Towards Object-Oriented Graphs and Grammars

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

[1] Centro de Ciências Exatas e Tecnológicas, Universidade do Vale do Rio dos Sinos
anapaula@exatas.unisinos.br
[2] Instituto de Informática, Universidade Federal do Rio Grande do Sul
leila@inf.ufrgs.br

Abstract. This work aims to extend the algebraical approach to graph transformation to model object-oriented systems structures and computations. A graph grammar based formal framework for object-oriented system modeling is presented, incorporating both the static aspects of system modeling and the dynamic aspects of computation of object-oriented programs.

1 Introduction

Graphs are a very natural way of describing complex situations on an intuitive level. Graph-based formal description techniques are, for that reason, easily used by non-specialists on formal methods. Graph transformation rules can be brought into those descriptions in order to enrich them with a dynamical behaviour, by modeling the evolution of the structures represented as graphs.

The algebraic approach to graph grammars has been presented for the first time in [5] in order to generalize Chomsky grammars from strings to graphs. 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, on the other hand, has derivations characterized as a pushout construction in the category of graphs and partial graph morphisms. As presented in [4], [9] and [12], this approach is particularly adequate to model parallelism and distribution.

Graph grammars are appealing as a specification formalism because they are formal, they are based on simple yet powerful concepts to describe behaviour, they have a nice graphical layout which helps the understanding (even by non-specialists in formal methods) of a specification. Since graph grammars also provide a model of computation [7], they can serve as the basis for specifications which can be executed on a computer.

Different kinds of graph grammars have been proposed in the literature [11, 10, 9, 7, 1], aiming the solution of different problems. However, those focusing on object-oriented systems specification [3, 8] do not present a treatment on inheritance and polymorphism, which make object-oriented systems analysis and testing so difficult.

The use of the object-oriented paradigm has increased over the past years, becoming perhaps the most popular paradigm of system development in use

E. Najm, U. Nestmann, and P. Stevens (Eds.): FMOODS 2003, LNCS 2884, pp. 16–31, 2003.

nowadays. The growing use of Java as a language to support Internet applications has also contributed to this popularity. Object-based systems have a number of advantages over traditional ones, such as ease of specification, code reuse, modular development and implementation independence. However, they also present difficulties, derived from the very same features that allow the mentioned advantages.

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. It should be expected that formalisms for the specification of object-oriented architectures or programs reflect these particularities, otherwise the use of such formalisms will neglect key concepts that have a major influence in their organization. According to [6], a specification language for object-oriented conceptual modeling must at least include constructs for specifying primitive objects, particularizations of predefined objects, inheritance relationships between objects and aggregation of objects in order to define more complex objects. We also believe that the concepts of polymorphism and dynamic binding are essential if we intend to model static *and* dynamic aspects of object-oriented systems. So, in order to correctly model object-oriented systems, the key concepts related to it must be present within the formalism used.

This work aims to extend the algebraical approach to graph transformations to model object-oriented systems structures and computations. More accurately, the single pushout approach in the category of typed hypergraphs and partial typed hypergraph morphisms will be adapted to fit more adequately the object-oriented approach to software development. We will also show how the structures developed along the text are compatible with the notion of specification and computation within the object-oriented paradigm.

This paper is organized as follows: Section 2 presents a number of special kinds of (hyper)graphs, from where the main concepts of objects, attributes, methods and inheritance are developed. Section 3 presents the fundamental notions of object-oriented graph productions, grammars and derivations. Some examples of productions and its consequences on graph derivations are portrayed. Finally, Section 4 presents some conclusions from the work presented here.

2 Object-Oriented Graphs

The general definition of graphs is specialized to deal with the object-oriented aspects of program specification. This specialization is meant to reflect more precisely the underlying structure of the object-oriented paradigm, and so improve the compactness and understandability of specifications. Object-oriented systems consist of instances of previously defined classes (or objects[1]) which have an internal structure defined by attributes and communicate among themselves

[1] Object-based and class-based models are actually equivalent in terms of what they can represent and compute [13], so we will use the more commonly used class-based approach.

solely through message passing, so that approach underlies the structure of the graphs used to model those systems. Such structures are called *object-model graphs* and their formal definition is given next.

Definition 1 (Object-model graph). *An* object-model graph $\mathcal{M}$ *is a labelled hypergraph* $\langle V_{\mathcal{M}}, E_{\mathcal{M}}, L_{\mathcal{M}}, src_{\mathcal{M}}, tar_{\mathcal{M}}, lab_{\mathcal{M}}\rangle$ *where* $V_{\mathcal{M}}$ *is a finite set of vertices,* $E_{\mathcal{M}}$ *is a finite set of hyperarcs,* $L_{\mathcal{M}} = \{\text{attr}, \text{msg}\}$ *is the set of hyperarcs labels,* $src_{\mathcal{M}}, tar_{\mathcal{M}} : E_{\mathcal{M}} \to V^*_{\mathcal{M}}$ *are the hyperarcs source and target functions,* $lab_{\mathcal{M}} : E_{\mathcal{M}} \to L_{\mathcal{M}}$ *is the hyperarcs labelling function and, for all* $e \in E_{\mathcal{M}}$*, the following holds:*

- *if* $lab_{\mathcal{M}}(e) = \text{attr}$ *then* $src_{\mathcal{M}}(e) \in V_{\mathcal{M}}$ *and* $tar_{\mathcal{M}}(e) \in V^*_{\mathcal{M}}$*, and*
- *if* $lab_{\mathcal{M}}(e) = \text{msg}$ *then* $src_{\mathcal{M}}(e) \in V^*_{\mathcal{M}}$ *and* $tar_{\mathcal{M}}(e) \in V_{\mathcal{M}}$*.*

Sets $\{e \in E_{\mathcal{M}} \mid lab_{\mathcal{M}}(e) = \text{attr}\}$ *and* $\{e \in E_{\mathcal{M}} \mid lab_{\mathcal{M}}(e) = \text{msg}\}$ *are denoted by* $E_{\mathcal{M}}|_{\text{attr}}$ *and* $E_{\mathcal{M}}|_{\text{msg}}$*, respectively.*

Object-model graphs can also be viewed as a definition of the *classes* belonging to a system, where each node is a class identifier, hyperarcs departing from it correspond to its internal attributes, and messages addressed to it consist on 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.

