented graphs as objects and hierarchical graph morphisms as arrows.*

Proof. (sketch) It is easy to see that object-oriented graphs are just a special kind of hierarchical graphs, and that $\mathbf{OOGraphP}(\mathcal{C})$ is a subcategory of $\mathbf{GraphP}(\mathcal{C})$(Theorem 1).

3 Object-oriented graphs grammars

Complete object-oriented graphs (Definition 7) can model an object-oriented system. However, in order to capture the system computations, a graph grammar formalism should be introduced. The most fundamental notion in a graph grammar is the concept of graph production, or rule. When a rule is intended to represent some system evolution, it must reflect the way this evolution takes place. The rule restrictions presented in this text are object-oriented programming principles, as described next.

First, no object may have its type altered nor can any two different elements be identified by the rule morphism. This is accomplished by requiring the rule morphism to be injective on nodes and arcs (different elements cannot be merged by the rule application), and the mapping on nodes to be invertible (object types are not modified).

The left-hand side of a rule is required to contain exactly one element of type message, and this particular message must be deleted by the rule application, i.e., each rule represents an object reaction to a message which is consumed in the process. This demand poses no restriction, since systems may have many rules specifying reactions to the same type of message (non-determinism) and many rules can be applied in parallel if their triggers are present at an actual state and the referred rules are not in conflict [13], [23]. Systems' concurrent capabilities are so expressed by the grammar rules, which can be applied concurrently (accordingly to the graph grammar semantics), so one object can treat any number of messages at the same time.

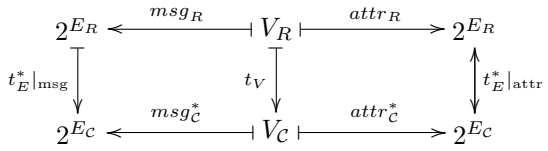
Additionally, only one object having attributes will be allowed on the left-hand side of a rule, along with the requirement that this same object must be the target of the above cited message. This restriction implements the principle

of *information hiding*, which states that the internal configuration (implementation) of an object can only be visible, and therefore accessed, by itself.

Finally, although message attributes can be deleted (so they can have their value altered[2]), a corresponding attribute must be added to the rule's right-hand side, in order to prevent an object from gaining or losing attributes along the computation. Notice that this is a *rule* restriction, for if a vertex is deleted, its incident edges will also be deleted [16]. This situation is interesting in modelling distributed systems, when pointers to objects could be made meaningless if the objects they point to are deleted. If a vertex is deleted, no rule that uses it can ever be applied, which is compatible with the idea that following a broken link is not meaningful. This consequences, however, will not be explored here.

Definition 8 (Object-oriented rule). *An* object-oriented rule *is a tuple* $\langle L^{\mathcal{C}}, r, R^{\mathcal{C}} \rangle$ *where* $L^{\mathcal{C}} = \langle L, t_L, \mathcal{C} \rangle$ *and* $R^{\mathcal{C}} = \langle R, t_R, \mathcal{C} \rangle$ *are strict object-oriented graphs and* $r = \langle r_V, r_E \rangle : L^{\mathcal{C}} \to R^{\mathcal{C}}$ *is a hierarchical graph morphism holding the following properties:*

- r_V *is injective and invertible,* r_E *is injective,*
- $\{v \in V_L \mid \exists e \in E_L : src_L(e) = v \wedge (lab \circ t_L)(e) = \text{attr}\}$ *is a singleton, whose unique element is called* attribute vertex*,*
- $\{e \in E_L \mid (lab \circ t_L)(e) = \text{msg}\}$ *is a singleton, whose unique element is called* left-side message*, having as target object the attribute vertex,*
- *the left-side message does not belong to the domain of* r*,*
- *for all* $v \in V_L$ *there is a bijection* $b_v : \{e \in E_L \mid src_L(e) = v, lab(t_L(e)) = \text{attr}\} \leftrightarrow \{e \in E_R \mid src_R(e) = r_V(v), lab(t_R(e)) = \text{attr}\}$*, such that* $t_R \circ b_v = t_L$ *and* $t_L \circ b_v^{-1} = t_R$*, and*
- *for all* $v \in V_R$*, such that* $v \notin im(r_V)$ *the diagram*



