Available online at www.sciencedirect.com
SCIENCE DIRECT®
Electronic Notes in Theoretical Computer Science
ELSEVIER Electronic Notes in Theoretical Computer Science 122 (2005) 89–104
www.elsevier.com/locate/entcs

A Graph-based Semantics For Object-oriented Programming Constructs

Ana Paula Lüdtke Ferreira[1]

Universidade do Vale do Rio dos Sinos – UNISINOS
São Leopoldo, RS, Brazil

Leila Ribeiro[2]

Universidade Federal do Rio Grande do Sul – UFRGS
Porto Alegre, RS, Brazil

Abstract

This paper presents a graph-based formalism for object-oriented class structure specifications. The formalism combines labelled graphs with partial orders, to adequately model the (single) inheritance relation among objects and the overriding relation between methods within derived classes. The semantics of system extension by inheritance and aggregation is then defined as colimits in a suitable category of object-oriented system specifications and their morphisms.

Keywords: Graph grammars, object-oriented programming.

1 Introduction

Object-orientation is perhaps the most popular paradigm of system development in use nowadays. The principles behind it — *data and code encapsulation*, *information hiding*, *inheritance* and *polymorphism* — fit very well into the needs of modular system development, distributed testing, and reuse of

[1] Email: anapaula@exatas.unisinos.br
[2] Email: leila@inf.ufrgs.br

doi:10.1016/j.entcs.2004.06.053

software faced by system designers and engineers. The variety of today's software applications, which range from traditional payroll systems to airplane control, e-mail clients and Internet browsers, also benefits from them.

There is a plethora of formal and semi-formal methods proposed in the literature for the specification of object-oriented systems, as well as a continuously growing number of object-oriented programming languages, whose suitability is usually determined by the application at hand. Our focus is to build a specification formalism for object-oriented systems which has the following characteristics: (i) it can be easily understood by both software developers and final users; (ii) systems' static and dynamic aspects can be specified in an integrated way; (iii) it has a semantical basis, allowing the composition of modular specifications in a consistent and significant manner; and (iv) high level specifications can be refined into lower ones, or even into actual programs.

Graphs can convey a significant amount of information in a compact, visual and, frequently, understandable way. The specification of computational systems using graphs offers two advantages, which rarely appear together in specification methods: (i) being formal mathematical structures, they have a well defined semantics and, (ii) having a diagrammatical layout, graph specifications can be more easily produced and understood by all participants in the software development process.

Graphs can provide a model of computation if combined with graph rules, to form a graph transformation system. The theory of graph grammars (graph transformation systems with an initial graph) studies a variety of formalisms which expand the theory of formal languages, to encompass more general structures specified as graphs [5]. The *algebraical approach to graph grammars*, presented for the first time in [7] makes use of categorical constructs to define the relevant aspects of the model of computation provided by graphs grammars. That approach is currently known as *double-pushout* approach, because derivations are based on two pushout constructions in the category of graphs and total graph morphisms. The *single-pushout* approach [13], on the other hand, has derivations characterized as a pushout construction in the category of graphs and partial graph morphisms. It is a proper extension of the double-pushout approach [4] capable of dealing with addition and deletion of items in unknown contexts, which is an important feature for distributed systems. 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 [6]. Concepts of parallel and distributed productions and derivations in the algebraic approach are very useful to model concurrent access, aspects of synchronization, and distributed

systems based on local and global graphs (see, for example, [8], [13], [6], [12], [17], [11], and [14]).

System specifications through graphs often rely on labelled graphs or typed graphs categories to represent different system entities. But labelling and typing do not reflect the inheritance relation among objects, and polymorphism cannot be applied if it is not made explicit. Subclass polymorphism [2] specifies that an instance of a subclass can appear wherever an instance of a superclass is required. So a class, if represented by a node or edge in a graph, should have a multiplicity of labels assigned to it or a typing morphism which could no longer be a function.

In [9] object-oriented graphs and grammars were first introduced, as graphs and graph productions typed over *type hierarchies*, which were graph structures very similar to the *class-model graphs* presented in this text. There, it was shown how graph grammar rules should be structured to reflect code encapsulation and information hiding, and how the semantics of object-oriented computations can be described by object-oriented graph grammars. In [10] it was shown how the semantics of dynamic binding can be viewed as a pushout on a suitable category. In this paper we improve our presentation of the fundamental typing structure and show how the two most common operations of object-oriented system extension, namely *object extension by inheritance* and *object creation by aggregation*, can be explained in terms of colimits on a category of system specifications and their morphisms. Since we want a specification formalism which can accurately reflect the way object-oriented systems are designed and programmed, we need their most common features formally explained into our framework.

