Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 338 (2018) 3–21

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Labelled Port Graph – A Formal Structure for Models and Computations

Maribel Fernández[1]

King's College London, UK

Hélène Kirchner[2]

Inria, France

Bruno Pinaud[3]

University of Bordeaux, CNRS UMR5800 LaBRI, France

Abstract

We present a general definition of labelled port graph that serves as a basis for the design of graph-based programming and modelling frameworks (syntax and semantics). We show that this structure provides the syntax for programs, which are composed of an initial graph, a set of rules and a strategy. Rules, represented as labelled port graphs, apply to states, also represented as labelled port graphs, and compute their successors according to the given strategy. The description of states, rules, and computations controlled by strategies, using labelled port graphs, is detailed and illustrated with examples from PORGY, a strategic port graph rewriting environment for the design of executable specifications of complex systems.

Keywords: labelled port graph, rewrite rule, strategy, rule-based modelling, PORGY

1 Introduction

Various notions of labelled graphs and attributed graphs can be found in the literature, but defining for these notions an appropriate theory of graph transformation is not trivial, mainly because labels or attributes are interpreted in an algebraic framework, not easily combined with the categorical framework usually used for graphs. In the last 20 years however different approaches have been proposed to overcome this difficulty: In [22], graphs are coded as algebras and the data algebra

[1] Email: maribel.fernandez@kcl.ac.uk

[2] Email: helene.kirchner@inria.fr

[3] Email: bruno.pinaud@u-bordeaux.fr

https://doi.org/10.1016/j.entcs.2018.10.002

is embedded in the graph. In [29], labelled graphs have labels which are elements of an algebra and graph transformation rules involve computations on the labels. In [25], symbolic graphs include variable nodes which represent the values of the attributes together with a set of formulas that constrain the value of these variables.

Port graphs, i.e., graphs where edges are attached to nodes at specific points called *ports*, have been used in various contexts. Ports permit to model in a natural way concepts such as binding or phosphorylation sites in protein interactions, communication ports in computer networks, or users' links in different social networks. Examples of port graphs used to model biochemical systems and to specify generation and propagation algorithms for social networks can be found in [3,2,11].

Port graphs with *attributes* associated with nodes, ports and edges, called attributed port graphs for short, are formally defined in [13], where a port graph rewriting relation is specified using the single pushout approach to graph transformation. This formal structure is implemented since 2010 in PORGY [26], a visual environment that allows users to define port graphs and port graph rewrite rules, and to apply the rewrite rules in an interactive way, or via the use of strategies. To control the application of rewrite rules, PORGY provides a *strategy language*. The latest version of PORGY can be downloaded from `http://porgy.labri.fr` either as source code or binaries for MacOS, Windows or Linux machines.

Building on these previous works, this paper focuses on the formal structure of *labelled port graph* and explains its role in the design of executable specifications of complex systems. Labelled port graphs are port graphs where ports, nodes and edges carry a label, which is an expression from a formal language given as a parameter together with its interpretation domain. The notion of attribute used in PORGY to define properties of nodes, ports and edges can be seen as a particular case of label, where a set of built-in function symbols and predicates are available (for example, arithmetic operators, and predicates interpreted over a graph domain, such as $edge(n, n')$, which holds if there exists an edge linking the nodes n and n' in a given graph).

We show that all the ingredients of a graph transformation system can be specified as labelled port graphs. Port graph rewrite rules are labelled port graphs consisting of two port graphs (the left- and right-hand sides) and a special node (the *arrow* node) that links ports from the left-hand side and right-hand side. Formally, the arrow node defines a morphism which is used to give a single pushout semantics for rewriting. Since there is usually more than one way to apply rules to a graph to generate rewriting steps, a strategy expression is used to select the rule to be applied and the position in the graph where rules should (or not) apply. We define *located graphs* as labelled port graphs that include labels to specify the rewriting position and the subgraphs that should be protected (i.e., not rewritten). Rewriting derivations, controlled by a strategy, can also be represented as a labelled port graph, whose nodes are labelled by graphs and strategies.

This approach to graph rewriting, where all the components of the system are represented using a unique concept (namely, labelled port graphs), has advantages both from a theoretical and a practical point of view: there is one main data struc-

ture to design and implement, so the implementation efforts can focus on this task, and since the rewrite rules are themselves graphs, the formalism is by construction reflective. Reflection is a key property in logical frameworks and facilitates the design of extensions (see, e.g., rewriting logic [23]).

The paper is organised as follows: Section 2 introduces the concept of labelled port graph as a generic structure, and presents the specific instance used in PORGY. Section 3 illustrates this structure with examples taken from various domains. Rules and rewriting are defined in Section 4, where it is shown that rules are also labelled port graphs and the rewriting mechanism on this structure is presented. Section 5 presents located port graphs as labelled port graphs. Derivation graphs, another instance of labelled graphs, are defined in Section 6 and used as a mechanism to visualise the dynamic behaviour of systems modelled by means of labelled port graphs, rewrite rules and strategies. Section 7 describes related works and gives directions for future work.

2 Definition of labelled port graphs

A port graph is a graph where nodes have ports, which are the points where edges are attached. Nodes, ports and edges are labelled.

In this paper, we propose the notion of symbolic label that is a parameter of the labelled port graph definition.

Definition 2.1 [Symbolic Label] A symbolic label (or just label for short) ll is an element of a formal language $\mathscr{L}$ given by its syntax and semantics.

All labels have a name, are built using $\mathscr{L}$'s syntax and are interpreted in the given semantic domain.

Definition 2.2 [Labelled port graph] A *labelled port graph* $G = (V, P, E, D)_{\mathcal{F}}$ over $\mathscr{L}$ is given by

- a 4-tuple (V, P, E, D) of pairwise disjoint sets, where:
  - V is a finite set of nodes; $n, n_1, \ldots$ range over nodes;
  - P is a finite set of ports; $p, p_1, \ldots$ range over ports;
  - E is a finite set of edges between ports; $e, e_1, \ldots$ range over edges; two ports may be connected by more than one edge;
  - D is a set of labels from $\mathscr{L}$;
- and a 3-tuple $\mathcal{F}$ of functions *Connect*, *Attach* and *Label* such that:
  - for each edge $e \in E$, $Connect(e)$ is the pair (p_1, p_2) of ports connected by e;
  - for each port $p \in P$, $Attach(p)$ is the node n to which the port belongs;
  - $Label : V \cup P \cup E \mapsto \mathcal{P}(D)$ is a labelling function that returns a finite set of labels for each element in $V \cup P \cup E$.

If edges are not oriented, the order of the ports in the result of *Connect* can be ignored.