exists, commutes, and $t_E^*(attr_R(v))$ *is a bijection.*

The last condition is needed to assure that creation of objects does not generate an incomplete system (i.e., a system with an object lacking attributes). Notice that r_V is not required to be surjective, so new vertices can be added by the rule, as long as all created vertices must have exactly the attributes defined along its class-model graph.

Definition 9 (Object-oriented match). *An* object-oriented match *between a strict object-oriented graph* $L^{\mathcal{C}}$ *and a complete object-oriented graph* $G^{\mathcal{C}}$ *is a*

[2] Graphs can be enriched with algebras in order to deal with sorts, values and operations. Although we do not develop these concepts here, they can easily be added to this framework.

hierarchical graph morphism $m = \langle m_V, m_E \rangle : L^{\mathcal{C}} \to G^{\mathcal{C}}$ *such that* m_V *is total,* m_E *is total and injective, and for any two elements* $a, b \in L$*, if* $m(a) = m(b)$ *then either* $a, b \in dom(r)$ *or* $a, b \notin dom(r)$*.*

Definition 10 (Object-oriented direct derivation). *Given a complete object-oriented graph* $G^{\mathcal{C}} = \langle G, t_G, \mathcal{C} \rangle$*, an object-oriented rule* $r : L^{\mathcal{C}} \to R^{\mathcal{C}}$*, and an object-oriented match* $m : L^{\mathcal{C}} \to G^{\mathcal{C}}$*, their* object-oriented direct derivation*, or rule* application*, can be computed in two steps:*

1. *Construct the pushout of* $\Phi(r) : \Phi(L^{\mathcal{C}}) \to \Phi(R^{\mathcal{C}})$ *and* $\Phi(m) : \Phi(L^{\mathcal{C}}) \to \Phi(G^{\mathcal{C}})$ *in* **GraphP***, where* Φ *is the forgetful functor which sends an object-oriented graph to a hypergraph without any typing structure and a hierarchical graph morphism to a hypergraph morphism,* $\langle H, r' : G \to H, m' : R \to H \rangle$ *[13];*
2. *equip the result with the following typing structure on nodes and edges, resulting in the graph* $H^{\mathcal{C}} = \langle H, t_H, \mathcal{C} \rangle$ *where,*
   - *for each* $v \in V_H$*,* $t_H(v) = \sqcap \left(t_G({r'}^{-1}(v)) \cup t_R({m'}^{-1}(v)) \right)$*,*
   - *for each* $e \in E_H|_{\text{attr}}$*,* $t_H(e) = \sqcap \left(t_G({r'}^{-1}(e)) \cup t_R({m'}^{-1}(e)) \right)$*,*
   - *for each* $e \in E_H|_{\text{msg}}$*,* $t_H(e) = e'$*, where* $e' \in msg^*_{\mathcal{C}}(t_H(tar_H(e)))$*, and* $e' \sqsubseteq^*_E \sqcap [t_G({r'}^{-1}(e)) \cup t_R({m'}^{-1}(e))]$*.*

The tuple $\langle H^{\mathcal{C}}, r', m' \rangle$ *is then the resulting derivation of rule* r *at match* m*.*