A key feature of the object-oriented paradigm is the notion of *inheritance*. Inheritance is the construction which permits an object to be specialized from a pre-existing one. The newly created object carries all the functionality from its primitive object. This relation induces a hierarchical relationship among the objects from a system, which can be viewed as a set of trees (single inheritance) or as an acyclic graph (multiple inheritance). Both structures can be formally characterized as strict order relations, as follows.

Definition 2 (Strict order relation). *A binary relation* $R \subseteq A \times A$ *is said a* strict order relation *if and only if it has the following properties:*

1. *if* $(a, a') \in R$ *then* $a \neq a'$ *(R has no reflexive pairs);*
2. *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);*
3. *for any* $a, a', a'' \in A$*, if* $(a, a'), (a, a'') \in R$ *then* $a' = a''$ *(R is a function).*

Definition 3 (Type hierarchy). *Let* $\mathcal{M} = \langle V_{\mathcal{M}}, E_{\mathcal{M}}, L_{\mathcal{M}}, src_{\mathcal{M}}, tar_{\mathcal{M}}, lab_{\mathcal{M}}\rangle$ *be an object-model graph. A* type hierarchy *over* $\mathcal{M}$ *is a tuple* $\mathcal{H}_{\mathcal{M}} = \langle \mathcal{M}, isa, redef\rangle$*, where* $isa \subseteq V_{\mathcal{M}} \times V_{\mathcal{M}}$ *and* $redef \subseteq E_{\mathcal{M}} \times E_{\mathcal{M}}$ *are strict order relations*[2] *holding the following properties:*

1. *for each* $(e, e') \in redef$*,* $lab_{\mathcal{M}}(e) = lab_{\mathcal{M}}(e') = \text{msg}$*,*
2. *for each* $(e, e') \in redef$*,* $src_{\mathcal{M}}(e) = src_{\mathcal{M}}(e')$*,*

[2] For any binary relation $r \subseteq A \times A$, r^+ will denote its transitive closure and r^* will denote its reflexive and transitive closure.

3. *for each* $(e, e') \in redef$, $(tar_{\mathcal{M}}(e), tar_{\mathcal{M}}(e')) \in isa^+$, *and*
4. *for each* $(e', e), (e'', e) \in redef$, *if* $e' \neq e''$ *then* $(tar(e'), tar(e'')) \notin isa^*$ *and* $(tar(e''), tar(e')) \notin isa^*$.

The purpose of the relation *isa* is to establish an inheritance relation between objects. Notice that only single inheritance is allowed, since both *isa* and *redef* are required to be functions. Function *redef* establishes which methods will be redefined within the derived object, by mapping them. The restrictions applied to function *redef* ensure that methods are redefined consistently, i.e., only two message arcs can be mapped (1), their parameters are the same (2), the method being redefined is located somewhere (strictly) above in the type hierarchy (under isa^+) (3), and only the closest message with respect to relations *isa* and *redef* can be redefined (4).

Notice that the requirement concerning the acyclicity and the non reflexivity on *isa* and *redef* is consistent with the definition of classes in the object-oriented paradigm. A class is created as a specialization of at most one other class (single inheritance), which must exist prior to the creation of the new class.

Remark 1. Since type hierarchies are algebraic structures, operations over them can be defined. *Composition* (done with or without identification of elements on the structures being composed) plays an important role, since it corresponds to system composition. Although composition is an extremely relevant feature for system development, it is beyond the scope of this article.

Definition 4 (Hierarchy compatible strings). *Given a type hierarchy* $\mathcal{H}_{\mathcal{M}} = \langle \mathcal{M}, isa, redef \rangle$, *two node strings* $u, v \in V^*_{\mathcal{M}}$ *are* hierarchy compatible *if and only if* $|u| = |v|$ *and* $(u_i, v_i) \in isa^*$, $i = 1, \ldots, |u|$. *If* u *and* v *are hierarchy compatible we write* $u \propto_{\mathcal{H}} v$.

The definition of hierarchy compatible strings extends the relation isa^* to strings, which must have the same length, and the corresponding elements must be connected within that relation. It is easy to see that both isa^* and $\propto_{\mathcal{H}}$ are partial order relations.

Example 1. Fig. 1 presents a (naïve) type hierarchy for geometric shapes and figures. The nodes in the graph denote objects (shape, round, circle, ellipse, Figure, Drawing, Color and Integer), while object attributes and messages are represented by hyperarcs. The inheritance relation *isa* is represented by dotted arrows and the redefinition function *redef* is represented by solid thin ones.

Definition 5 (Hierarchical graph). *A hierarchical graph* $G^{\mathcal{H}}$ *is a tuple* $\langle G, t, \mathcal{H}_{\mathcal{M}} \rangle$, *where* $\mathcal{H}_{\mathcal{M}} = \langle \mathcal{M}, isa, redef \rangle$ *is a type hierarchy,* G *is a hypergraph, and* t *is a pair of total functions* $\langle t_V : V_G \to V_{\mathcal{M}}, t_E : E_G \to E_{\mathcal{M}} \rangle$ *such that* $(t_V \circ src_G) \propto_{\mathcal{H}} (src_{\mathcal{M}} \circ t_E)$, *and* $(t_V \circ tar_G) \propto_{\mathcal{H}} (tar_{\mathcal{M}} \circ t_E)$.