Example 2.3 An example of labelled port graph is given in Figure 1. V is a set of

11 nodes, each of them labelled by a molecule name (Raf-1, PDE8A1, AKAP, PKA, cAMP or S). P is a set of 21 ports, also labelled by their names (S1, S2,P1, ...) and the *Attach* function is visualised by drawing the port p as a red or green square inside the node n when $Attach(p) = n$. Each node and port has also a Colour label visualised by the colour of the element on the figure. E is a set of 7 edges, which are not labelled in this example. The *Connect* function is visualised by a line between two ports (p_1, p_2) if $Connect(e) = (p_1, p_2)$.

Definition 2.2 is generic in the sense that the set D of labels is actually a parameter that can be instantiated in different ways. If D is empty, we obtain plain (unlabelled) port graphs, consisting only of sets of nodes with ports and edges connecting nodes via ports; if D is a set of atomic labels, we obtain the notion of labelled port graphs defined in [1,2,12]. Richer definitions of labels have been proposed for graphs, which bring much more expressivity: for instance, in [25], the labels of E-graphs are represented by label nodes connected to edges and nodes, while in symbolic graphs, they are variables constrained by a set of formulas and interpreted over an algebra; in [9], attributed graph labels are the values of a given data algebra. Below we assume familiarity with basic notions of universal algebra (we refer the reader to [10] for details).

Labels are implemented in PORGY as records, whose fields have values interpreted in algebras, as described in [13]. In PORGY, a label ll is a list of pairs $(a_1 := v_1, \ldots, a_n := v_n)$, where a_i, called *attribute*, is a constant in a set $\mathcal{A}$ or a variable in a set $\mathcal{X}_\mathcal{A}$, and v_i is the value of a_i. The elements a_i are pairwise distinct. The first attribute in each label ll is its name and identifies the type of the label in the following sense: for all $ll = (Name := v_1, \ldots, a_n := v_n)$, $ll' = (Name := v'_1, \ldots, a'_m := v'_m)$, if $v_1 = v'_1$, then $n = m$ and $a_k = a'_k$ for any $1 < k \leq n$. An example of record label is given in Figure 2.

Note that according to Definition 2.1, symbolic labels may be variables, first-order terms and formulas involving variables. For instance in PORGY, records may contain variables as values of attributes, and variables can be used to denote generic attributes or generic records in port graph rewrite rules. These variables may be instantiated in the given semantic domain. More precisely, values in PORGY's labels are either concrete values (numbers, Booleans, strings, etc.), or symbolic terms built on a signature $\Sigma = (\mathcal{S}, \mathcal{O}p)$ of an abstract data type and a set $\mathcal{X}_\mathcal{S}$ of variables of sorts $\mathcal{S}$. We denote by $T(\Sigma, \mathcal{X}_S)$ the set of terms over Σ and $\mathcal{X}_S$. We use a set $\mathcal{P}red$ of predicates involving equality, disequality and ordering on numerical values.

Definition 2.4 [Ground and symbolic labelled port graph] Given a formal language $\mathscr{L}$ defined by a signature $\Sigma = (\mathcal{S}, \mathcal{O}p, \mathcal{P}red)$ and a set $\mathcal{X}_\mathcal{S}$ of variables of sorts $\mathcal{S}$, we call *symbolic port graph* a labelled port graph whose labels contain variables, and *ground port graph* a labelled port graph without variables. Sorted variables are instantiated in the sorted semantic domain associated to $\mathscr{L}$.

In PORGY, labels with abstract values (i.e., expressions $v_i \in T(\Sigma, \mathcal{X}_S)$ that may contain variables), allow us to define generic patterns in rewrite rules: abstract values in left-hand sides of rewrite rules are matched against concrete data in the

graphs to be rewritten.

Below we may refer to labelled port graphs simply as port graphs.

3 Domains

Labelled Port Graphs have been used to describe complex models in various domains: biology, social networks, interaction nets, capital markets, etc. We give examples below, including visual representations of system states (port graphs), obtained using PORGY. These examples illustrate the kind of labels supported in the current version of PORGY.

3.1 A biochemical process

Molecular species are represented by labelled port graphs in a natural way: each molecule is represented by a node whose ports correspond to its binding or, for instance, phosphorylation sites. The attribute *Name* in each node identifies the type of species represented by the node. Attributes *Colour* and *Shape* have the same value for all nodes with the same *Name*. Ports have an attribute *State* indicating whether it is bound or phosphorylated and a related attribute *Colour* that reflects the value of the state.

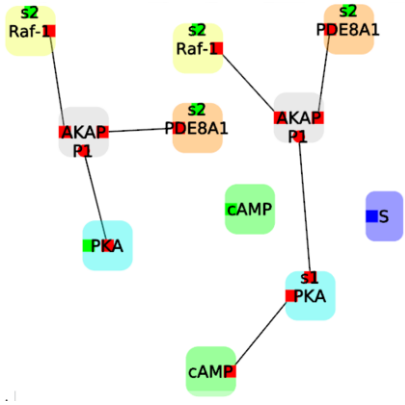


Fig. 1. Example of port graph for a biochemical process

Figure 1 shows an example of a port graph used in a biological case study [2] with two groups of complex molecules connected by many edges, and two simpler molecule (two green "cAMPs" and one purple "A"). As explained earlier, labels are used in the graphical interface to improve the visualisation of the graph. Each node is shown with its *Name* and the ports attached to it are displayed inside; the values of the attributes *Colour* and *Shape* are taken into account when displaying the node.

3.2 Social networks

A social network is usually described as a graph where nodes represent users and edges represent their relationships. Some real-world social relations involve mu-

tual recognition (e.g., friendship), whereas others present an asymmetric model of acknowledgement (e.g., follower/followee). In a first approach, nodes representing users have only one port gathering directed connections and edges are directed. Multiple ports are useful, either to connect users according to the nature of their relation (e.g., friend, parent, co-worker, ...) or to model situations where a user is connected to friends via different social networks.

We present in Figure 2 an example of port graph studied in [14]. In this example, nodes have attributes *State*, and *Tau* (used in an information propagation algorithm to handle influence of users on each others), as well as an attribute *Colour*, for visual purposes. The attribute *Name* is not mentioned because it has the same value for all nodes (*Name* := `user`) and edges (*Name* := `follower`). An edge attribute *Marked* is used to control iteration.