An object-oriented derivation collapses the elements identified simultaneously by the rule and by the match, and copies the rest of the context and the added elements. Element typing, needed to transform the result into an object-oriented graph is done by getting the greatest lower bound (with respect the partial order relations on nodes and edges) of the elements mapped by morphisms m' and r' to the same element (the other elements have their types merely copied). The object-oriented rule restriction concerning object types (which cannot be altered by the rule) assures that it always exist, as shown in more detail below. Messages, however, need some extra care. Since graph $L^{\mathcal{C}}$ presents a single message, which is deleted by the rule application, a message on $H^{\mathcal{C}}$ comes either from $G^{\mathcal{C}}$ or from $R^{\mathcal{C}}$. If it comes from $G^{\mathcal{C}}$, which is an object-oriented graph itself, no retyping is needed. However, if it comes from R, in order to assure that $H^{\mathcal{C}}$ is also an object-oriented graph, it must be retyped according to the type of the element it is targeting on the graph $H^{\mathcal{C}}$. Notice that this element can have a different type from the one in the rule, since the match can be done to any element belonging to the lower set of the mapped entity.

It is important to realize that an object-oriented direct derivation is well defined, as shown by the proposition below.

Proposition 1. *The structure proposed in Definition 10 always exists.*

Proof. V_H is a pushout in **SetP**, hence r'_V is injective (because r_V is injective), and r'_V and m'_V are jointly surjective. For any $h \in V_H$, let ${m'_V}^{-1}(h) = \{r \in V_R \mid m'_V(r) = h\}$ and ${r'_V}^{-1}(h) = \{g \in V_G \mid r'_V(g) = h\}$. Now, for any $h \in V_H$,

$r_V'^{-1}(h)$ is a singleton (because r'_V is invertible) with unique element g, then, by definition, $\sqcap t_G(r_V'^{-1}(h)) = g$; for any $h \in V_H$ such that $m_V'^{-1}(h) \notin im(r_V)$, $m_V'^{-1}(h)$ is also a singleton, and the same reasoning applies; for any $h \in V_H$ such that $m_V'^{-1}(h) \subseteq im(r_V)$, we have that $(t_R \circ m_V'^{-1})(h) =_{V_\mathcal{C}^*} (t_L \circ r_V^{-1} \circ m_V'^{-1})(h)$; but, for the same element h we have that $(t_G \circ r_V'^{-1})(h) \sqsubseteq^*_{V_\mathcal{C}^*} (t_L \circ m_V^{-1} \circ r_V'^{-1})(h)$. But, as mentioned before, V_H is a pushout, then $(r_V^{-1} \circ m_V'^{-1})(h) = (m_V^{-1} \circ r_V'^{-1})(h)$ and therefore $(t_G \circ r_V'^{-1})(h) \sqsubseteq^*_{V_\mathcal{C}^*} (t_R \circ m_V'^{-1})(h)$. Since $\sqcap t_G(r_V'^{-1}(h))$ is defined, $\sqcap (t_G(r_V'^{-1}(h)) \cup t_R(m_V'^{-1})(h))$ always exists.

For all attributes arcs in E_L, E_R and E_G, the morphism can be reduced to an ordinary partial graph morphism (since they are only connected to themselves via the order relation), and then $\sqcap (t_G(r_E'^{-1}(h)) \cup t_R(m_E'^{-1}(h)))$ is always well defined. For message arcs, however, notice that object-oriented rules require that the only message in E_L is deleted by the rule. It means that messages arcs of E_H either come from E_G or from E_R. A message e is typed over the least element from the overriding relation with respect to its actual target vertex's type. This is assured by the fact that the typing edge must belong to the set $msg^*_\mathcal{C}(t_H(tar_H(e)))$, which, by definition, assures that there is only one edge of choice (there are no related message arcs in the set $msg^*_\mathcal{C}(v)$, for any vertex type v. □

Lemma 1. *The object $H^\mathcal{C} = \langle H, t_H, \mathcal{C} \rangle$ built according to Definition 10 is a complete object-oriented graph.*