Hierarchical graphs are hypergraphs typed over an object-model graph which carries a hierarchical relation among its nodes. Notice that the typing morphism is slightly different from the traditional one [12]: a hierarchical graph arc can

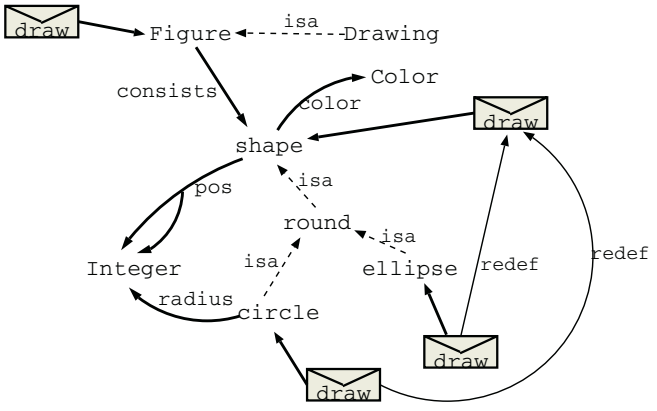


Fig. 1. Type hierarchy for geometric figures

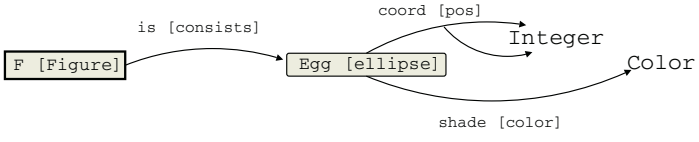


Fig. 2. Example of a hierarchical graph

be incident to any string of nodes which is hierarchy compatible to the one connected to its typing edge. 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.

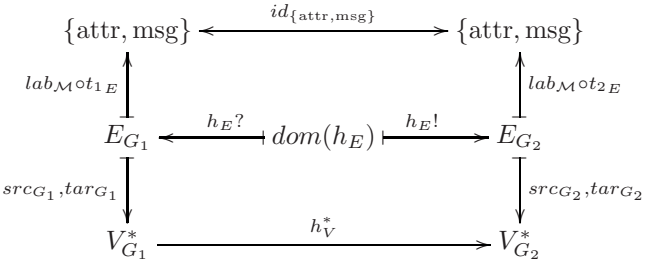
Example 2. Fig. 2 shows a hierarchical graph typed over the type hierarchy portrayed in Fig. 1. The typing morphism is revealed by the names between brackets. Notice that an ellipse has no attribute directly connected to it in the object-model graph. However, since an *ellipse* is a specialized *shape*, it inherits all its attributes.

Notice that all attributes belonging to the hierarchical graph on Fig. 2 are allowed by Definition 5. The referred graph has three edges, namely *is* (typed as *consists*), *coord* (typed as *pos*), and *shade* (typed as *color*). For *coord* we have (the same can be done to the other two):

$$(t_V \circ src)(coord) = ellipse \propto_{\mathcal{H}} shape = (src_{\mathcal{M}} \circ t_E)$$
$$(t_V \circ tar)(coord) = Integer\, Integer \propto_{\mathcal{H}} Integer\, Integer = (tar_{\mathcal{M}} \circ t_E)$$

Remark 2. For all diagrams presented in the rest of this text, $\mapsto$-arrows denote total morphisms whereas $\rightarrow$-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 $g!$.

Definition 6 (Hierarchical graph morphism). *Let $G_1^{\mathcal{H}} = \langle G_1, t_1, \mathcal{H}_{\mathcal{M}} \rangle$ and $G_2^{\mathcal{H}} = \langle G_2, t_2, \mathcal{H}_{\mathcal{M}} \rangle$ be two hierarchical graphs typed over the same type hierarchy $\mathcal{H}_{\mathcal{M}}$. A* hierarchical graph morphism *$h : G_1^{\mathcal{H}} \to G_2^{\mathcal{H}}$ between $G_1^{\mathcal{H}}$ and $G_2^{\mathcal{H}}$, 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**)*



commutes, for all elements $v \in dom(h_V)$, $(t_{2V} \circ h_V)(v) \propto_{\mathcal{H}} t_{1V}(v)$, and for all elements $e \in dom(h_E)$, $((t_{2E} \circ h_E)(e), t_{1E}(e)) \in redef^*$. 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 isa^*).

The meaning of the connection of two elements $x \dashrightarrow y$ by the relation *isa* is the usual: in any place that an object of type y is expected, an object of type[3] x can appear, since an object of type x is also an object of type y. A hierarchical graph morphism reflects this situation, since two nodes can be identified by the morphism as long as they are connected in the reflexive and transitive closure of *isa* within the type hierarchy. Similarly, two arcs can be identified by a hierarchical graph morphism if their types are related by the method redefinition relation. Since attribute arcs are only related (under $redef^*$) to themselves, two of them can only be identified if they have the *same* type in the underlying object-model graph. A message, however, can be identified with any other message which redefines it. The reason for this will be clear in Section 3.

It should be noticed that the arity of methods is preserved by the morphism, since two hyperarcs can only be mapped if they have the same number of parameters with compatible types.

[3] The word *type* is used here in a less strict sense than it is used in programming language design texts. Although the literature makes a difference on subtyping and inheritance relationships between objects [2], such differentiation will not be made here, since this work is being done in an upper level of abstraction. It is hoped that this will not cause any confusion to the reader.