This paper is structured as follows: Section 2 presents the order relations used to express the inheritance relation on objects and overriding relation between methods. Section 3 shows how those relations can be combined into a graph structure to represent an object-oriented class structure, and how this construction gives rise to a category whose objects are object-oriented system specifications and whose morphisms are relations between them. Section 4 presents how the semantics of common features of object-oriented languages, namely extension by inheritance, object aggregation and system composition can be defined in terms of special colimits of the category defined in the previous section. Most of the proofs of Sections 2, 3 and 4 are omitted because of space limitations, but they are really straightforward, and the details can be easily checked. Finally, Section 5 draws some conclusions about the work developed herein.

2 Strict relations

Inheritance, in the context of the object-orientation paradigm, is the construction which permits a class (in the class-based approach [3]) or an object (in the object-based approach [18]) to be specialized from an already existing one. This newly created entity carries (or "inherits") all the data and the actions belonging to its primitive one, in addition to its own data and actions. If this particularly class 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 definition of a *strict relation*, given below, formalizes what should be the fundamental object-oriented hierarchical structure of classes, when only single inheritance is allowed.

Definition 2.1 [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:

(i) if $(a, a') \in R$ then $a \neq a'$ (R has no reflexive pairs);

(ii) if $(a, a_1), (a_1, a_2), \ldots, (a_{n-1}, a_n), (a_n, a') \in R$, $n \geq 0$, then $(a', a) \notin R$ (R has no cycles);

(iii) 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 be created nor a method can be redefined in a circular or reflexive way.

If strict relations represent the hierarchical inheritance and method redefinition relations, it would be of interest to investigate some of its properties, which will be important in what follows. 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.

Remark 2.2 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 $a \sqsubseteq x$ 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 $\sqcup A$, can be defined dually.

Definition 2.3 [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.

Definition 2.4 [Strict ordered function] Let $\langle P, \sqsubseteq_P^* \rangle$ and $\langle Q, \sqsubseteq_Q^* \rangle$ be two strict ordered sets. A partial monotone function $f : P \to Q$ is a *strict ordered function* if and only if for all elements $x \in dom(f)$, we have that $\uparrow x \subseteq dom(f)$ and $f(\uparrow x) = \uparrow f(x) \cap \downarrow f(\sqcup \uparrow x)$.

The restrictions imposed to a strict ordered function are related to the mapping coherence between the strict ordered sets underlying relations. Specifically, if an element is mapped then *all* elements from the chain to which it belongs (respecting the strict relation on its set) must also be mapped accordingly. It can be shown that the upper set of any element is indeed a (finite) chain, and therefore has both a least and a greatest element. This restriction is needed to assure that the strict relation structure is maintained when the sets are combined. Before showing how this combinations can be performed, however, some properties of strict ordered functions will be shown, together with the proof that strict ordered sets and strict ordered functions constitute a category.

Theorem 2.5 (Category SOSet) *There is a category* **SOSet** *which has strict ordered sets as objects and strict ordered functions as arrows.*

Proof (sketch) Strict ordered sets and strict ordered functions are special kinds of, respectively, partially ordered sets and monotone functions. It can be easily shown that strict ordered functions are closed under composition, and that their composition is associative. Identities are build as in **POSET**. □

Theorem 2.6 (Colimits in SOSet) *The category* **SOSet** *is cocomplete.*

Proof (sketch) The initial object of **SOSet** is the empty strict ordered set $\langle \emptyset, \emptyset \rangle$. Coequalizers can be built in the same fashion they are built in **SetP**, the category which has sets as objects and partial functions as morphisms, which is cocomplete. The coequalizer $\langle C, c : B \to C \rangle$ of two arrows $f, g : A \to B$ in **SetP** is constructed by copying the elements which are not in the image of f and g and identifies the others in order to make $c \circ f = c \circ g$,

and the morphisms going into the object C are total and jointly surjective, which makes relatively easy to see that being f and g strict ordered functions, the functions going into the coequalizer's object must also be strict ordered functions. Since there exists an initial object and for each pair of arrows in **SOSet** there is a coequalizer, then the category has colimits. □

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 [1], 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 [15] 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.

3 Class-model graphs

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. 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.

The inheritance hierarchy is also portrayed, by imposing a strict relation (Definition 2.1) amongst the graph nodes. Hyperarcs also possess an order structure, which reflects the possibility of a derived object to redefine the methods inherited from their ancestors. This feature is used to define a formal semantics for dynamic binding based on graph computations [9], [10].

Such graph structure is called a *class-model graph*, and its formal definition

is given next.

Definition 3.1 [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 $tar(e) \in V$.

