Strategic Port Graph Rewriting: an Interactive Modelling Framework

Maribel Fernández, Hélène Kirchner, Bruno Pinaud

Strategic Port Graph Rewriting: an Interactive Modelling Framework

Maribel Fernández[1], Hélène Kirchner[2], and Bruno Pinaud[3]

[1] King's College London, Department of Informatics, Strand, London WC2R 2LS, UK `maribel.fernandez@kcl.ac.uk`
[2] Inria, 200 avenue de la Vieille Tour, 33405 Talence, France `helene.kirchner@inria.fr`
[3] University of Bordeaux, LaBRI CNRS UMR5800, 33405 Talence Cedex, France`bruno.pinaud@u-bordeaux.fr`

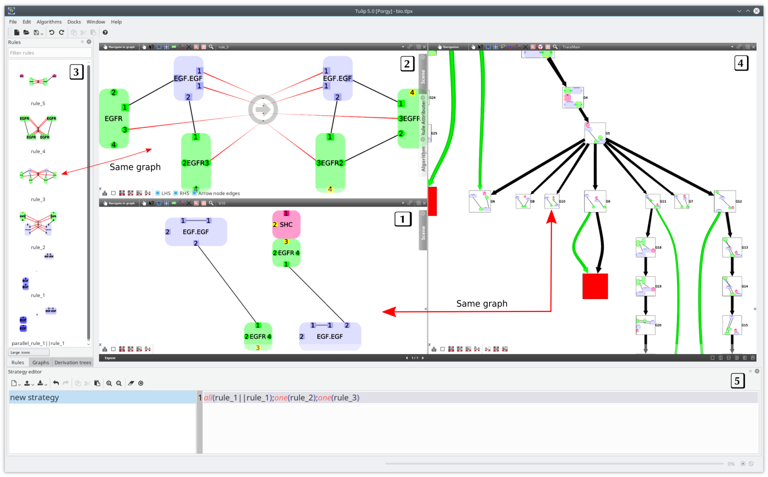
Abstract. We present strategic port graph rewriting as a basis for the implementation of visual modelling tools. The goal is to facilitate the specification and programming tasks associated with the modelling of complex systems. A system is represented by an initial graph and a collection of graph rewrite rules, together with a user-defined strategy to control the application of rules. The traditional operators found in strategy languages for term rewriting have been adapted to deal with the more general setting of graph rewriting, and some new constructs have been included in the strategy language to deal with graph traversal and management of rewriting positions in the graph. We give a formal semantics for the language, and describe its implementation: the graph transformation and visualisation tool PORGY.

1 Introduction

In this paper we present strategic port graph rewriting as a basis for the design of PORGY– a visual, interactive environment for the specification, debugging, simulation and analysis of complex systems. PORGY is a graphical, executable specification language, with an interface that allows users to visualise and analyse the dynamics of the system being modelled (see Fig. 1).

To model complex systems, graphical formalisms are often preferred to textual ones, since diagrams make it easier to understand the system and convey intuitions about it. The dynamics of the system can then be specified using graph rewrite rules. Graph rewriting has solid logic, algebraic and categorical foundations [20,25], and graph transformations have many applications in specification, programming, and simulation tools [25]. In this paper, we focus on *port graph rewriting systems* [4], a general class of graph rewriting systems that has been used to model systems in a wide variety of domains such as biochemistry, interaction nets, games and social networks [3,4,30,31,54,73].

PORGY [58] is 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



```
new strategy
all(rule_1||rule_1);one(rule_2);one(rule_3)
```

Fig. 1. Overview of PORGY: (1) editing one state of the graph being rewritten; (2) editing a rule; (3) some available rewrite rules; (4) portion of the derivation tree, a complete trace of the computing history; (5) the strategy editor.

provides a *strategy language*. In this paper, we give a categorical semantics for rewriting steps, a formal operational semantics for the strategy language, and examples of application inside this environment.

Reduction strategies define which (sub)expression(s) should be selected for evaluation and which rule(s) should be applied (see [13, 42, 43] for general definitions). These choices affect fundamental properties of computations such as laziness, strictness, completeness, termination and efficiency, to name a few (see, e.g., [51, 71, 75]). Used for a long time in λ-calculus [9], strategies are present in programming languages such as Clean [60], Curry [38], and Haskell [39] and can be explicitly defined to rewrite terms in languages such as ELAN [12], Stratego [74], Maude [52] or Tom [8]. They are also present in graph transformation tools such as PROGRES [68], AGG [27], Fujaba [55], GROOVE [67], GrGen [33] and GP [63, 64]. PORGY's strategy language draws inspiration from these previous works, but a distinctive feature of PORGY's language is that it allows users to define strategies using not only operators to combine graph rewrite rules but also operators to define the location in the target graph where rules should, or should not, apply.