Lemma 1. *Hierarchical graph morphisms are closed under composition.*

Proof. Hierarchical graph morphisms are componentwise composable. Given two hierarchical graph morphisms $f = \langle f_V, f_E \rangle : G_1^{\mathcal{H}} \rightarrow G_2^{\mathcal{H}}$ and $h = \langle h_V, h_E \rangle : G_2^{\mathcal{H}} \rightarrow G_3^{\mathcal{H}}$, the compound morphism $h \circ f = \langle h_V \circ f_V, h_E \circ f_E \rangle : G_1^{\mathcal{H}} \rightarrow G_3^{\mathcal{H}}$ exists, since morphisms on **SetP** are composable.
Additionally, for all elements $v \in dom(f_V)$, $(t_{2V} \circ f_V)(v) \propto_{\mathcal{H}} t_{1V}(v)$, and for all elements $v \in dom(h_V)$, $(t_{3V} \circ h_V)(v) \propto_{\mathcal{H}} t_{2V}(v)$ (f and g are both hierarchical graph morphisms). Then, for all $v_1 \in V_{G_1}$, $(t_{2V} \circ f_V)(v_1) \propto_{\mathcal{H}} t_{1V}(v_1)$, and for all $f_V(v_1) \in V_{G_2}$, $(t_{3V} \circ h_V \circ f_V)(v_1) \propto_{\mathcal{H}} (t_{2V} \circ f_V)(v_1)$. Since $\propto_{\mathcal{H}}$ is transitive, $(t_{3V} \circ h_V \circ f_V)(v_1) \propto_{\mathcal{H}} t_{1V}(v_1)$. Thus, $h \circ f$ is a hierarchical graph morphism. □

Lemma 2. *Composition of hierarchical graph morphisms is associative.*

Proof. Composition of hierarchical graph morphisms is done componentwise, and each of the components are functions. Since composition of partial functions and the transitivity of binary relations are associative, so is the composition of hierarchical graph morphisms. □

Proposition 1 (Category GraphP($\mathcal{H}_{\mathcal{M}}$)). *There is a category* **GraphP** $(\mathcal{H}_{\mathcal{M}})$ *which has hierarchical graphs over a type hierarchy* $\mathcal{H}_{\mathcal{M}}$ *as objects and hierarchical graph morphisms as arrows.*

Proof. Lemma 1 proves that the composition of two hierarchical graph morphisms is a hierarchical graph morphism. Lemma 2 states that composition of hierarchical graph morphisms is associative.

The identity morphism for a given hierarchical graph G is the trivial morphism $id_G = \langle id_V, id_E \rangle : G \rightarrow G$, where for any vertex $v \in V_G$, $id_{GV}(v) = v$, and for any edge $e \in E_G$, $id_{GE}(e) = e$. So, given any hierarchical graph morphism $h = \langle h_V, h_E \rangle : G_1^{\mathcal{H}} \rightarrow G_2^{\mathcal{H}}$, for any vertex $v \in V_{G_1}$, $(h_V \circ id_{G_1V})(v) = h_V(id_{G_1V}(v)) = h_V(v) = id_{G_2V}(h_V(v)) = (id_{G_2V} \circ h_V)(v)$. Similarly, for any edge $e \in E_{G_1}$, $(h_E \circ id_{G_1E})(e) = h_E(id_{G_1E}(e)) = h_E(e) = id_{G_2E}(h_E(e)) = (id_{G_2E} \circ h_E)(e)$.

The existence of identity, composition, and associativity of composition proves that $\mathbf{GraphP}(\mathcal{H}_{\mathcal{M}})$ is a category. □

Since $\mathbf{GraphP}(\mathcal{H}_{\mathcal{M}})$ is a category, the existence of categorical constructs within it can be investigated. However, the general definition of hierarchical graphs must be narrowed to correctly represent object-oriented systems. To achieve this goal, some additional functions and structures will be defined next.

Definition 7 (Attribute set function). *Given a hierarchical graph* $\langle G, t, \mathcal{H}_{\mathcal{M}} \rangle$*, where* $\langle \mathcal{M}, isa, redef \rangle$ *is a type hierarchy, let the* attribute set function $attr_G : V_G \rightarrow 2^{E_G}$ *return for each vertex* $v \in V_G$ *the set* $\{e \in E_G \mid src_G(e) = v \wedge lab_{\mathcal{M}}(t(e)) = \text{attr}\}$.

The extended attribute set function, $attr^*_{\mathcal{M}} : V_{\mathcal{M}} \rightarrow 2^{E_{\mathcal{M}}}$*, returns for each vertex* $v \in V_{\mathcal{M}}$ *the set* $\{e \in E_{\mathcal{M}} \mid lab_{\mathcal{M}}(e) = \text{attr} \wedge src_{\mathcal{M}}(e) = v' \wedge (v, v') \in isa^*\}$.

The *attribute set function* and the *extended attribute set function* will help define some other functions, relations and structures along this text. Basically, for any vertex v of a hierarchical graph, the attribute set function returns the set of all attribute arcs having v as their source. Similarly, given a type hierarchy, and a vertex v of its object-model graph, the extended attribute set function returns the set of all attribute arcs whose source is v or any other vertex to which v connected via isa^*.

Definition 8 (Message set function). *Given a hierarchical graph* $\langle G, t, \mathcal{H}_\mathcal{M}\rangle$, *where* $\langle \mathcal{M}, isa, redef\rangle$ *is a type hierarchy, let the* message set function $msg_G : V_G \to 2^{E_G}$ *returns for each vertex* $v \in V_G$ *the set* $\{e \in E_G \mid tar_G(e) = v \wedge lab_\mathcal{M}(t_E(e)) = \text{msg}\}$.