Order relations constraints: for all $e \in E$, the following holds:
(i) if $e \sqsubseteq_E e'$ then $lab(e) = lab(e') = \text{msg}$,
(ii) if $e \sqsubseteq_E e'$ then $src(e) = src(e')$,
(iii) if $e \sqsubseteq_E e'$ then $tar(e) \sqsubseteq_V^+ tar(e')$, and
(iv) if $e \sqsubseteq_E e'$ and $e'' \sqsubseteq_E e$, with $e' \neq e''$, then $(tar(e'), tar(e'')) \notin \sqsubseteq_V^*$ and $(tar(e''), tar(e')) \notin \sqsubseteq_V^*$.

The purpose of the relation between nodes, $\sqsubseteq_V$, is to establish an inheritance relation between objects. 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).

A classical example of a class structure for geometric shapes is portrait in Figure 1. Nodes denote objects (shape, round, circle, ellipse, Figure, Drawing, Color and Integer), while attributes and messages are represented by hyperarcs. The inheritance relation is represented by dotted arrows and the redefinition function is represented by solid thin ones.

Since class-model graphs are algebraic structures, morphisms between them can be defined. A class-model graph morphism formalizes the relationship between elements used by two different applications.

Definition 3.2 [Class-model graph morphism] Given two class-model graphs, $\mathcal{G}_1 = \langle V_{1\sqsubseteq}, E_{1\sqsubseteq}, L, src_1, tar_1, lab_1\rangle$ and $\mathcal{G}_2 = \langle V_{2\sqsubseteq}, E_{2\sqsubseteq}, L, src_2, tar_2, lab_2\rangle$, with $L = \{\text{attr}, \text{msg}\}$, the tuple $t = \langle t_V, t_E, id_L\rangle : \mathcal{G}_1 \to \mathcal{G}_2$ is a *class-model graph morphism* if and only if $t_V : V_1 \to V_2$ and $t_E : E_1 \to E_2$ are strict

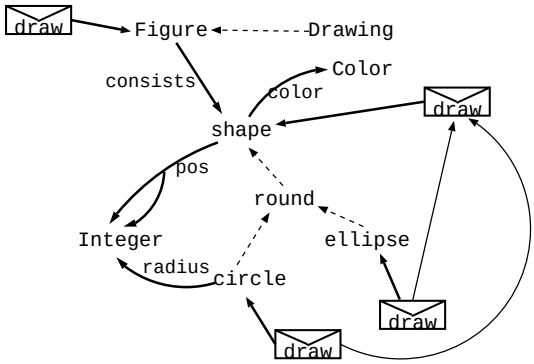

Figure 1. Class-model graph for geometric figures

ordered functions and t is a partial labelled hypergraph morphism.

A class-model graph morphism is a restricted labelled hypergraph morphism. The restrictions are related to the mapping coherence regarding the order relations on nodes and edges. Specifically, if a vertex (or arc) is mapped then *all* elements from the chain to which it belongs regarding to the inheritance (or redefinition) relation in the first graph must also be mapped accordingly. This restriction is needed to assure that single inheritance is maintained when the structures are combined. Before showing how this combinations can be performed, however, the following result must be stated:

Theorem 3.3 (Category CGraph) *There is a category* **CGraph** *which has class-model graphs as objects and class-model graph morphisms as arrows.* □

Characterizing objects, attributes and methods this way creates a situation where graphs are no longer defined over sets and functions, but over strict ordered sets and strict ordered functions. The very abstract way that graphs and graph morphisms are dealt with within category theory can be maintained by moving from diagrams into the category **SetP** of sets and partial functions to the category **SOSet** of strict ordered sets and strict ordered functions.

The algebraic approach to graph grammars rely on categorical constructs to express most of its results. Having graphs expressed in a category other than **Set** is useful, in the sense that if the constructs used within the 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. We end this section with the following result:

Theorem 3.4 (Colimits in CGraph) *The category* **CGraph** *is (finitely) cocomplete.*

Proof (sketch) Given two class-model graphs $\mathcal{G}_1 = \langle V_{1_\sqsubseteq}, E_{1_\sqsubseteq}, L, src_1, tar_1,$

$lab_1\rangle$ and $\mathcal{G}_2 = \langle V_{2\sqsubseteq}, E_{2\sqsubseteq}, L, src_2, tar_2, lab_2\rangle$ together with a class-model graph morphism $t = \langle t_V, t_E, id_L\rangle : \mathcal{G}_1 \to \mathcal{G}_2$, *the composition of* $\mathcal{G}_1$ *and* $\mathcal{G}_2$ *under* t, denoted by $\mathcal{G}_1 \diamond_t \mathcal{G}_2$, is the structure $\langle V_{C\sqsubseteq}, E_{C\sqsubseteq}, L, src_C, tar_C, lab_C\rangle$ (up to isomorphism) generated accordingly to the following steps:

(i) construct the colimit $\langle\langle C_V, \sqsubseteq_{C_V}\rangle, v_1 : V_1 \to C_V, v_2 : V_2 \to C_V\rangle$ in the category **SOSet** of the diagram containing objects $V_{1\sqsubseteq} = \langle V_1, \sqsubseteq_{V_1}\rangle$, $V_{2\sqsubseteq} = \langle V_2, \sqsubseteq_{V_2}\rangle$, and arrow $t_V : V_{1\sqsubseteq} \to V_{2\sqsubseteq}$;

(ii) construct the colimit $\langle\langle C_E, \sqsubseteq_{C_E}\rangle, e_1 : E_1 \to C_E, e_2 : E_2 \to C_E\rangle$ in the category **SOSet** of the diagram containing objects $E_{1\sqsubseteq} = \langle E_1, \sqsubseteq_{E_1}\rangle$, $E_{2\sqsubseteq} = \langle E_2, \sqsubseteq_{E_2}\rangle$, and arrow $t_E : E_{1\sqsubseteq} \to E_{2\sqsubseteq}$;

(iii) construct the colimit $\langle G, g_1 : \Phi(\mathcal{G}_1) \to G, g_2 : \Phi(\mathcal{G}_2) \to G\rangle$, with $G = \langle V_G, E_G, L_G, src_G, tar_G, lab_G\rangle$, in the category **LabHGraphP** of the objects $\Phi(\mathcal{G}_1)$, $\Phi(\mathcal{G}_2)$ and arrow $\Phi(t)$, where Φ is the forgetful functor which sends a class-model graph to a labelled hypergraph by eliminating the order structure on nodes and edges;

(iv) the colimits built in steps 1, 2 and 3 are constructed as in **SetP** (by definition). So, because they are colimits, sets C_V and V_C are isomorphic; the same is true for sets C_E and E_C. So, let $i_V : C_V \to V_G$ and $i_E : C_E \to E_G$ be the isomorphisms between those sets and $\sqsubseteq^*_{V_G}$ and $\sqsubseteq^*_{E_G}$ be the partial orders induced by, respectively, functions i_V and i_E, i.e., $\sqsubseteq^*_{V_G} = \{(i_V(v), i_V(v')) \mid (v, v') \in \sqsubseteq^*_{C_V}\}$ and $\sqsubseteq^*_{E_G} = \{(i_E(e), i_E(e')) \mid (e, e') \in \sqsubseteq^*_{C_E}\}$.

The colimit of $\mathcal{G}_1$ and $\mathcal{G}_2$ under t, is the class-model graph $G = \langle V_{G\sqsubseteq}, E_{G\sqsubseteq}, L, src_G, tar_G, lab_G\rangle$, where $V_{G\sqsubseteq} = \langle V_G, \sqsubseteq^*_{V_G}\rangle$ and $E_{G\sqsubseteq} = \langle E_G, \sqsubseteq^*_{E_G}\rangle$, together with arrows g_1 and g_2 when interpreted as class-model graph morphisms.

$\langle V_{G\sqsubseteq}, E_{G\sqsubseteq}, L, src_G, tar_G, lab_G\rangle$ is indeed a class-model graph, since by construction the edge and node sets are strict ordered sets, which maintain the order relation constraints. The structural constraints are guaranteed by the colimit construction in the category of labelled hypergraphs.

This proof can be completed by showing that, for any other class-model graph H and morphisms $h_1 : \mathcal{G}_1 \to H$ and $h_2 : \mathcal{G}_2 \to H$, such that $h_1(x) = (h_2 \circ t)(x)$ for all $x \in dom(t)$, there is a unique class-model graph morphism $u : G \to H$ such that $g_1 \circ h = h_1$ and $g_2 \circ h = h_2$. This part of the proof will not be done here, but it is easy to see that, since the graph part of the class-model graph structure as well as the order relation part are generated from colimits on the respective categories, the uniqueness of u derives from them. □

4 System extension

Software systems are generally built from previously constructed subsystems, which are later combined. The object-oriented paradigm favors that approach: existent objects can be aggregated or derived to form new ones. Composition is one of the most fundamental operations over systems, and it must be formalized in such a way that its result is compatible with the way systems are in fact combined. Modularity plays a key role in software development, allowing a complete specification to be constructed from different pieces of specifications. The need for integration tools is also a key issue in software development [16], since that task is considerably demanding in terms of effort if it is not automatized. Hence, specification formalisms should allow composition which can be performed systematically, guaranteeing a meaning for the operations in terms of the composed result. Composition of class-model graphs is described next.

Definition 4.1 [Class-model graph composition] Given two class-model graphs $\mathcal{G}_1$ and $\mathcal{G}_2$ together with a class-model graph morphism $t : \mathcal{G}_1 \to \mathcal{G}_2$, *the composition of* $\mathcal{G}_1$ *and* $\mathcal{G}_2$ *under* t is the colimit object of the diagram containing class-model graphs $\mathcal{G}_1$, $\mathcal{G}_2$, and arrow t in the category **CGraph**.