Proof. (sketch) $L^\mathcal{C}$, $R^\mathcal{C}$ are strict object-oriented graphs, and $G^\mathcal{C}$ is a complete one. If $H^\mathcal{C}$ is an object-oriented graph, the following diagram can be constructed:

$$\begin{array}{ccccc} 2^{E_H} & \xleftarrow{msg_H} & V_H & \xrightarrow{attr_H} & 2^{E_H} \\ \downarrow{\scriptstyle t^*_E|_{\text{msg}}} & & \downarrow{\scriptstyle t_V} & & \downarrow{\scriptstyle t^*_E|_{\text{attr}}} \\ 2^{E_\mathcal{C}} & \xleftarrow{msg^*_\mathcal{C}} & V_\mathcal{C} & \xrightarrow{attr^*_\mathcal{C}} & 2^{E_\mathcal{C}} \end{array}$$

Notice that the set of added vertices $V_R \setminus r_V(V_L)$ can be viewed as a complete object-oriented graph, for it has all necessary attributes. Now, restricting the reasoning to the set of mapped vertices and attributes, one has the following: for each $v \in V_L$, let b_v be the bijection existing between the attribute edges from $A_L \subseteq E_L|_{\text{attr}}$ and $A_R \subseteq E_R|_{\text{attr}}$ defined as the last object-oriented rule restriction in Definition 8. Match m between the rule's left-side and graph $G^\mathcal{C}$ is total on vertices and arcs, and injective on arcs, and by the characteristics of the pushout construction, function m'_E is also total and injective on arcs. Notice that all edges from $G^\mathcal{C}$ are either belonging to the image of m_E (the mapped edges) or not (the context edges). Since the context edges are mapped unchanged to the graph $H^\mathcal{C}$ (and so there is a natural bijection between them), it must exist a bijection $B : E_G \leftrightarrow E_H$ which implies the existence of the trivial bijection $2^B : 2^{E_G} \to 2^{E_H}$, and since the sets V_G and V_H are isomorphic if we disregard the added vertices (note the existence of an implicit property of an

object-oriented rule that prevents it from deleting vertices, since a deletion of a vertex implies the deletion of an edge, which cannot occur, otherwise there would be no bijection b_v), it can be concluded that the right square on the diagram can be constructed. The same reasoning applies to the left square of the diagram, since the rule application assures that messages are typed right. Hence, one can conclude that $H^{\mathcal{C}}$ is a complete object-oriented graph. □

Lemma 2. *The morphisms $r' : G^{\mathcal{C}} \to H^{\mathcal{C}}$ and $m' : R^{\mathcal{C}} \to H^{\mathcal{C}}$, built according to Definition 10 are hierachical graph morphisms.*

Proof. (sketch) The morphisms r' and m' preserve the order structure, which is a sufficient condition to assure that they are actually hierarchical graph morphisms. □

Theorem 3. *Given an object-oriented graph $G^{\mathcal{C}}$, an object-oriented rule $r : L^{\mathcal{C}} \to R^{\mathcal{C}}$, and an object-oriented match $m : L^{\mathcal{C}} \to G^{\mathcal{C}}$, the resulting derivation of rule r at match m, $\langle H^{\mathcal{C}}, r', m' \rangle$ is a pushout of the arrows r and m in the category* $\mathbf{OOGraphP}(\mathcal{C})$.