The extended message set function, $msg^*_\mathcal{M} : V_\mathcal{M} \to 2^{E_\mathcal{M}}$, *returns for each vertex* $v \in V_\mathcal{M}$ *the set* $\{e \in E_\mathcal{M}|_{\text{msg}} \mid tar_\mathcal{M}(e) = v' \wedge (v, v') \in isa^* \wedge \forall(e' \neq e) \in E_\mathcal{M}|_{\text{msg}}((v, tar(e')) \in isa^*, (tar(e'), v') \in isa^* \to (e', e) \notin redef^*)\}$.

The *message set function* returns all messages an object within an hierarchical hypergraph is currently receiving, while the *extended message set function* returns all messages an object of a specific type may receive. Notice that message redefinition within objects, expressed by the relation $redef^*$ on the type hierarchy, must be taken into account, since the redefinition of a class method implies that only the redefined method can be seen within the scope of a specialized class.

For any hierarchical graph $\langle G, t, \mathcal{H}_\mathcal{M}\rangle$ 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 $t^*_E|_{\text{msg}}$ and $t^*_E|_{\text{attr}}$ 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 9 (Object-oriented graph). *Let* $\mathcal{H}_\mathcal{M}$ *be a type hierarchy. A hierarchical hypergraph* $\langle G, t, \mathcal{H}_\mathcal{M}\rangle$ *is an* object-oriented graph *if and only if all squares in the diagram (in* ***Set****)*

$$\begin{array}{ccccc} 2^{E_G} & \xleftarrow{msg_G} & V_G & \xrightarrow{attr_G} & 2^{E_G} \\ \downarrow{\scriptstyle t^*_E|_{\text{msg}}} & & \downarrow{\scriptstyle t_V} & & \downarrow{\scriptstyle t^*_E|_{\text{attr}}} \\ 2^{E_\mathcal{M}} & \xleftarrow{msg^*_\mathcal{M}} & V_\mathcal{M} & \xrightarrow{attr^*_\mathcal{M}} & 2^{E_\mathcal{M}} \end{array}$$

commute. If, for each $v \in V_G$, *the function* $t^*_E|_{\text{attr}}(attr_G(v))$ *is injective,* $G^\mathcal{H}$ *is said a* strict *object-oriented graph. If* $t^*_E|_{\text{attr}}(attr_G(v))$ *is also surjective,* $G^\mathcal{H}$ *is called a* complete *object-oriented graph.*

It is important to realize what sort of message is allowed to target a vertex on an object-oriented graph. The left square on the diagram presented in Definition 9 ensures that an object can only have a message edge targeting it

if that message is typed over one of those returned by the extended message set function. It means that the only messages allowed are the least ones in the redefinition chain to which the typing message belongs. This is compatible with the notion of *dynamic binding*, since the method actually called by any object is determined by the actual object present at a certain point of the computation.

Object-oriented graphs can also be *strict* or *complete*. Strict graphs require that nodes do not possess two arcs typed as the same element on the underlying object-model graph. The requirement concerned the injectivity of t_E^* guarantees that there will be no such exceeding attribute connected to any vertex. For an object-oriented graph to be *complete*, however, it is also necessary that all attributes defined on all levels along the type hierarchy (via relation isa^*) are present. The definition of a complete object-oriented graph is coherent with the notion of inheritance within object-oriented framework, since an object inherits all attributes, and exactly those, from its primitive classes.

Object-oriented systems are often composed by a large number of objects, which can receive messages from other objects (including themselves) and react to the messages received. Object-oriented graphs also may have as many objects as desired, since the number and type of attributes (arcs) in each object (vertex) is limited, but the number and type of vertices in the graph representing the system is restricted only by the typing morphism.

Object-oriented graphs are just a special kind of hierarchical hypergraphs. It can be proved the existence of a subcategory of **GraphP**($\mathcal{H}_\mathcal{M}$), **OOGraphP** ($\mathcal{H}_\mathcal{M}$), which has object-oriented graphs as objects and hierarchical graph morphisms as arrows.

3 Object-Oriented Graph Grammars

Complete object-oriented graphs (Definition 9) can model an object-oriented system. However, in order to capture the system evolution through time, we need a graph grammar formalism to be introduced.

A *graph production*, or simply a *rule*, specifies how a system configuration may change. A rule has a *left-hand side* and a *right-hand side*, which are both strict object-oriented graphs, and a hierarchical graph morphism to determine what should be altered. Intuitively, a system configuration change occurs in the following way: all items belonging to the left-hand side must be present at the current state to allow the rule to be applied; all items mapped from the left to the right-hand side (via the graph morphism) will be preserved; all items not mapped will be deleted from the current state; and all items present in the right-hand side but not in the left-hand side will be added to the current state to obtain the next one.

Rule restrictions may vary, depending on what is intended for them to represent/implement. Unrestricted rules give rise to a very powerful system in terms of representation capabilities, but they also lead to many undecidable problems. Restrictions are needed not just to make interesting problems decidable (which is important *per se*) but also to reflect restrictions presented in the real system

we are modeling. All rule restrictions presented in this text are object-oriented programming principles, as described next.

First of all, 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 [4]. 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[4]), 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. This situation will be explored next, as different kinds of rules are defined.