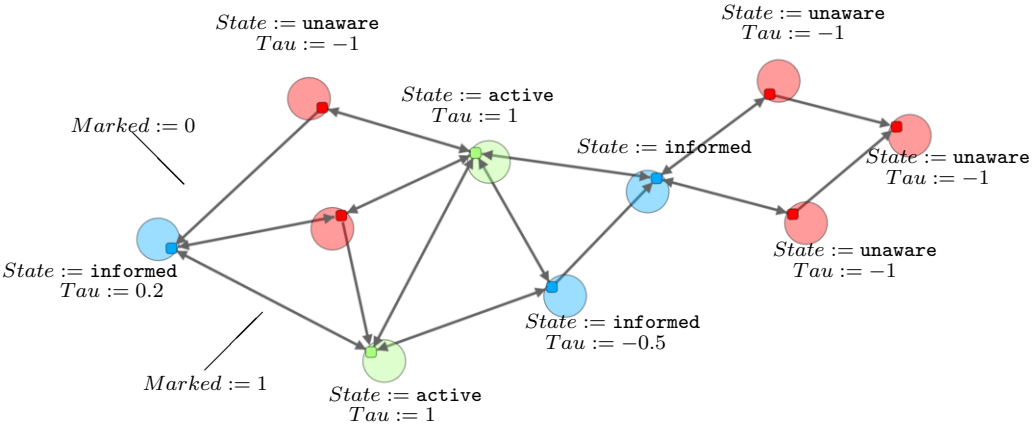


Fig. 2. Example of port graph for a toy social network with some attributes.

3.3 Interaction nets

Interaction nets [21] are a graphical model of computation inspired by proof nets from Linear Logic. In interaction net systems, programs consist of a net and a set of interaction rules describing the possible interactions between agents. Nets are graphs where nodes represent agents. Each agent has one principal port, where interaction can take place, and a (possibly empty) set of auxiliary ports. Edges connect agents by linking their ports. Interaction can only take place when two agents are connected via their principal ports. In that case, if an interaction rule for that pair of agents is provided, the pair of agents is replaced by the right-hand side of the rule. In interaction rules, the left-hand side is restricted to a pair of agents, and the right-hand side is a net with exactly the same number of free ports as the left-hand side. Despite these restrictions, interaction nets are a universal model of computation. Moreover, they are particularly suitable to analyse the cost of computation, since the restrictions imposed on interaction rules ensure that all steps of computation are represented as interaction steps.

Figure 3 shows an interaction net represented in PORGY. We use labels to identify the name of the agent, and the kind of port (*P* for the principal port,

numbers for auxiliary ports).

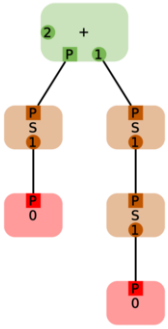


Fig. 3. Example of an interaction net representing the arrithmetic expression $1 + 2$.

4 Rules

To specify the dynamic behaviour of a modelled system, graph transformation rules are a useful tool. In this section, we first give the general definition of port graph rewrite rules with symbolic labels, which we illustrate using PORGY's rewrite rules, and then define the rewriting process, based on the notion of matching morphism.

4.1 Labelled port graph rewrite rule

A *port graph rewrite rule* is a labelled port graph consisting of two subgraphs L and R together with an *arrow* node that links them. Each rule is characterised by its arrow node, which delimitates left and right-hand sides of the rule; the arrow node has a name, labels that express conditions restricting the rule's matching, and ports to control the rewiring operations at rewriting time.

Definition 4.1 [Labelled port graph rewrite rule] A *labelled port graph rewrite rule* is a labelled port graph consisting of:

- two disjoint labelled port graphs L and R, called *left-hand side* and *right-hand side*, respectively, such that all variables in R occur in L;
- an *arrow* node with a set of *rewiring ports* and a set of edges that each connect a port of the arrow node to ports in L or R. Each port has a label $Type$ that can have one of three different values: *bridge*, *wire* and *blackhole*. The value indicates how a rewriting step using this rule should affect the edges that connect the redex to the rest of the graph.
  (1) A port of type *bridge* must have edges connecting it to L and to R (one edge to L and one or more to R): it thus connects a port from L to ports in R.
  (2) A port of type *wire* must have exactly two edges connecting to L and no edge connecting to R.
  (3) A port of type *blackhole* must have edges connecting it only to L (one edge or more).

In addition, the arrow node has two ports, labelled *Left* and *Right*, and edges connecting *Left* to all the nodes in L and *Right* to all the nodes in R, using a distinguished port *Side* in nodes of L and R. The arrow node also has the following predefined labels: a label $Where$ that expresses a condition to trigger rule application, and *Saturated*, whose value is the list of ports in L that are not connected to a bridge, wire or blackhole port of the arrow node. These two labels are used to select appropriate matching morphisms and to ensure non-dangling conditions in the rewriting process.

Example 4.2 Figure 5 shows a rewrite rule in PORGY. In the visual representation of the rule, the ports *Left* and *Right* in the arrow node and their associated edges and ports *Side* in nodes of L and R are omitted, since this information is conveyed by the position of the subgraphs L and R in the diagram, respectively to the left and right-hand side of the arrow node. In this example, the arrow node has three rewiring ports of type bridge, and the six associated edges are depicted in red (in PORGY the user can choose whether to display these edges or not, for example in Figure 4 some red edges have been omitted).

The *Saturated* attribute in the arrow node lists the ports in L not linked by an edge to a rewiring port of the arrow node; for the rule in Figure 5, this list is empty, since every port in L is connected to a bridge port in the arrow node.

The $Where$ attribute in port graph rewrite rules is an optional user-defined Boolean expression involving elements of L (edges, nodes, ports and their attributes). Such a condition may be used to specify the absence of specific edges. For instance, a condition $Where := notEdge(p, p')$ requires that no edge exists between the images of the ports p and p'. For the rule in Figure 5, this condition is represented by a crossed edge between the ports in nodes A and C.

Note that the labels may involve variables, functions or predicates that are interpreted in the port graph structure. This is the case for instance for the Boolean expression $notEdge(p, p')$ where p, p' are variables of a ground port graph and $Edge$ a predicate interpreted as the existence of an edge between these two ports.

4.2 Labelled port graph rewriting

Now, applying a rule to a ground labelled port graph requires first to find a port graph morphism between the rule's left-hand side and a subgraph of the target graph.

If L and G are two port graphs, a *port graph morphism* $f : L \mapsto G$ maps nodes, ports and edges of L to those of G such that the attachment of ports to nodes and the edge connections are preserved, as well as the labels. A detailed definition is given in [13]. We just explain the intuition below.

A (partial) *morphism* $f : L \mapsto G$ from L to G is a family of (partial) functions f_V, f_P, f_E, f_D such that: f_V, f_P, f_E are injective (the morphism does not identify distinct nodes, ports or edges), and preserve the edge connections and the port attachments; f_D may instantiate variables in labels and for any label ll, $f_D(ll)$ must be valid in the interpretation domain $\mathcal{D}$.

For instance, when a label ll is a formula, all its free variables are instantiated by f_D and the formula is interpreted in the domain $\mathcal{D}$. The port graph morphism exists only when $f_D(ll)$ is a valid formula in the domain $\mathcal{D}$.