The class-model graph morphism t from Definition 4.1 represents a mapping between elements (objects, attributes or methods) which are considered to be *the same* in two different subsystems. It is fairly common, when programming a class, to make use of methods from objects defined elsewhere. It is not necessary to have knowledge of the complete specification of a class to use it as an attribute or to call on some of its methods. However, when the compiled files are linked together, the whole system must be fully specified. The morphism used to perform specifications composition plays the role of identify which elements are shared by both subsystems and which ones belong to just one of them. Since composition is given by a colimit, it is well defined and unique up to isomorphism.

Composition of class-model graphs can be used to perform different tasks other than plain system composition. Namely, specialization through inheritance and object aggregation, which are the most common way of augmenting a specification can be understood in terms of class-model graphs composition. This is important for it generates an uniform treatment on the ways systems are combined and augmented. So, all results applied to system's composition (as colimits in the category **SOSet**) are also applied to object creation by inheritance or aggregation.

Example 4.2 shows how specialization through inheritance can be defined as class-model graphs composition. Object construction by aggregation can

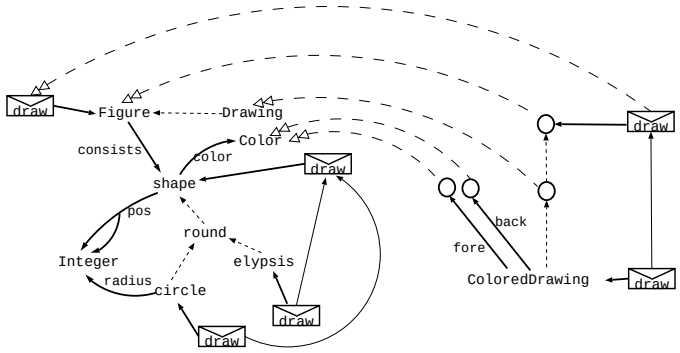

Figure 2. Specialization through inheritance as class-model graph composition

be achieved in a similar manner, as explained in Example 4.4.

Example 4.2 [Specialization through inheritance] The most common way of code reuse in the object-oriented paradigm is done through inheritance. This operation can be formalized by class-model graph composition (notice that the creation of a new object always alters a system specification, so it is coherent to formalize it by composition). To do so, it is necessary to create the specification of the new object (all attributes and messages included) and connect it to a chain of nodes as long as necessary. For instance, suppose we want to specialize an object of type *Drawing* from the class-model graph portrayed in Figure 1 to add to it a background and a foreground color. Besides these attributes, it should also redefine method *Draw*, which is defined at the level of object *Figure*. The resulting class-model graph, along with the class-model graph morphism (represented as dashed double arrows) which relates the corresponding elements on both class-model graphs is shown in Figure 2.

The resulting composite system is shown in Figure 3. Notice how the whole system structure was maintained, with the exception of the new added class *ColoredDrawing* which is derived from the class *Drawing*, as intended.

Notice that in order to construct a class-model graph to perform specialization through inheritance, there must be at least one node to which the new element must be connected (i.e., at least one primitive class). This particular node must be connected to the primitive object (the one to be derived) on the existing class-model graph. The number of such objects on the constructed hierarchy depends on the methods the derived object is intended to redefine: there must be as many objects on the node chain as there are elements between the primitive object and the one to which the method to be redefined belongs. The coherent mapping is achieved by assuring that it is a strict ordered function.

The process described in the Example 4.2 can be formally stated as follows.

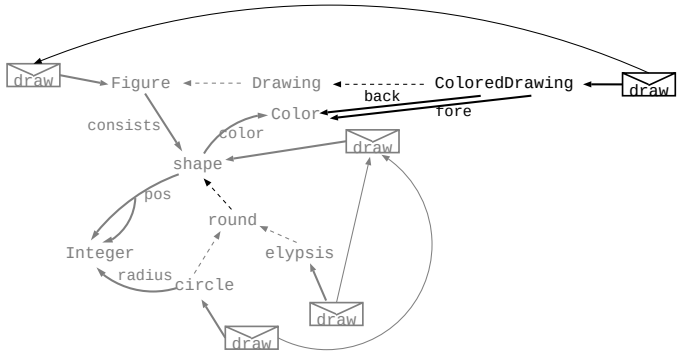

Figure 3. Specialization through inheritance as class-model graph composition

Definition 4.3 [Inheritance as class-model graph composition] Let $\mathcal{G} = \langle V_{\sqsubseteq}, E_{\sqsubseteq}, L, src, tar, lab\rangle$ be a class-model graph, $p \in V$ the object one wants to derive and $M = \{m_1, \ldots, m_n\}$, $n \geq 0$, a collection of messages to be redefined by the newly created class. Let $u = \sqcup\{tar(m) \mid m \in M\}$ if $M \neq \emptyset$ and $u = p$ otherwise. Now let U be a strict ordered set isomorphic to $\uparrow p \cap \downarrow u$ (with isomorphism ι_U).