Strategies are used to control PORGY's rewrite engine: users can create graph rewriting derivations and specify graph traversals using the language primitives to select rewrite rules and the position where the rules apply. Subgraphs can be selected as focusing positions for rewriting interactively (in a visual way), or intentionally (using a focusing expression). Alternatively, rewrite positions could be encoded in the rewrite rules using markers or conditions, as done in other languages based on graph rewriting which do not have explicit position primitives. We prefer to separate the two notions of position and rule to make programs more readable and easier to maintain and adapt. In this sense, the language follows the separation of concerns principle [23]. For example, to change a traversal algorithm, it is sufficient to change the strategy and not the whole rewriting system.

Our main contributions are:

– A formal definition of *attributed port graphs*, where attributes are associated with nodes, port and edges, generalising the notion of port graph defined in [2, 4] and used in [3, 30]. We define port graph morphisms that take attributes and their values into account, and use them in the definition of rewriting.
– A definition of rewrite rule and rewriting step that generalises both port graph rewriting [2, 4], and interaction net rewriting [47]. From a categorical point of view, we mainly follow the *single push out approach* [24] and view a rewrite rule as a pair of graphs (the left and right-hand sides) with a partial morphism that relates them (specified via an *arrow* node). Rewrite rules are used to define *strategic graph programs* – a key notion in PORGY.
– We formalise the concept of strategic graph program. A strategic graph program consists of an initial *located graph* (that is, a port graph with two distinguished subgraphs P and Q specifying the position where rewriting should take place, and the subgraph where rewriting is banned, respectively), and a set of rewrite rules describing its dynamic behaviour, controlled by a strategy. Located graphs generalise the notion of "terms with redexes".
– We provide a language to specify strategies with a formal operational semantics. More precisely, we give a small-step operational semantics for strategic graph programs, specified by a transition system such that each strategic graph program is associated with a set of rewriting derivations, or traces, which can be represented as a *derivation tree*. The strategy language includes probabilistic primitives, for which we provide an operational semantics using a probabilistic transition system.
– We provide an implementation of strategic graph programs in PORGY. PORGY offers visual representations not only for port graphs and rewrite rules but also for the derivation tree. Its user interface permits to interact with the system to select where and how rewrite rules should be applied. Users can see how a specific subgraph of the initial graph evolves, extract strategies that ensure specific behaviours and simulate different runs of the system. Moreover, PORGY can help users debug their system, thanks to features such as cycle detection (for example, PORGY can detect if the application of a rule brings the system back to a previous state).

This paper builds on previous work [3, 30], where PORGY and its strategy language were first presented, and [31, 73], where applications in the areas of graph algorithms and social networks were described. Unlike [3, 30], the notion of port graph considered in this paper includes attributes for nodes, ports and also edges, and attributed port graphs are given a formal, algebraic semantics as a *graph structure* [49,50]. A categorical semantics for port graph rewriting is also provided, following the single pushout approach, and the operational semantics of the strategy language is formally defined. The definition of strategic graph program is more general than the one considered in [3, 30], and easier to use because the strategy language includes a sublanguage to deal with properties, which facilitates the specification of rewrite positions and banned subgraphs (to be protected during rewriting).

The paper is organised as follows. In Section 2, we define the concepts of attributed port graph and port graph rewriting, and give a single pushout semantics for rewriting steps. In Section 3, we present strategic graph programs and the syntax of the strategy language. Section 4 illustrates the language with examples. Section 5 formally defines its semantics and states some properties. Section 6 describes PORGY's implementation. Related languages are presented in Section 7 and Section 8 gives directions for future work.

2 Port Graph Rewriting

Several definitions of graph rewriting are available, using different kinds of graphs and rewrite rules (see, for instance, [10,19,24,36,47,62]). In this paper we consider *port graphs* with *attributes* associated with nodes, ports and edges, generalising the notion of port graph introduced in [2, 4, 5].