This definition ensures that L and $f(L)$ have the same port graph structure, and each corresponding pair of nodes, ports and edges in L and G have the same set of labels (attributes and associated values in PORGY), except at positions where there are variables. When using this definition to define rewriting below, L will be the port graph on the left-hand side of the rewrite rule, which may include variables, and G will be the ground port graph to be rewritten, without variables.

Definition 4.3 [Matching morphism] Let $L \Rightarrow R$ be a port graph rewrite rule and G a ground port graph. A *redex* $g(L)$ of the left-hand side is found in G if there is a total port graph morphism g, called *matching morphism*, from L to G such that $g(L)$ is a subgraph of G where for all labels ll of the arrow node $\Rightarrow$, $g(ll)$ is valid.

In PORGY, the graph morphism g is such that
- if the arrow node has an attribute *Where* with value B, then $g(B)$ is true for $g(L)$
- if the arrow node has an attribute $Saturated = (p_1, \ldots, p_n)$, for each p_k, $1 \leq k \leq n$, there are no edges between $g(p_k)$ and ports outside $g(L)$ in G.

Definition 4.4 [Rewriting step] A *rewriting step* on G uses a rule $L \Rightarrow R$ and a total matching morphism $g : L \mapsto G$ such that the two conditions expressed via the labels $Where$ and $Saturated$ are satisfied. It transforms G into a new graph G' obtained from G by performing the following operations in three ordered phases already described in [13]:

(i) In the *build* phase, after a redex $g(L)$ is found in G, a copy $R_c = g(R)$ is added to G.

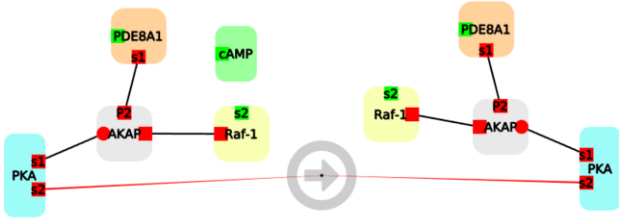
(ii) The *rewiring* phase then redirects edges from G to R_c in the following order: for each port p in the arrow node:
   (a) If p is a blackhole: for each port $p_L \in L$ connected to p, destroy all the edges connected to $g(p_L)$ in G.
   (b) If p is a bridge port and $p_L \in L$ is connected to p: for each port $p^i_R \in R$ connected to p, find all the ports p^k_G in G that are connected to $g(p_L)$ and are not in $g(L)$, and redirect each edge connecting p^k_G and $g(p_L)$ to connect p^k_G and $p^i_{R_c}$.
   (c) If p is a wire port connected to two ports p_1 and p_2 in L, then take all the ports outside $g(L)$ that are connected to $g(p_1)$ in G and connect each of them to each port outside $g(L)$ connected by an edge to $g(p_2)$.

(iii) The *deletion* phase simply deletes $g(L)$. This creates the final graph G'.

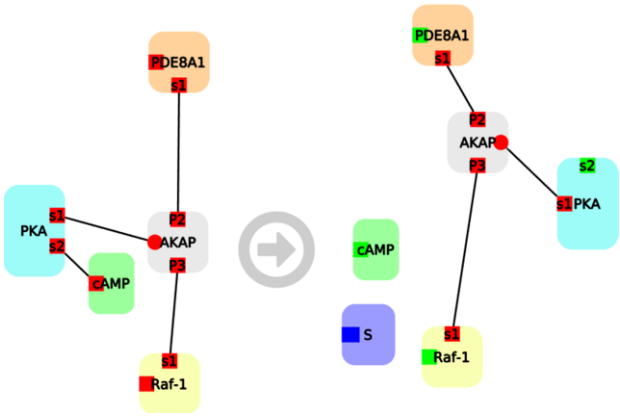
Note that the order in which the rewiring is performed on the different types of ports is important: for example, if an edge exists in G between a port p_1 connected to a bridge port and a port p_2 connected to a blackhole port, priority is given to deletion of this edge.

Figure 4 gives an example of two port graph rewrite rules defined for the biochemical process of Section 3.1. Rule (a) has red edges connecting the port $s2$ in

PKA to a bridge port in the arrow node. This is because in the graph where this rule is applied, there could be other edges arriving to $s2$ from the outside. However, no edges can be connected to $cAMP$ since its port is not connected to the arrow node: if an edge arrives to cAMP from the outside the Saturated condition fails and the rule does not apply.



(a) A rule that "eats" a "cAMP" node.



(b) Another rule that disconnects the "cAMP" and creates a "S" node. The labels of several ports are also changed (the Colour attribute changed from red to green).

Fig. 4. Two rules used to describe the biochemical process of Section 3.1

Figure 5 gives another example with a $Where$ attribute, defined for the social network model of Section 3.2.

The $Where$ attribute can also specify the existence of an external edge between the image in G of a port p in L and a port outside $g(L)$, as required in Kappa [8], written as $Where := ExternalLinked(p)$ or $Where := Arity(p) > n$.

Figure 6 shows two rules in an interaction net system defining the operation of addition on natural numbers represented by 0 and S (successor). In the left-hand side of the first rule (a), the agents $+$ and S are connected via their "principal ports", called P in the picture. The right-hand side of the rule shows the result of the interaction: the auxiliary port of S (labelled 1) is now connected to $+$. Note that there is a wire port in the arrow node of rule (b), to handle the fact that the result of the addition of 0 and a number n is n.

In general, the behaviour of a system is modelled by several rules. A *rewrite system* is then a port graph made of this set of rules. Structuring a rewrite system

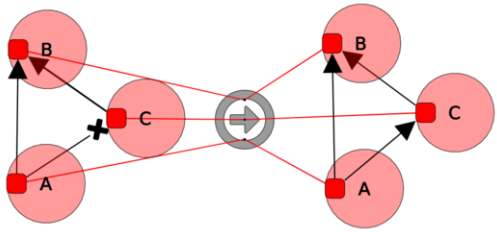
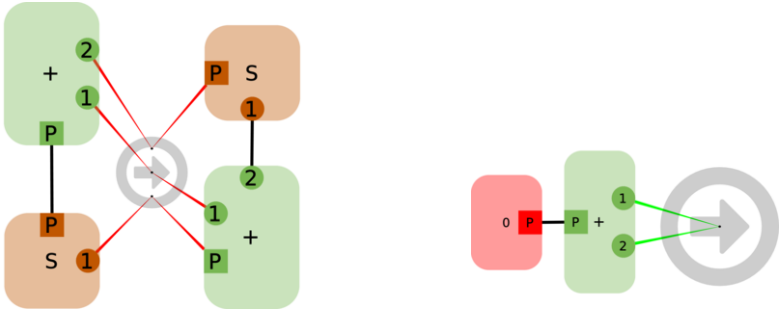


Fig. 5. A rule used to describe interaction in a social network. If A and C both know B and if A and C do not know each other, then they should meet.