Let $\mathcal{G}_O = \langle V_{O\sqsubseteq}, E_{O\sqsubseteq}, L, src_O, tar_O, lab_O\rangle$ be a class-model graph with the following characteristics:

- $V_O = U \cup \{o\} \cup A_o \cup A_m \cup A_r$, where o is called the *object-vertex*, $A_o = \{t^a_1, \ldots, t^a_k\}$ its set of attributes types, A_m is the set of the new messages parameter's types, A_r is the set of the to be redefined messages parameter's types which is isomorphic (with isomorphism ι_V) to the set $\{src(m) \mid m \in M\}$; $\sqsubseteq^*_V = \sqsubseteq^*_U \cup \{(o, u) \mid u \in U\}$;
- $E_O = M_N \cup M' \cup M_R \cup \overline{K}_a$, where M' is a set isomorphic to M (with isomorphism ι'_E), M_R is also isomorphic to M (with isomorphism ι_E) and contains the messages that will be redefined within the new object, M_N is a set of hyperarcs representing the methods belonging to the new object o, and $\overline{K}_a = \{1_a, 2_a, \ldots, k_a\}$ contains the new object attribute arcs; $lab_O(x_a) =$ attr, $src_O(x_a) = o$ and $tar_O(x_a) = t^a_x \in A_o$ for all $x_a \in \overline{K}_a$; $lab_O(x) =$ msg, $src_O(x) \subseteq A^*_m$ and $tar_O(x) = o$ for all $x \in M_N$; $lab_O(x) =$ msg, $src_O(x) \subseteq A^*_r$ and $tar_O(x) = o$ for all $x \in M_R$, with $(\iota_V \circ src_O)(x) = (src \circ i_E)(x)$; $\sqsubseteq_E = \{(m, m') \mid m \in M_R \wedge m' \in M' \wedge \iota_E(m) = \iota'_E(m')\}$, relating the messages which redefine and which are redefined within the new object;

Now, let t be a class-model graph morphism defined as follows:

- $dom(t_V) = V \setminus \{o\}$, where o is the object-vertex of O; $t_V(u) = \iota_U(u)$, for all $u \in U$, $t_V(a) = \iota_V(a)$, for all $a \in A_r$, and for all vertices $v \in (A_o \cup A_m)$, $t_V(v)$ is the mapping which connects the vertex v to its actual type in the

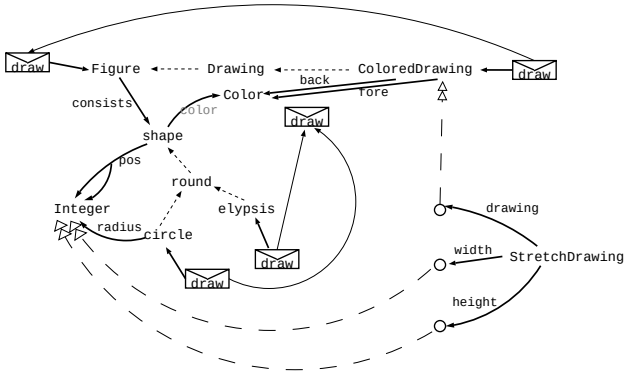

Figure 4. Aggregation as class-model graph composition

existing class-model graph $\mathcal{G}$;

- $dom(t_E) = M'$ which coincides with ι'_E, i.e., $t_E(m) = \iota'_E(m)$ for all $m \in M'$.

The object of the colimit given by the diagram containing objects $\mathcal{G}$ and $\mathcal{G}_O$, and morphism $t = \langle t_V, t_E, id_L \rangle$ as described above, correspond to the object-oriented system specification defined by $\mathcal{G}$ augmented with one object o which was formed by the derivation (via inheritance) of the existing object p in the class-model graph $\mathcal{G}$.

Example 4.4 [Aggregation] Aggregation is the operation used to combine two or more existent objects to construct a new one, allowing the new object to use all its constituent object functionalities in a transparent way. Given a class-model graph which contains the objects we want to aggregate, it is easy to augment it using the composition operation, in such a way that the resulting class-model graph contains the new object. For example, suppose we want an object called *StretchDrawing*, which embraces the functionality of the object *ColoredDrawing* (specialized from *Drawing* in the Example 4.2), plus two attributes of type *Integer*, called *width* and *height*. The class-model graph that should be constructed to aggregate existing objects belonging to another one has as constituents a node along with its attributes and messages, and which is connected to as many nodes as the objects one wants to aggregate. The morphisms should then identified the last ones with the actual objects to be aggregated. Figure 4 presents this new class-model graph, together with the morphism represented by dashed lines with double hollow triangles at their ends.