2.1 Port graphs

Intuitively, a port graph is a graph where nodes have explicit connection points called *ports*; edges are attached to ports. The advantage of using port graphs rather than plain graphs is that they allow us to express in a convenient way the properties of the connections: ports represent the connection points between edges and nodes.

Nodes, ports and edges are labelled by a set of attributes. For instance, a port may have a state (e.g., active/inactive or principal/auxiliary) and a node may have properties such as colour, shape, etc.

In order to describe port graphs and port graph rewriting, we consider sets $\mathscr{N}, \mathscr{E}, \mathscr{P}$ of nodes, edges and ports, respectively, which are used to build port graphs. Every element is labelled by a record that contains all its properties. Records are a central data structure in modern programming languages, and are equally important in software management and computational linguistics.

Definition 1 (Record). *A record r is a set 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, denoted by $r \cdot a_i$; the elements a_i are pairwise distinct.*

The function Atts applies to records and returns all the attributes:

$$Atts(r) = \{a_1, \ldots, a_n\}$$

if $r = \{a_1 := v_1, \ldots, a_n := v_n\}$.

Each record $r = \{a_1 := v_1, \ldots, a_n := v_n\}$ contains one pair where $a_i = Name$. The attribute $Name$ defines the type of the record in the following sense: for all r_1, r_2, $Atts(r_1) = Atts(r_2)$ if $r_1.Name = r_2.Name$.

Values in records can be concrete (numbers, Booleans, etc.), or can be 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 built over Σ and $\mathcal{X}_S$.

Records with abstract values (i.e., expressions $v_i \in T(\Sigma, \mathcal{X}_S)$ that may contain variables), will allow us to define generic patterns in rewrite rules: abstract values in left-hand sides of rewrite rules will be matched against concrete data in the graphs to be rewritten. We use variables not only in values but also to denote generic attributes and generic records in port graph rewrite rules.

Port graphs are now defined as an algebra (sets and functions defined on these sets) in the following way:

Definition 2 (Attributed port graph). *An* attributed port graph $G = (V, P, E, D)_\mathcal{F}$ *is given by a tuple* (V, P, E, D) *where:*

- $V \subseteq \mathscr{N}$ *is a finite set of nodes;* $n, n_1, \ldots$ *range over nodes;*
- $P \subseteq \mathscr{P}$ *is a finite set of ports;* $p, p_1, \ldots$ *range over ports;*
- $E \subseteq \mathscr{E}$ *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 records;*

and a set $\mathcal{F}$ *of functions Connect, Attach and* $\mathcal{L}$ *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;*
- $\mathcal{L} : V \cup P \cup E \mapsto D$ *is a labelling function that returns a record for each element in* $V \cup P \cup E$*.*

Moreover, we assume that for each node $n \in V$*,* $\mathcal{L}(n)$ *contains an attribute Interface whose value is the list of names of its ports, that is,* $\mathcal{L}(n) \cdot Interface = [\mathcal{L}(p_i) \cdot Name \mid Attach(p_i) = n]$ *such as the following constraint is satisfied:*

$$\mathcal{L}(n_1) \cdot Name = \mathcal{L}(n_2) \cdot Name \Rightarrow \mathcal{L}(n_1) \cdot Interface = \mathcal{L}(n_2) \cdot Interface.$$

By definition of record, nodes/ports/edges with same name (i.e., the same value for the attribute *Name*) have the same attributes, but could have different values. This type constraint is stronger for nodes: Definition 2 forces nodes with the same name to have the same port names (i.e., the same interface) although other attribute values may be different.

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

Unlabelled port graphs, which consist of sets of nodes with ports and edges connecting nodes via ports, and labelled port graphs, where graph elements have atomic labels [2,3,30], can be seen as particular cases of the definition above.

Panel 1 in Figure 1 is an example of a port graph used in a biological case study [3]. It shows two pairs of complex molecules connected by an edge, and one simpler molecule (the pink "SHC"). In the graphical interface, each node is shown with its *Name* and the ports attached to it displayed inside. The values of the attributes *Colour* and *Shape* are taken into account when displaying the node.