(a) Rule describing the interaction between + and S.

(b) Addition with 0.

Fig. 6. Interaction net system defining addition of natural numbers. Rule (a) defines the interaction between the agents + and S and represents the standard reduction $n + S(m) \to S(n + m)$). Rule (b) specifies the interaction between 0 and + and represents $0 + n \to n$.

is easily performed by adding new nodes with different names for each subset and linking this node to the arrow node of each rule in the subset.

5 Located graphs and rules

Located port graphs have been introduced in PORGY to indicate explicitly the *positions* in a graph where rewriting should be performed [2]. They can also specify parts of the graph that should be protected (i.e., sub-graphs where rewriting is *banned*).

Definition 5.1 [Located graph] [13]. A *located graph* G_P^Q consists of a port graph G and two distinguished subgraphs P and Q of G, called respectively the *position subgraph*, or simply *position*, and the *banned subgraph*.

At this point, it is easy to see located graphs as labelled port graphs: we may introduce two labels *Pos* and *Ban* taking two possible Boolean values *on* (true) and *off* (false).

In a located graph G_P^Q, P is the subgraph of G made of nodes where the *Pos* label has value *on*, and related edges. This is the focus of the next step(s). Q is a protected subgraph, made of nodes with a *Ban* label *on*, where transformations are forbidden. We put the additional restriction that initially it is not possible to have the same node in the position and in the banned subgraph. P and Q are disjoint.

This property is then maintained by the rewriting process we are defining below.

When applying a port graph rewrite rule, not only the underlying graph G but also the position and banned subgraphs are updated. A *located rewrite rule*, defined below, specifies two disjoint subgraphs M and M' of the right-hand side R that are respectively used to update the position and banned subgraphs. If M (resp. M') is not specified, R (resp. the empty graph $\emptyset$) is used as default. A subgraph W in the left-hand side specifies which nodes are expected to be in the position subgraph P of G. If W is not specified then the requirement is that at least one node from L should be in the position subgraph.

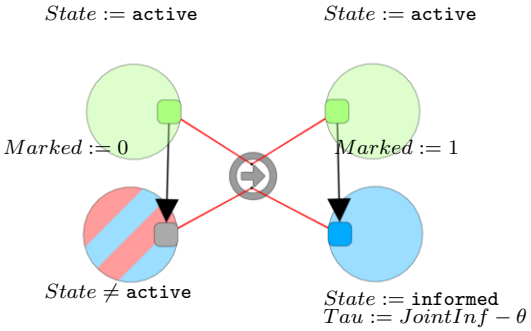
As above, nodes in L and R have labels Pos and Ban either being a variable x or taking one of the two possible values *on* and *off*. We give details below, where we use the operators $\cup, \cap, \setminus$ to denote union, intersection and complement of port graphs. These operators are defined on port graphs from the usual set operations on sets of nodes, ports and edges, except for $\setminus$ where edges attached to ports are dropped when the ports are not in the difference to avoid dangling edges.

Definition 5.2 [Located rewrite rule] A *located rewrite rule* is given by a port graph rewrite rule $L \Rightarrow R$, together with two disjoint subgraphs M and M' of R where: the labels Pos and Ban for any node n in M have respectively the values *on* and *off*, for any node in M' values *off* and *on* respectively, and optionally, a subgraph W of L where node labels Pos have value *on* and labels Ban have value *off*; for the rest of the nodes in L, the labels Pos are variables (different for each node) and Ban labels have value *off*. It is denoted $L_W \Rightarrow R_M^{M'}$.

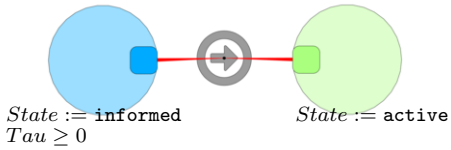
We write $G_P^Q \rightarrow^g_{L_W \Rightarrow R_M^{M'}} {G'}_{P'}^{Q'}$ and say that the located graph G_P^Q *rewrites to* ${G'}_{P'}^{Q'}$ *using* $L_W \Rightarrow R_M^{M'}$ *at position* P *avoiding* Q, if $G \rightarrow_{L \Rightarrow R} G'$ with a morphism g satisfying the condition: there exists at least one node n in L such that the label Pos in n is matched by the value *on* in $g(n)$, that is, $g(n) \in P$.

In other words, in a located rewrite rule $L_W \Rightarrow R_M^{M'}$, W is the subgraph of L made of nodes with a Pos label equal to *on* and a Ban label equal to *off*, and related edges. M is the subgraph of R made of nodes with a Pos label equal to *on* and a Ban label equal to *off*, with related edges. M' is the subgraph of R made of nodes with a Ban label *on* and a Pos label *off*. The morphism g used in a rewriting step with a located rule $L_W \Rightarrow R_M^{M'}$ is such that $g(L) \cap P = g(W)$ or simply $g(L) \cap P \neq \emptyset$ if W is not provided, and $g(L) \cap Q = \emptyset$. The new position subgraph P' and banned subgraph Q' are defined as $P' = (P \setminus g(L)) \cup g(M)$, and $Q' = (Q \cup g(M')$; if M (resp. M') are not provided then we assume $M = R$ (resp. $M' = \emptyset$).

Located graphs have been used in [14] to model propagation in social networks. In the linear threshold propagation model for instance, propagation is specified with two rules given in Figure 7. When applying the first rule *LT influence trial*, the active green node in the left-hand side must correspond to a node in the position subgraph P and the informed blue node in the right-hand side has a label Pos equal to *on*, so that its image belongs to the updated P in the transformed network. It can then be selected to apply the second rule *LT activate*.



(a) *LT influence trial*: Joint influence computation from an `active` neighbour on an *inactive* node (either `unaware` or just `informed`).



(b) *LT activate*: an `informed` node becomes `active` when sufficiently influenced.

Fig. 7. Rules used to express the Linear Threshold model **LT**. `Active` nodes are `green`, `informed` nodes are `blue` and `unaware` nodes are `red`. A bi-colour `red/blue` node can be in either of the two states `unaware` or `informed`.

6 Derivation graph and strategies

A sequence of rewriting steps is called a *rewriting derivation*. This is illustrated in Figure 8 where the initial interaction net of Figure 3 is rewritten in two steps with the rules of Figure 6.