Definition 10 (Basic object-oriented rule). *A* basic object-oriented rule *is a tuple* $\langle L^{\mathcal{H}}, r, R^{\mathcal{H}} \rangle$ *where* $L^{\mathcal{H}} = \langle L, t_L, \mathcal{H}_{\mathcal{M}} \rangle$ *and* $R^{\mathcal{H}} = \langle R, t_R, \mathcal{H}_{\mathcal{M}} \rangle$ *are strict object-oriented graphs and* $r = \langle r_V, r_E \rangle : L^{\mathcal{H}} \to R^{\mathcal{H}}$ *is a hierarchical graph morphism holding the following properties:*

- r_V *is injective and invertible,* r_E *is injective,*
- $\{v \in V_L | e \in E_L, src_L(e) = v, lab_{\mathcal{M}}(t_L(e)) = \text{attr}\}$ *is a singleton, whose unique element is called* attribute vertex*,*
- $\{e \in E_L | lab_{\mathcal{M}}(t_L(e)) = \text{msg}\}$ *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*, and*
- *for all* $v \in V_L$ *there is a bijection* $b : \{e \in E_L | src_L(e) = v, lab_{\mathcal{M}}(t_L(e)) = \text{attr}\} \leftrightarrow \{e \in E_R | src_R(e) = r_V(v), lab_{\mathcal{M}}(t_R(e)) = \text{attr}\}$*, such that* $t_R \circ b = t_L$ *and* $t_L \circ b^{-1} = t_R$*.*

[4] 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.

Different kinds of rules can be defined based on basic object-oriented rules. We define three of them: strict object-oriented rules (Definition 11) do not allow for object creation of deletion; object-oriented rules with creation (Definition 12) allow the creation of new objects; and general object-oriented rules (Definition 13) permit both creation and deletion operations.

Definition 11 (Strict object-oriented rule). *A* strict object-oriented rule *is a basic object-oriented rule* $\langle L^{\mathcal{H}}, r, R^{\mathcal{H}} \rangle$ *where the hierarchical graph morphism* $r = \langle r_V, r_E \rangle$ *is such that* r_V *is total and surjective.*

A strict object-oriented rule presents only the restrictions connected to the object-oriented programming paradigm, along with restrictions to assure that no object is ever created or deleted along the computation. This goal is achieved by requiring a bijection between the vertex sets.

Definition 12 (Object-oriented rule with object creation). *An* object-oriented rule with object creation *is a basic object-oriented rule* $\langle L^{\mathcal{H}}, r, R^{\mathcal{H}} \rangle$ *where* r_V *is total, and for all* $v \in V_R$*, if* $v \notin im(r_V)$ *the diagram*

$$\begin{array}{ccccc} 2^{E_R} & \xleftarrow{msg_R} & V_R & \xrightarrow{attr_R} & 2^{E_R} \\ \Big\downarrow{\scriptstyle t_E^*|_{\text{msg}}} & & \Big\downarrow{\scriptstyle t_V} & & \Big\updownarrow{\scriptstyle t_E^*|_{\text{attr}}} \\ 2^{E_{\mathcal{M}}} & \xleftarrow{msg^*_{\mathcal{M}}} & V_{\mathcal{M}} & \xrightarrow{attr^*_{\mathcal{M}}} & 2^{E_{\mathcal{M}}} \end{array}$$

commutes and t_E^* *is a bijection.*

Object-oriented rules with object creation differ from strict object-oriented rules in two aspects: r_V is not necessarily surjective, so new vertices can be added by the rule, and all created vertices must have exactly the attributes defined along its type hierarchy.

Definition 13 (General object-oriented rule). *A general object-oriented rule is a object-oriented rule with object creation* $\langle L^{\mathcal{H}}, r, R^{\mathcal{H}} \rangle$ *where* $dom(r_V) = V_L$ *or* $dom(r_V) = V_L \backslash \{\text{attribute vertex}\}$.

General object-oriented rules allow object deletion. Notice, however, that an object can only delete itself.

Different types of rules give rise to different possible computations. The more restrictive the rules, the more easier becomes to derive properties from system computations. Verification of computational power of rules is, however, beyond the scope of this paper.

Definition 14 (Object-oriented match). *Given a strict object-oriented graph* $G^{\mathcal{H}}$ *and an object-oriented rule* $\langle L^{\mathcal{H}}, r, R^{\mathcal{H}} \rangle$*, an* object-oriented match *between* $L^{\mathcal{H}}$ *and* $G^{\mathcal{H}}$ *is a hierarchical graph morphism* $m = \langle m_V, m_E \rangle : L^{\mathcal{H}} \to G^{\mathcal{H}}$ *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* $a, b \notin dom(r)$.

The role of a *match* is to detect a situation when a rule can be applied. It occurs whenever a rule's left-hand side is present somewhere within the system graph. Notice that distinct vertices can be identified by the matching morphism. This is sensible, since an object can point to itself through one of its attributes, or pass itself as a message parameter to another object. However, it would make no sense to identify different attributes or messages, so the edge component of the matching morphism is required to be injective. Additionally, preserved elements cannot be identified with deleted elements.

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 modelled in our work by a retyping message function.

Definition 15 (Retyping function *ret*). *Let* $\mathcal{H}_\mathcal{M} = \langle \mathcal{M}, isa, redef \rangle$ *be a type hierarchy and let* $G^\mathcal{H} = \langle G, t, \mathcal{H}_\mathcal{M} \rangle$ *be a hierarchical graph. The retyping function, when applied over a message hyperedge* $m \in E_G$*, i.e.,* $(lab_\mathcal{M} \circ t_E)(m) =$ msg*, and over a node* $v \in V_G$ *such that* $(t_V(v), (t_V \circ tar_G)(m)) \in isa^*$*, returns a hyperedge* m' *where* $src_G(m') = src_G(m)$*,* $tar_G(m') = v$ *e* $t_E(m') = e \in (msg^*_\mathcal{M} \circ t_V)(v)$*, such that* $(e, t_E(m)) \in redef^*$*.*

It can be shown that the retyping function is well defined (i.e., the set $(msg^*_\mathcal{M} \circ t_V)(v)$ is always a singleton). It can also be shown that the message passed to the function is retyped as the least type within the redefinition chain (with respect relation $redef^*$) to which that message belongs.

A *derivation step*, or simply a *derivation*, will represent a discrete system change in time, i.e., a rule application over an actual system specified as a graph.