Proof. Proposition 1 assures that a direct derivation can always be constructed, and Lemmas 1 and 2 show that the resulting object and morphisms belong to the category $\mathbf{OOGraphP}(\mathcal{C})$. Then, let $\langle H^{\mathcal{C}}, r' : G^{\mathcal{C}} \to H^{\mathcal{C}}, m' : R^{\mathcal{C}} \to H^{\mathcal{C}} \rangle$ be the result obtained by application of rule r under match m. Now, let ${H'}^{\mathcal{C}}$ be an object-oriented graph and $h_R : R^{\mathcal{C}} \to {H'}^{\mathcal{C}}$, $h_G : G^{\mathcal{C}} \to {H'}^{\mathcal{C}}$ be two hierarchical graph morphisms such that $h_R \circ r = h_G \circ m$. Then let $h : H^{\mathcal{C}} \to {H'}^{\mathcal{C}}$ be the hierarchical graph morphism built as follows: for all graph element (node or edge) $e \in dom(h_R) \cap dom(m')$, $h(m'(e)) = h_R(e)$; for all $e \in dom(h_G) \cap dom(r')$, $h(r'(e)) = h_G(e)$; it is easy to see that, by construction, $h_R = h \circ m'$ and $h_G = h \circ r'$. Notice that h is a hierarchical graph morphism, since all elements $e \in H$ are typed over the greatest lower bound (respecting the concerning order relation) of the elements mapped to them. It means that if exists an element $e' \in H'$ such that there are elements $e_g \in G$ and $e_r \in R$ with $h_G(e_g) = e' = h_R(e_r)$, then (h_G and h_R are hierarchical graph morphisms) $t_{H'}(e') \sqsubseteq^*_{V_{\mathcal{C}}} t_G(e_g)$ and $t_{H'}(e') \sqsubseteq^*_{V_{\mathcal{C}}} t_R(e_r)$; since $t_H(e_h) = \sqcap (t_G({r'}^{-1}(e_h)) \cup t_R({m'}^{-1}(e_h)))$ if e_h is a vertex or an attribute, and $t_H(e_h) \in msg^*_{\mathcal{C}}(t_H(tar_H(e_h)))$ if e_h is a message, then $t_{H'}(e') \sqsubseteq^*_{V_{\mathcal{C}}} t_H(e_h)$ for any $e' = h(e_h)$. Hence, h is a hierarchical graph morphism.

Suppose there is another hierarchical graph morphism $h' : H^{\mathcal{C}} \to {H'}^{\mathcal{C}}$ such that $h_R = h' \circ m'$ and $h_G = h' \circ r'$ but $h' \neq h$. Then there must be at least one graph element $e \in H$ such that $h(e) \neq h'(e)$. But r' and m' are jointly surjective, so all elements of H belong to the domain of h, so if there is an element $e \in H$ such that $h(e) \neq h'(e)$, the equalities $h_R = h' \circ m'$ and $h_G = h' \circ r'$ would not hold. □

The purpose of method redefinition is to take advantage of the polymorphism concept through the mechanism known as *dynamic binding*. Dynamic binding is usually implemented in object-oriented languages by a function pointer virtual

table, from which it is decided which method should be called at that execution point. This decision is often presented as an algorithm, which inspects the runtime type of the message receiving object to determine what is the closest redefinition (if any) of the method being sent. This decision is modelled in our work by the direct derivation construction, which assures that the type chosen for methods being sent (i.e., messages created by rule application) is the least possible regarding the redefinition chain given by the order relation on message edges.

The algebraic approach to graph grammars rely on categorical constructs to express most of its results. Semantics of computations, for instance, are usually given as categories of concrete or abstract derivations [7], which rely on the fact that direct derivations are pushouts on a suitable category of graphs and graph morphisms. It means that if the constructs used within the classical theory of graph grammars can be proven to exist in the new setting, the conclusions drawn from the former could be automatically transferred to the latter. Having a direct derivation characterized as a pushout construction in the category $\mathbf{OOGraphP}(\mathcal{C})$ is hence fundamental to inherit all previously achieved theoretical results.

4 Conclusions and future work

This paper presented a graph-based formal framework to model object-oriented specifications and computations. More specifically, an extension of the algebraic single-pushout approach to (typed) graph grammars was developed, where typing morphisms are compatible with the order relations defined over nodes and edges. The typing graphs, called class-model graphs, have partially ordered sets of nodes and edges, whose base structure follows the possible hierarchy of types when single inheritance in used.