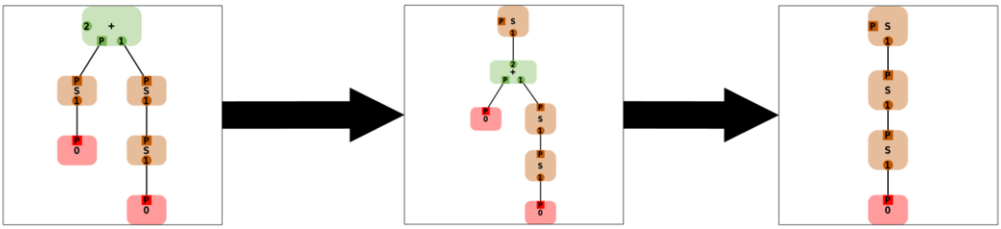


Fig. 8. Example of a rewriting derivation

Although the rewriting process naturally generates a derivation tree, some nodes may be isomorphic, for instance in case of confluent rewrite systems. So in general, we consider derivation graphs. Starting from an input state formalised as an attributed port graph, rewriting steps (applied sequentially, concurrently or probabilistically) build a graph consisting of derivations, which correspond to sequential transformations. In this graph, nodes are states and edges represent transitions (e.g., rewriting steps).

Labels are quite useful in this graph too. Edges have labels recording information on the rewriting step: the rule applied, the redex(es); in case of a probabilistic choice of transitions, the probability associated to the choice of this rule. Different types of edges can be visually distinguished thanks to labels *Colour* and *Shape*. For example, additional edges can be created as shortcuts between two states, for

instance to represent a derivation in a more concise way. Such a shortcut has a label that records the sequence of steps involved in the shortcut. Figure 9 gives an example of a derivation graph, where shortcut edges are depicted in green.

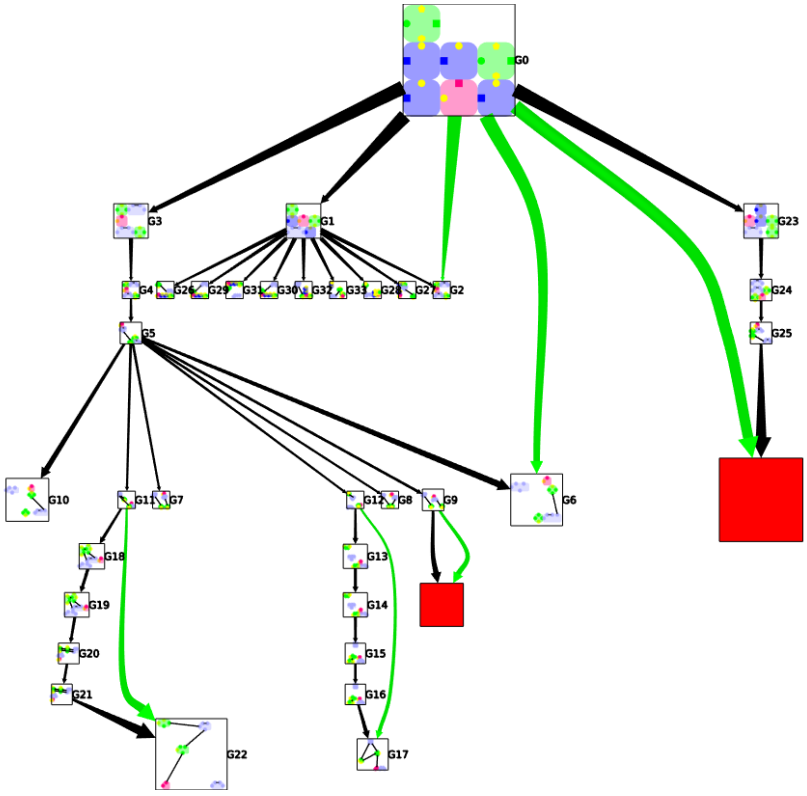


Fig. 9. A derivation graph built from a toy biochemical process example. Shortcuts are green edges. Red nodes are failure states.

Strategies are used to control the application of rules (including probabilistic application) and to focus on points of interest: strategies define which rule(s) should be applied and where (see [17,5,18] for general definitions). They may also express shortcuts between two states as shown in Figure 9. The language used in PORGY to write strategies for port graph rewriting is defined in [13].

Given graphs, rules and strategies, we can now introduce a more abstract notion of *symbolic derivation port graph*, closely connected to the operational semantics of *graph programs*.

Definition 6.1 [Graph program] A *(strategic rewrite) graph program* consists of a finite graph of located rewrite rules $\mathcal{R}$, a strategy expression $S_{\mathcal{R}}$ (built from $\mathcal{R}$ using the strategy language $\mathcal{L}$) and a located graph G_P^Q. When $\mathcal{R}$ is clear from the context, we write simply (S, G_P^Q) to denote a strategic rewrite graph program and call it a *graph program*.

Formally, the semantics of a graph program (S, G_P^Q) is specified in [13] using a transition system, defining a *small step* operational semantics in SOS style [27].

Here, we introduce *symbolic derivation graphs* to represent graph programs and their execution.

Definition 6.2 [Symbolic derivation graph] A *symbolic derivation graph* for a strategy language $\mathcal{L}$ is a labelled port graph where nodes have two distinguished labels: *Current graph* is a located port graph, and *Strategy* is an expression of the strategy language $\mathcal{L}$. Edges are labelled by strategy expressions. Each node has two ports, named Parent and Successors, respectively.

Nodes in symbolic derivation graphs represent graph programs, and edges represent transition steps. Looking at the transition rules of the small step operational semantics in [13] that apply to a node labelled by a strategy S and the current graph G, the application condition of each step is expressed either through a new expression of the strategy S' or a new port graph G' (that may result from the execution of another graph program).

A symbolic derivation graph is *valid* if whenever two nodes labelled by (S, G_P^Q) and $(S', G'^{Q'}_{P'})$ are linked by an edge, there is a transition between the graph programs (S, G_P^Q) and $(S', G'^{Q'}_{P'})$ in the operational semantics for the strategy language. It is *complete* if for every transition $(S, G_P^Q) \longmapsto (S', G'^{Q'}_{P'})$ according to the operational semantics, there exists an edge between the nodes labelled by S, G_P^Q and $S', G'^{Q'}_{P'}$. The latter implies that all the graphs that can be derived from G_P^Q according to S can be found in the derivation graph by following the paths starting from (S, G_P^Q). In this sense, the derivation graph represents the execution of the graph program (S, G_P^Q).

7 Conclusion

We have presented in this paper the concepts underlying the Porgy framework using the structure of labelled port graphs with symbolic labels. This point of view is also reflected at the implementation level: Porgy is implemented on top of the visualisation framework Tulip [4], which is based on a notion of labelled graph where labels are properties. More precisely, a Tulip graph is basically made of three sets: a set of nodes, a set of edges and a set of properties that are defined for every node and edge. The notion of property in Tulip is close to Porgy's notion of attribute in a record.