Definition 16 (Object-oriented derivation). *Given a strict object-oriented graph* $G^\mathcal{H} = \langle G, t_G, \mathcal{H}_\mathcal{M} \rangle$*, an object-oriented rule* $\langle \langle L, t_L, \mathcal{H}_\mathcal{M} \rangle, r, \langle R, t_R, \mathcal{H}_\mathcal{M} \rangle \rangle$*, and an object-oriented match* $m : L^\mathcal{H} \to G^\mathcal{H}$*, their* object-oriented derivation *can be computed in two steps:*

1. *Construct the pushout of* $r : L \to R$ *and* $m : L \to G$ *in* ***GraphP****,* $\langle H, r' : G \to H, m' : R \to H \rangle$ *[4];*
2. *construct the strict object-oriented graph* $H^\mathcal{H} = \langle H, t_H, \mathcal{H}_\mathcal{M} \rangle$ *where, for each* $v \in V_H$*,* $t_H(v) = glb_{isa^*}(r'^{-1}(v) \cup m'^{-1}(v))$*, for each* $e \in E_H|_{\text{attr}}$*,* $t_H(e) = glb_{redef^*}(r'^{-1}(e) \cup m'^{-1}(e))$*, for each* $e \in E_H|_{\text{msg}}$*,* $t_H(e) = t_G(e)$ *if* $r'(x) = e$ *for some* $x \in E_G$*, or* $t_H(e) = ret(m'^{-1}(e), tar_H(e))$ *if* $m'(x) = e$ *for some* $x \in E_R$*.*

The tuple $\langle H^\mathcal{H}, r', m' \rangle$ *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. Element typing, needed to transform the resulting hypergraph into an object-oriented graph is done by getting the greatest lower bound (with respect the partial order relations isa^* and $redef^*$) of the elements

mapped by morphisms m' and r' to the same element. The basic object-oriented rule restriction concerning object types (which cannot be altered by the rule) assures that it always exist. Messages, however, need some extra care. Since graph L presents a single message, which is deleted by the rule application, a message on H comes from either G or R. If it comes from G, which is an object-oriented graph itself, no retyping is needed. However, if it comes from R, in order to assure that H is also an object-oriented graph, it must be retyped according to the type of the element it is targeting on the graph H.

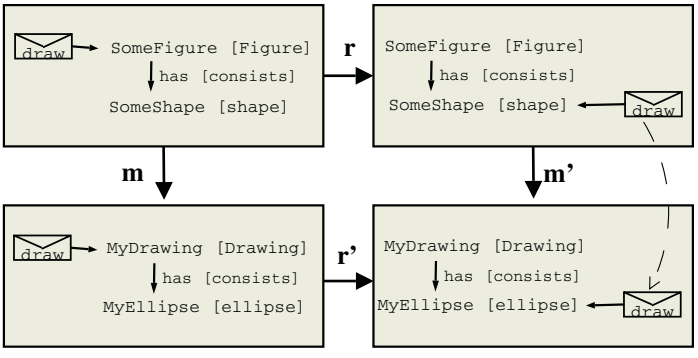


Fig. 3. Dynamic binding as message retyping

Figure 3 shows a situation where the need for a message retyping is made clear. As usual, the typing morphism is shown between brackets. Rule r portrays a common situation: the action resulting from a method calling is the calling of another method from one of the object's attributes. Here, a *figure* is drawn by making its constituent *shape* be drawn. However, since the rule's left-hand side is matched to a drawing which has an *ellipse* as constituent, and since the method *Draw* is redefined within that level, the resulting message cannot be typed as a *shape Draw*, but as an *ellipse Draw* (indicated by the only explicit arrow from R to H). Notice that m' is still a hierarchical graph morphism (although it is not strict). Hence, message retyping is the construction that implements dynamic binding on the computational process.

It can be shown that the an object-oriented derivation is a pushout structure in the category $\mathbf{OOGraphP}(\mathcal{H}_{\mathcal{M}})$.

Given the graph structures presented earlier and the rules to be applied to them, some interesting properties can be demonstrated. Closure properties are especially interesting, such as the ones expressed by the theorems below.

Theorem 1. *The class of complete object-oriented graphs is closed under object-oriented derivation using strict object-oriented rules.*

Proof. (sketch) Let $G^{\mathcal{H}}$ be a complete object-oriented graph, $\langle L^{\mathcal{H}}, r, R^{\mathcal{H}} \rangle$ be a strict object-oriented rule and $\langle L^{\mathcal{H}}, m, G^{\mathcal{H}} \rangle$ be an object-oriented match, and

$\langle L^{\mathcal{H}}, r', m' \rangle$ the resulting derivation of rule r at match m. Being r_V is a total bijection, for any $v \in V_L$ $t_L(v) = t_R(r_V(v))$ holds. Since m is a hierarchical graph morphism, for any vertex $v \in dom(r_V) \cap dom(m_V)$, $(t_G \circ m_V(v), t_R \circ r_V(v)) isa^*$, and so $t_H \circ r'_V \circ m_V(v) = t_H \circ m'_V \circ r_V(v) = t_G \circ m_V(v)$. So, V_H is isomorphic to V_G.