The four core characteristics of the object-oriented paradigm for system development are contemplated in this work. *Encapsulation* of data and methods is achieved by structural constraints on the class-model graphs, which assure that an attribute belong to exactly one class, and a method can be received by exactly one object. *Information hiding* is obtained through restrictions on the rules used to model computations: an object which is receiving a message has access only to its attributes, although it can send messages to any object it has knowledge of (again, through its own attributes). *Inheritance* appears through the typing morphism, which allows an object to make use of any attribute or method defined along the hierarchy of types provided by the order relation on the class-model graph nodes. *Polymorphism* is implemented by the morphisms between object-oriented graphs, assuring that an object belonging to a class can be mapped to any object belonging to one of its derived classes. Additionally, dynamic binding occurs in the model of computation provided by those grammars, where the application of a rule which sends a message to an object has as its direct derivation a message typed according to the actual type of the object

mapped by the morphism, which is compatible with the way it is implemented in object-oriented programming languages.

Derivations are essentially the computations of a graph grammar, when regarded as a computational formalism. As pointed out by [8], since the pushout object of two arrows is unique up to isomorphism, the application of a production to a graph can produce an unbounded number of different results. This fact is highly counter-intuitive, because in the above situation one would expect a deterministic result, or, at most, a finite set of possible outcomes. Indeed, in the algebraic approach to graph grammars, one usually considers a concrete graph as a specific representation of a "system state", and since any kind of abstract semantics should be representation independent, one handles (more or less explicitly) *abstract graphs*, i.e., isomorphism classes of concrete graphs: with this choice, a direct derivation becomes clearly deterministic.

If the semantics we are interested in associates with each grammar all its possible derivations, we must reason also in terms of *abstract derivations*, i.e., equivalence classes of derivations with respect to a suitable equivalence. However, because of inheritance, polymorphism and method redefinition, what constitutes an equivalence class of object-oriented graphs, of an equivalence class of object-oriented systems derivations can differ considerably from other systems. We are presently investigating how those equivalence classes should be constructed, and how those results can be related to the ones already obtained for categories of concrete and abstract derivations.

References

1. Andries, M., Engels, G., Habel, A., Hoffmann, B., Kreowski, H.-J., Kuske, S., Plump, D., Schurr, A., and Taentzer, G. Graph transformation for specification and programming. *Science of Computer Programming 34* (1999), 1–54.
2. Blostein, D., Fahmy, H., and Grbavec, A. Practical use of graph rewriting. Tech. Rep. 95-373, Queen's University, Kingston, Ontario, Canada, 1995.
3. Brandenburg, F. J. On partially ordered graph grammars. In *3rd International Workshop on Graph Grammars and their Application to Computer Science* (Warrenton, Virginia, USA, 1986), H. Ehrig, M. Nagl, G. Rozenberg, and A. Rosenfeld, Eds., Lecture Notes in Computer Science 291, Springer-Verlag, pp. 99–111.
4. Cardelli, L., and Wegner, P. On understanding types, data abstraction and polymorphism. *ACM Computing Surveys 17*, 4 (1985), 471–522.
5. Cook, W. R., Hill, W. L., and Canning, P. S. Inheritance is not subtyping. In *POPL'90 - 17th Annual ACM Symposium on Principles of Programming Languages* (January 1990), Kluwer Academic Publishers.
6. Cook, W. R. *Object-oriented programming versus abstract data type*, vol. 489 of *Lecture Notes in Computer Science*. Springer, Berlin, 1990, pp. 151–178.
7. Corradini, A., Ehrig, H., Löwe, M., Montanari, U., and Padberg, J. The category of typed graph grammars and its adjunctions with categories of derivations. In *[12]* (1994), pp. 56–74.
8. Corradini, A., Ehrig, H., Löwe, M., Montanari, U., and Rossi, F. Abstract graph derivations in the double pushout approach. In *[24]* (1993), pp. 86–103.