Before looking at the perspectives opened by this work, let us first mention a few other systems or approaches closely related to ours.

7.1 Related works

Graphs are used in many forms and contexts in computer science and the need to generate, visualise and transform them led to the development of a variety of tools implementing labelled graph transformation and rewriting. With the aim of promoting these concepts in the software developers community using UML and Java, the language of Story diagrams [15], embedded in the Fujaba Tool Suite [24],

adopts most of the features of Progres [31] but avoids the backtracking mechanism related to the non-determinism of graph rewriting. GROOVE [30] is another graph transformation system closely-related to PORGY. It uses labelled graphs, and transformations are specified by rules and control. GROOVE has been used to model and analyse complex systems in various domains as illustrated in [16]. It is a versatile tool; however, it does not provide the visualisation and animation features available in PORGY. GP [28] is also a closely related rule-based, non-deterministic programming language, where programs are defined by sets of graph rewrite rules and a textual strategy expression. The strategy language has three main control constructs: sequence, repetition and conditional. Since the aim is to execute graphs programs efficiently, GP builds only one rewriting derivation, although early versions of GP used a Prolog-like backtracking technique to explore the whole derivation graph. GP does not provide mechanisms to visualise the derivation tree, unlike PORGY, where users can interactively navigate on the tree, visualise alternative derivations, follow the evolution of specific redexes, etc. None of the languages above has Position constructs. Compared to these systems, PORGY's strategy language clearly separates the issues of selecting positions for rewriting and selecting rules, with primitives for focusing as well as traditional strategy constructs.

Graph rewriting is also widely used in chemistry and biology. Systems such as BioNetGen [11], RuleBender [32], Mosbie [33] address the problem of modelling huge graphs. They integrate visualisation with modelling and simulation of rule-based intracellular biochemistry, but do not provide a strategy language. However the rules are quite similar to PORGY's and BioNetGen uses port graphs.

The graph transformation approach developed in [20] encapsulate in "units" rules and control conditions as in our strategic rewrite graph programs. Control is expressed through regular expressions in a less powerful language than our strategy language. But their independence approach applying to all kinds of graphs, rules, rule applications and control conditions is somehow close to our concern of generic structure provided by labelled port graphs.

7.2 Perspectives

The generic notions of symbolic label and corresponding labelled port graphs seem to offer a lot of expressivity but raise several questions that need to be further explored.

Semantic considerations for labelled port graph rewriting as defined in this paper need to be addressed. In [13], we show that a large subclass of labelled port graphs are attributed graph structures as defined in [22], and explore the correspondence between the operational notion of rewriting given above and the construction of single pushout (SPO) objects. However these first results have to be extended to the whole class of labelled port graphs and graph rewriting as defined in this paper and covering for instance node duplication and port graph cloning, in the direction proposed by [7].

Another research direction is to consider as symbolic labels first-order formulas as in [25]. They show that their grounded symbolic graphs coincide with attributed

graphs, which justifies to consider DPO/SPO rewriting semantics for labelled port graphs. Symbolic labels may in particular allow us to take into account constraint satisfiability. In this line, we plan to consider a more abstract notion of constraint where graph structures and labels interpreted in semantic domains are used to generate graphs. Such a generic notion of constraint-based labelled port graph may be interesting for reasoning on graph rewriting, narrowing and completion.

In this work, strategies are expressions in a formal language to control rule application. An open question is to represent strategies as labelled port graphs, in a way similar to the representation of strategies as rho-terms in rho-calculus [6]. The first step is already achieved since a strategy reduced to one rule is already a labelled port graph and preliminary work in this direction is provided in [3]. This would open the way to design a reflective logical framework based on a rho-graph calculus.

Another direction for further work is to introduce structuring mechanisms on strategic rewrite programs and labelled port graphs. A promising direction we want to explore is the concept of multilayer graph, inspired by multilayer networks [19].

Acknowledgement

We thank Oana Andrei, Guy Melançon and Olivier Namet for their work in the initial PORGY project (2009–2012); their ideas and enthusiasm were invaluable during the early stages of development of this tool. We also thank Jason Vallet for implementing several features of PORGY, writing the documentation and developing the social network propagation influence example. Our thanks also go to the anonymous referees whose valuable remarks and suggestions helped us to improve the first version of this paper.

References

[1] Andrei, O., "A Rewriting Calculus for Graphs: Applications to Biology and Autonomous Systems," Ph.D. thesis, Institut National Polytechnique de Lorraine (2008).

[2] Andrei, O., M. Fernández, H. Kirchner, G. Melançon, O. Namet and B. Pinaud, *PORGY: Strategy-Driven Interactive Transformation of Graphs*, in: R. Echahed, editor, 6^{th} *Int. Workshop on Computing with Terms and Graphs*, Electronic Proceedings in Theoretical Computer Science **48**, 2011, pp. 54–68. URL http://hal.inria.fr/inria-00563249/en

[3] Andrei, O. and H. Kirchner, *A Higher-Order Graph Calculus for Autonomic Computing*, in: *Graph Theory, Computational Intelligence and Thought. Golumbic Festschrift*, Lecture Notes in Computer Science **5420** (2009), pp. 15–26.

[4] Auber, D., D. Archambault, R. Bourqui, M. Delest, J. Dubois, A. Lambert, P. Mary, M. Mathiaut, G. Mélançon, **Bruno Pinaud**, B. Renoust and J. Vallet, *TULIP 5*, in: R. Alhajj and J. Rokne, editors, *Encyclopedia of Social Network Analysis and Mining*, Springer new-York, 2017 pp. 1–28.

[5] Bourdier, T., H. Cirstea, D. J. Dougherty and H. Kirchner, *Extensional and intensional strategies*, in: *Proceedings Ninth International Workshop on Reduction Strategies in Rewriting and Programming*, Electronic Proceedings in Theoretical Computer Science **15**, 2009, pp. 1–19.

[6] Cirstea, H. and C. Kirchner, *The rewriting calculus — Part I* and *II*, Logic Journal of the Interest Group in Pure and Applied Logics **9** (2001), pp. 427–498.

[7] Corradini, A., D. Duval, R. Echahed, F. Prost and L. Ribeiro, *The pullback-pushout approach to algebraic graph transformation*, in: J. de Lara and D. Plump, editors, *Graph Transformation* (2017), pp. 3–19.