Now, let b be the bijection existing between the attribute edges from $A_L \subseteq E_L$ and $A_R \subseteq E_R$ defined as the last basic object-oriented rule restriction in Definition 10. Notice b is defined over *all* attribute edges of both graphs L and R. The match m between the rule's left-side and graph G is total on vertices and arcs, and injective on arcs, and by the characteristics of the pushout construction, function m' is also total and injective on arcs. Notice that all edges from G 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 (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 $2^B : 2^{E_G} \rightarrow 2^{E_H}$, and since the sets V_G and V_H are equal (up to isomorphism), it can be concluded that $H^{\mathcal{H}}$ is a complete object-oriented graph. □

Theorem 2. *The class of complete object-oriented graphs is closed under object-oriented derivation using object-oriented rules with object creation.*

Proof. (sketch) The same reasoning applied to the proof of Theorem 1 can be used to show that, in this case, V_G is isomorphic to a subset of V_H. The additional vertices of V_H are those created by the rule application (i.e., those isomorphic to the set $\{v \in V_R \mid v \notin im(r_V)\}$). But since all $v \notin im(r_V)$ is required to behave like a complete object-oriented graph when considered alone, so its inclusion on H will assure, along with Theorem 1, that it is also a complete object-oriented graph. □

Theorem 3. *The class of complete object-oriented graphs is not closed under object-oriented derivation using general object-oriented rules.*

Proof. (sketch) This theorem can be easily proven by a counterexample, since the deletion of a node causes the deletion on any of its incident arcs. The resulting graph will not be a complete object-oriented graph anymore. □

Theorem 3 describes a situation known as deletion in unknown contexts. This situation is very common in distributed systems, where the deletion of an object causes a number of dangling pointers to occur in the system as a whole. So, rules that allow object deletion can be used to find this kind of undesirable situations within a specification.

An interesting side effect derived from the use of rules that allow object deletion is that any dangling pointer would cause a edge cease to exist. In this case, any rule which takes that particular edge into consideration can no longer be applied (for no match can be found for that rule). When modeling system execution, this situation leads to the prevention of an execution runtime error, which would occur if an attempt to access an object which is no longer there is made.

Definition 17 (Object-oriented graph grammar). *An object-oriented graph grammar is a tuple* $\langle I^{\mathcal{H}}, P^{\mathcal{H}}, \mathcal{H}_{\mathcal{M}} \rangle$ *where* $I^{\mathcal{H}}$ *is a complete object-oriented graph,* $P^{\mathcal{H}}$ *is a finite set of object-oriented rules, and* $\mathcal{H}_{\mathcal{M}}$ *is a type hierarchy.*

Graph $I^{\mathcal{H}}$ portrays the initial system configuration. The system can evolve through the application of the productions in the grammar. All possible system future configurations are given by the set $\{G^{\mathcal{H}} \mid I^{\mathcal{H}} \Rightarrow^* G^{\mathcal{H}}\}$.

4 Conclusions

This work presented a first step towards a very high level and intuitive formal framework compatible with the main principles of object-oriented specification and programming. More specifically it presents, in terms of object-oriented graphs and morphisms, a way of defining classes, which can be primitive or specialized from others (though the *isa* relationship) together with a graph transformation-based model of computation compatible with polymorphism and dynamic binding, which are fundamental in the object-oriented programming model of execution.

A significant advantage to the use of a formal framework for object-oriented system specification is in the ability to apply rigorous inference rules so as to allow reasoning with the specifications and deriving conclusions on their properties. Fixing the sort of rules to be used within a graph grammar, properties regarding the computational model can be derived. Being this a formal framework, the semantics of operations (such as system and grammar composition) can also be derived.

Graph grammars are well suited for system specification, and object-oriented graph grammars, as presented in this text, fill the need for the key features of object-oriented systems be incorporated into a formal framework.

References

1. Marc Andries, Gregor Engels, Annegret Habel, Berthold Hoffmann, Hans-Jorg Kreowski, Sabine Kuske, Detlef Plump, Andy Schurr, and Gabriele Taentzer. Graph transformation for specification and programming. *Science of Computer Programming*, 34:1–54, 1999.
2. W. R. Cook, W. L. Hill, and P. S. Canning. Inheritance is not subtyping. In *POPL'90 - 17th Annual ACM Symposium on Principles of Programming Languages*. Kluwer Academic Publishers, 1990.
3. Fernando Luís Dotti and Leila Ribeiro. Specification of mobile code using graph grammars. In *Formal Methods for Open Object-Based Distributed Systems IV*. Kluwer Academic Publishers, 2000.
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 G. Rozemberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 1 – Foundations, chapter 4, pages 247–312. World Scientific, Singapore, 1996.

5. 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.
6. J. L. Fiadeiro, C. Sernadas, T. Maibaum, and G. Saake. Proof-theoretic semantics of object-oriented specification constructs. In W. Kent, R. Meersman, and S. Khosla, editors, *Object-Oriented Databases: Analysis, Design and Construction*, pages 243–284. North-Holland, 1991.
7. Annegret Habel. *Hyperedge Replacement: Grammars and Languages*, volume 643 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1992.
8. Aline Brum Loreto, Leila Ribeiro, and Laira Vieira Toscani. Decodability and tractability of a problem in object-based graph grammars. In *17th IFIP World Computer Congress - Theoretical Computer Science*, Montreal, 2002. Kluwer.
9. Michael Löwe. *Extended Algebraic Graph Transformation*. PhD thesis, Technischen Universität Berlin, Berlin, Feb 1991.
10. Ugo Montanari, Marco Pistore, and Francesca Rossi. Modeling concurrent, mobile and coordinated systems via graph transformations. In H. Ehrig, H-J. Kreowski, U. Montanari, and G. Rozemberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 3 – Concurrency, Parallelism and Distribution, chapter 4. World Scientific, 2000.
11. George A. Papadopoulos. Concurrent object-oriented programming using term graph rewriting techniques. *Information and Software Technology*, (38):539–547, 1996.
12. Leila Ribeiro. *Parallel Composition and Unfolding Semantics of Graph Grammars*. Phd thesis, Technische Universität Berlin, Berlin, June 1996. 202p.
13. David Ungar, Craig Chambers, Bay-Wei Chang, and Urs Hölzle. Organizing programs without classes. *Lisp and Symbolic Computation*, 3(4), 1991.