9. Corradini, A., Montanari, U., and Rossi, F. Graph processes. *Fundamentae Informatica 26*, 3-4 (1996), 241–265.
10. Davey, B. A., and Priestley, H. A. *Introduction to Lattices and Order*, 2 ed. Cambridge University Press, Cambridge, 2002. 298p.
11. Dotti, F. L., and Ribeiro, L. Specification of mobile code using graph grammars. In *Formal Methods for Open Object-Based Distributed Systems IV* (2000), Kluwer Academic Publishers, pp. 45–64.
12. Ehrig, H., Engels, G., and Rozenberg, G., Eds. *5th International Workshop on Graph Grammars and their Application to Computer Science* (Williamsburg, 1994), Lecture Notes in Computer Science 1073, Springer-Verlag.
13. Ehrig, H., Heckel, R., Korff, M., Löwe, M., Ribeiro, L., Wagner, A., and Corradini, A. Algebraic approaches to graph transformation. Part II: single-pushout approach and comparison with double pushout approach. In *[14]*. ch. 4, pp. 247–312.
14. Ehrig, H., Kreowski, H.-J., Montanari, U., and Rozemberg, G. *Handbook of Graph Grammars and Computing by Graph Transformation*, vol. 1 (Foundations). World Scientific, Singapore, 1996.
15. Ehrig, H., Kreowski, H.-J., Montanari, U., and Rozemberg, G. *Handbook of Graph Grammars and Computing by Graph Transformation*, vol. 3 (Concurrency, Parallelism, and Distribution). World Scientific, Singapore, 1999.
16. Ehrig, H., and Löwe, M. Parallel and distributed derivations in the single-pushout approach. *Theoretical Computer Science 109* (1993), 123–143.
17. Ferreira, A. P. L., and Ribeiro, L. Towards object-oriented graphs and grammars. In *Proceedings of the 6th IFIP TC6/WG6.1 International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS 2003)* (Paris, November 19-21 2003), E. Najm, U. Nestmann, and P. Stevens, Eds., vol. 2884 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 16–31.
18. Ferreira, A. P. L., and Ribeiro, L. A graph-based semantics for object-oriented programming constructs. *Submitted for publication* (2004).
19. Korff, M. *Generalized Graph Structure Grammars with Applications to Concurrent Object-Oriented Systems*. PhD Thesis, Technische Universität Berlin, Berlin, 1995.
20. Löwe, M. *Extended Algebraic Graph Transformation*. PhD thesis, Technischen Universität Berlin, Berlin, Feb 1991.
21. Montanari, U., Pistore, M., and Rossi, F. Modeling concurrent, mobile and coordinated systems via graph transformations. In *[15]*. ch. 4, pp. 189–268.
22. Parisi-Presicce, F., Ehrig, H., and Montanari, U. Graph rewriting with unification and composition. In *3rd International Workshop on Graph Grammars and their Application to Computer Science* (Warrenton, Virginia, USA, 1986), H. Ehrig, M. Nagl, G. Rozenberg, and A. Rosenfeld, Eds., Lecture Notes in Computer Science 291, Springer-Verlag, pp. 496–514.
23. Ribeiro, L. *Parallel Composition and Unfolding Semantics of Graph Grammars*. PhD Thesis, Technische Universität Berlin, Berlin, June 1996. 202p.
24. Schneider, H. J., and Ehrig, H., Eds. *International Workshop on Graph Transforations in Computer Science* (Dagstuhl Castle, Germany, January 1993), Lecture Notes in Computer Science 1073, Springer-Verlag.
25. Taentzer, G. *Parallel and Distributed Graph Transformation Formal Description and Application to Communication-Based Systems*. PhD Thesis, TU Berlin, Berlin, 1996.

26. Troyer, O. D., and Janssen, R. On modularity for conceptual data models and the consequences for subtyping, inheritance and overriding. In *Proceedings of the 9th IEEE Conference on Data Engineering (ICDE 93)* (1993), IEEE CS Press, pp. 678–685.
27. Ungar, D., Chambers, C., Chang, B.-W., and Hölzle, U. Organizing programs without classes. *Lisp and Symbolic Computation 3*, 4 (1991).