Figure 5 portrays the resulting class-model graph (generated as the colimit of the diagram presented in Figure 4), which contains a single new object with the required attributes.

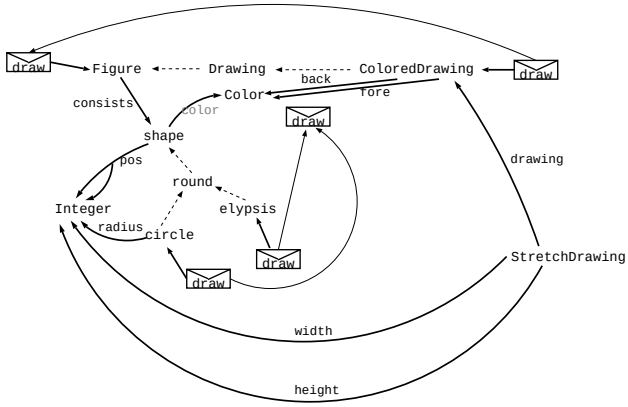

Figure 5. Aggregation as class-model graph composition

Definition 4.5 [Aggregation as class-model graph composition] Let $\mathcal{G} = \langle V_{\sqsubseteq}, E_{\sqsubseteq}, L, src, tar, lab\rangle$ be an arbitrary class-model graph and $\mathcal{G}_O = \langle V_{O\sqsubseteq}, E_{O\sqsubseteq}, L, src_O, tar_O, lab_O\rangle$ be a class-model graph where

- $V_O = \{o\} \cup A_o \cup A_m$ where o is the new object to be created, $A_o = \{t_1^a, \ldots, t_n^a\}$ its set of attributes types, $A_m = \{t_1^m, \ldots, t_l^m\}$ is the set of message parameter's types; $\sqsubseteq_{V_O} = \emptyset$;
- $E_O = \overline{N}_a \cup M_o$, where $\overline{N}_a = \{1_a, 2_a, \ldots, n_a\}$ and $M_o = \{m_1, \ldots, m_k\}$ is a set of k hyperarcs representing the methods belonging to the new object o; $lab_O(x_a) = \text{attr}$, $src_O(x_a) = o$ and $tar_O(x_a) = t_x^a \in A_o$ for all $x_a \in \overline{N}_a$; $lab_O(x) = \text{msg}$, $src_O(x) \subseteq A_m^*$ and $tar_O(x) = o$ for all $x \in M_o$; $\sqsubseteq_{E_O} = \emptyset$.

Let $t : \mathcal{G}_O \to \mathcal{G}$ be the following class-model graph morphism: $dom(t_V) = V_O \setminus \{o\}$, and for all $x \in (A_o \cup A_m)$, $t_V(x) = v$ for some $v \in V$ which reflects the actual type of the attribute x into $\mathcal{G}$; $t_E = \emptyset$.

The object of the colimit given by the diagram containing objects $\mathcal{G}$ and $\mathcal{G}_O$, and morphism t as described above, correspond to the object-oriented system specification defined by $\mathcal{G}$ aumengted with one object o which was formed by aggregation of existing objects in $\mathcal{G}$.

Although we have not proven herein that Definitions 4.3 and 4.5 are indeed correct, we believe that the central has been made clear. The next section closes the paper by making additional points on the relevance of those results.

5 Conclusions

A graph-based formalism for object-oriented class specification has been presented. The formalism, called class-model graph, is a labelled hypergraph whose node and hyperedge sets are equipped with a restricted partial order relation.

The relation on the nodes is used to make the (single) inheritance relation among objects explicit. The relation on edges is used to model method redefinition (overriding) within derived classes. Class-model graphs are meant to reflect more precisely the underlying structure of the object-oriented paradigm, and so improve the compactness and understandability of graph specifications.

Our main motivation is to use graphs as a mean of object-oriented system specification that are easy to produce and maintain, and which can be understood by all participants in the software development process, even if they are not experts in formal methods. The way classes in an object-oriented system can be specified with class-model graphs resembles the way they are created in most programming languages, such as C++ or Java. This is useful, in the sense that translations from one language to another can be made directly, and programmers do not need to worry about how a class can be defined in terms of nodes and edges.