[8] Danos, V., J. Feret, W. Fontana, R. Harmer, J. Hayman, J. Krivine, C. Thompson-Walsh and G. Winskel, *Graphs, Rewriting and Pathway Reconstruction for Rule-Based Models*, in: S. D. L.-Z. fuer Informatik, editor, *FSTTCS 2012 - IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science*, LIPIcs **18**, Hyderabad, India, 2012, pp. 276–288.
URL https://hal.archives-ouvertes.fr/hal-00809065

[9] Ehrig, H., K. Ehrig, U. Prange and G. Taentzer, "Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science. An EATCS Series)," Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[10] Ehrig, H. and B. Mahr, "Fundamentals of Algebraic Specification 1: Equations and Initial Semantics," Monographs in Theoretical Computer Science. An EATCS Series, Springer Publishing Company, 1985.

[11] Faeder, J., M. Blinov and W. Hlavacek, *Rule-based modeling of biochemical systems with bionetgen*, in: I. V. Maly, editor, *Systems Biology*, Methods in Molecular Biology **500**, Humana Press, 2009 pp. 113–167.
URL http://dx.doi.org/10.1007/978-1-59745-525-1_5

[12] Fernández, M., H. Kirchner and O. Namet, *A strategy language for graph rewriting*, in: G. Vidal, editor, *Logic-Based Program Synthesis and Transformation*, Lecture Notes in Computer Science **7225**, Springer Berlin Heidelberg, 2012 pp. 173–188.
URL http://dx.doi.org/10.1007/978-3-642-32211-2_12

[13] Fernández, M., H. Kirchner and B. Pinaud, *Strategic Port Graph Rewriting: an Interactive Modelling Framework*, Research report, Inria ; LaBRI - Laboratoire Bordelais de Recherche en Informatique ; King's College London (2017).
URL https://hal.inria.fr/hal-01251871

[14] Fernandez, M., H. Kirchner, B. Pinaud and J. Vallet, *Labelled Graph Strategic Rewriting for Social Networks*, Journal of Logical and Algebraic Methods in Programming (2018).
URL https://hal.archives-ouvertes.fr/hal-01664593

[15] Fischer, T., J. Niere, L. Torunski and A. Zündorf, *Story diagrams: A new graph rewrite language based on the unified modeling language and java*, in: H. Ehrig, G. Engels, H.-J. Kreowski and G. Rozenberg, editors, *Theory and Application of Graph Transformations* (2000), pp. 296–309.

[16] Ghamarian, A. H., M. de Mol, A. Rensink, E. Zambon and M. Zimakova, *Modelling and analysis using groove*, International Journal on Software Tools for Technology Transfer **14** (2012), pp. 15–40.
URL https://doi.org/10.1007/s10009-011-0186-x

[17] Kirchner, C., F. Kirchner and H. Kirchner, *Strategic computations and deductions*, in: *Reasoning in Simple Type Theory. Studies in Logic and the Foundations of Mathematics, vol.17*, College Publications, 2008 pp. 339–364.

[18] Kirchner, H., *Rewriting strategies and strategic rewrite programs*, in: *Logic, Rewriting, and Concurrency (LRC 2015), Festschrift Symposium in Honor of José Meseguer*, Lecture Notes in Computer Science (2015), pp. 380–403.
URL https://hal.inria.fr/hal-01143486

[19] Kivela, M., A. Arenas, M. Barthelemy, J. P. Gleeson, Y. Moreno and M. A. Porter, *Multilayer networks*, Journal of Complex Networks **2** (2014), pp. 203–271.
URL http://comnet.oxfordjournals.org/content/2/3/203.abstract

[20] Kreowski, H.-J., S. Kuske and G. Rozenberg, "Graph Transformation Units – An Overview," Springer Berlin Heidelberg, Berlin, Heidelberg, 2008 pp. 57–75.
URL https://doi.org/10.1007/978-3-540-68679-8_5

[21] Lafont, Y., *Interaction nets*, in: *Proceedings of the 17th ACM Symposium on Principles of Programming Languages (POPL'90)* (1990), pp. 95–108.

[22] Löwe, M., M. Korff and A. Wagner, *An algebraic framework for the transformation of attributed graphs*, in: M. R. Sleep, M. J. Plasmeijer and M. C. J. D. van Eekelen, editors, *Term Graph Rewriting*, John Wiley and Sons Ltd., Chichester, UK, 1993 pp. 185–199.
URL http://dl.acm.org/citation.cfm?id=167817.167848

[23] Meseguer, J., *Twenty years of rewriting logic*, The Journal of Logic and Algebraic Programming **81** (2012), pp. 721 – 781, rewriting Logic and its Applications.
URL http://www.sciencedirect.com/science/article/pii/S1567832612000707

[24] Nickel, U., J. Niere and A. Zündorf, *The FUJABA environment*, in: *Proccedings of International Conference on Software Engineering-ICSE*, 2000, pp. 742–745.

[25] Orejas, F. and L. Lambers, *Symbolic attributed graphs for attributed graph transformation*, ECEASST **30** (2010), pp. 1–33.

[26] Pinaud, B., G. Melançon and J. Dubois, *PORGY: A Visual Graph Rewriting Environment for Complex Systems*, Computer Graphics Forum **31** (2012), pp. 1265–1274.
URL http://hal.inria.fr/hal-00682550

[27] Plotkin, G. D., *A structural approach to operational semantics*, Journal of Logic and Algebraic Programming **60-61** (2004), pp. 17–139.

[28] Plump, D., *The Graph Programming Language GP*, in: S. Bozapalidis and G. Rahonis, editors, *Algebraic Informatics CAI*, Lecture Notes in Computer Science **5725** (2009), pp. 99–122.

[29] Plump, D. and S. Steinert, *The semantics of graph programs*, in: *Proceedings Tenth International Workshop on Rule-Based Programming, RULE 2009, Brasília, Brazil, 28th June 2009.*, 2009, pp. 27–38.
URL http://dx.doi.org/10.4204/EPTCS.21.3

[30] Rensink, A., *The GROOVE Simulator: A Tool for State Space Generation*, in: *Applications of Graph Transformations with Industrial Relevance (AGTIVE)*, Lecture Notes in Computer Science **3062** (2003), pp. 479–485.

[31] Schürr, A., A. J. Winter and A. Zündorf, *The PROGRES Approach: Language and Environment.*, in: H. Ehrig, G. Engels, H.-J. Kreowski and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 2: Applications, Languages, and Tools*, World Scientific, 1997 pp. 479–546.

[32] Smith, A. M., W. Xu, Y. Sun, J. R. Faeder and G. Marai, *Rulebender: integrated modeling, simulation and visualization for rule-based intracellular biochemistry*, BMC Bioinformatics **13** (2012).
URL http://dx.doi.org/10.1186/1471-2105-13-S8-S3

[33] Wenskovitch, J. E., L. A. Harris, J.-J. Tapia, J. R. Faeder and G. E. Marai, *Mosbie: a tool for comparison and analysis of rule-based biochemical models*, BMC Bioinformatics **15** (2014).
URL http://dx.doi.org/10.1186/1471-2105-15-316