Class-model graphs were built to provide a typing structure for graphs modelling object-oriented systems and their computations. A first step into this direction was done in [9]. However, if this typing structure is meant to reflect the way object-oriented systems are programmed, the most common features of programming must be also explained within this framework. We have described how new objects can be created by aggregating already specified objects as attributes or message parameters, or by extending an already existing object by inheritance. It was also shown how those constructions can be explained as colimits on the category **CGraph** of class-model graphs and their morphisms. Having the meaning of such constructions defined within category theory offers two major benefits: the first, and more obvious one, is that all knowledge available from research in category theory itself and in the specific categories of partial orders and graphs can be used; the second is that the results obtained, being colimits, are unique up to isomorphism, and can be defined without any ambiguity.

Class-model graphs are meant to define formally the static structure of object-oriented systems. Their dynamics can be described by graph rules, typed over a system's specific class-model graph. If different systems are put together, then their objects and rules (programs) should be meaningfully combined. This work is a first step towards a semantics of object-oriented program composition, when they are specified as an object-oriented graph grammar.

References

[1] Franz J. Brandenburg. On partially ordered graph grammars. In Hartmut Ehrig, Manfred Nagl, Grzegorz Rozenberg, and Azriel Rosenfeld, editors, *3rd International Workshop on Graph*

Grammars and their Application to Computer Science, Lecture Notes in Computer Science 291, pages 99–111, Warrenton, Virginia, USA, 1986. Springer-Verlag.

[2] Luca Cardelli and Peter Wegner. On understanding types, data abstraction and polymorphism. *ACM Computing Surveys*, 17(4):471–522, 1985.

[3] W. R. Cook. *Object-oriented programming versus abstract data type*, volume 489 of *Lecture Notes in Computer Science*, pages 151–178. Springer, Berlin, 1990.

[4] H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner, and A. Corradini. Algebraic approaches to graph transformation. Part II: single-pushout approach and comparison with double pushout approach. In *[5]*, chapter 4, pages 247–312.

[5] H. Ehrig, H-J. Kreowski, U. Montanari, and G. Rozemberg. *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 1 (Foundations). World Scientific, Singapore, 1996.

[6] H. Ehrig and M. Löwe. Parallel and distributed derivations in the single-pushout approach. *Theoretical Computer Science*, 109:123–143, 1993.

[7] H. Ehrig, M. Pfender, and H. J. Schneider. Graph grammars: an algebraic approach. In *14th Annual IEEE Symposium on Switching and Automata Theory*, pages 167–180, 1973.

[8] H. Ehrig and B.K. Rosen. Parallelism and concurrency of graph manipulations. *Theoretical Computer Science*, 11:247–275, 1980.

[9] Ana Paula Lüdtke Ferreira and Leila Ribeiro. Towards object-oriented graphs and grammars. In Elie Najm, Uwe Nestmann, and Perdita Stevens, editors, *Proceedings of the 6th International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS 2003)*, volume 2884 of *Lecture Notes in Computer Science*, pages 16–31, Paris, November 19-21 2003. Springer-Verlag.

[10] Ana Paula Lüdtke Ferreira and Leila Ribeiro. Derivations in object-oriented graph grammars. In *Proceedings of the 2nd International Conference on Graph Transformations (ICGT 2004), (to appear)*, Rome, September 28 - October 2 2004.

[11] Reiko Heckel. *Open Graph Transformation Systems: a New Approach to the Compositional Modelling of Concurrent and Reactive Systems*. PhD Thesis, Technische Universität Berlin, Berlin, 1998.

[12] Martin Korff. *Generalized Graph Structure Grammars with Applications to Concurrent Object-Oriented Systems*. PhD Thesis, Technische Universität Berlin, Berlin, 1995.

[13] Michael Löwe. *Extended Algebraic Graph Transformation*. PhD thesis, Technischen Universität Berlin, Berlin, Feb 1991.

[14] Ugo Montanari, Marco Pistore, and Francesca Rossi. Modeling concurrent, mobile and coordinated systems via graph transformations. In *[?]*, chapter 4, pages 189–268.

[15] Francesco Parisi-Presicce, Harmut Ehrig, and Ugo Montanari. Graph rewriting with unification and composition. In Hartmut Ehrig, Manfred Nagl, Grzegorz Rozenberg, and Azriel Rosenfeld, editors, *3rd International Workshop on Graph Grammars and their Application to Computer Science*, Lecture Notes in Computer Science 291, pages 496–514, Warrenton, Virginia, USA, 1986. Springer-Verlag.

[16] Thomas Reps. Algebraic properties of program integration. *Science of Computer Programming*, 17(1-3):139–215, 1991.

[17] Gabriele Taentzer. *Parallel and Distributed Graph Transformation Formal Description and Application to Communication-Based Systems*. PhD Thesis, TU Berlin, Berlin, 1996.

[18] David Ungar, Craig Chambers, Bay-Wei Chang, and Urs Hölzle. Organizing programs without classes. *Lisp and Symbolic Computation*, 3(4), 1991.