We recall that in graph theory, a subgraph of a graph $G = (V_G, E_G)$ is a graph H contained in G, that is, $V_H \subseteq V_G$ and $E_H \subseteq E_G$. The definition extends to port graphs in the natural way: let $G = (V_G, P_G, E_G, D_G)_{\mathcal{F}_G}$ and $H = (V_H, P_H, E_H, D_H)_{\mathcal{F}_H}$ be attributed port graphs. H is a subgraph of G if $V_H \subseteq V_G$, $P_H \subseteq P_G$, $E_H \subseteq E_G$, $D_H \subseteq D_G$ and $\mathcal{F}_H = \mathcal{F}_G|_{V_H \cup P_H \cup E_H \cup D_H}$, that is, the set $\mathcal{F}_H$ of functions defining H is the restriction to H of the functions defining G.

Definition 3 (Adjacent nodes). *Two nodes connected by an edge are* adjacent. *The set of nodes adjacent to a subgraph F in G consists of all the nodes in G* outside *F and adjacent to nodes in F.*

Note that if $F = G$ there are no nodes adjacent to F in G.
Below we refer to attributed port graphs simply as port graphs.

2.2 Port Graph Morphism

If G and H are two port graphs, a *port graph morphism* $f : G \mapsto H$ maps nodes, ports and edges of G to those of H such that the attachment of ports to nodes and the edge connections are preserved, as well as their data values.

Definition 4 (Morphism).
Given two port graphs $G = (V_G, P_G, E_G, D_G)_{\mathcal{F}_G}$ and $H = (V_H, P_H, E_H, D_H)_{\mathcal{F}_H}$ a (partial) morphism *f from G to H, denoted $f : G \mapsto H$, with definition domain $Dom(f)$, is a family of* injective functions *$\langle f_V : V_G \mapsto V_H, f_P : P_G \mapsto P_H, f_E : E_G \mapsto E_H, f_D : D_G \mapsto D_H \rangle$ such that:*

(1) $\forall e \in E_G$, *if* $Connect_G(e) = (p_1, p_2)$ *then* $(f_P(p_1), f_P(p_2)) = Connect_H(f_E(e))$. *This constraint ensures that the morphism preserves the edge connections.*
(2) $\forall n \in V_G$, *if* $Attach_G(p) = n$ *for some* p *then* $f_V(n) = Attach_H(f_P(p))$. *This constraint ensures that the morphism preserves the port attachments.*
(3) *For all* $n \in Dom(f)$, $f_D(\mathcal{L}_G(n)) = \mathcal{L}_H(f_V(n))$
For all $p \in Dom(f)$, $f_D(\mathcal{L}_G(p)) = \mathcal{L}_H(f_P(p))$
For all $e \in Dom(f)$, $f_D(\mathcal{L}_G(e)) = \mathcal{L}_H(f_E(e))$
This constraint ensures that the morphism preserves record attributes and their values; note that f_D may instantiate variables.

We denote by $f(G)$ the subgraph of H consisting of the set of nodes, ports, edges and records that are images of nodes, ports, edges and records in G.

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

2.3 Rewriting

We see a *port graph rewrite rule* as a 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 has a unique name (the rule's label), a condition restricting the rule's application at matching time, and ports to control the rewiring operations at replacement time.

Definition 5 (Port graph rewrite rule). *A port graph rewrite rule is a port graph consisting of:*

- *two 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 edges that each connect a port of the arrow node to ports in L or R. The arrow node has an attribute $Name$ with value $\Rightarrow_{lab}$ (in short, lab denotes the rule's name) which is unique; an attribute $Where := C$ where C is a Boolean expression such that all variables in C occur in L; and a number of ports connected to ports in left- and right-hand sides of the rule.*

Each port in the arrow node has an attribute $Type$ 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 blackhole *must have edges connecting it only to L (one edge or more).*
(3) A port of type wire *must have exactly two edges connecting to L and no edge connecting to R.*

The $Where$ attribute in the arrow node has a value of the form

$$saturated(p_1) \wedge ... \wedge saturated(p_n) \wedge B$$

where $p_1, \ldots, p_n$ are the ports in L not linked by an edge to the arrow node, saturated is a special predicate whose role is explained below, and B is an optional user-defined Boolean expression involving elements of L (edges, nodes, ports and their attributes).

Figure 2 gives an example of port graph rewrite rule.

The introduction of conditional rules is inspired by the GP programming system [63] and by a more general definition given in ELAN [12], in which rules may have Boolean conditions. The expression B is in this paper used to specify the absence of specific edges. For instance, a condition `Where not Edge(p,p')` requires that no edge exists between the ports p and p'; this condition is checked at matching time. The condition involving the *saturated* predicate is automatically generated for every port in L not connected to the arrow node and also

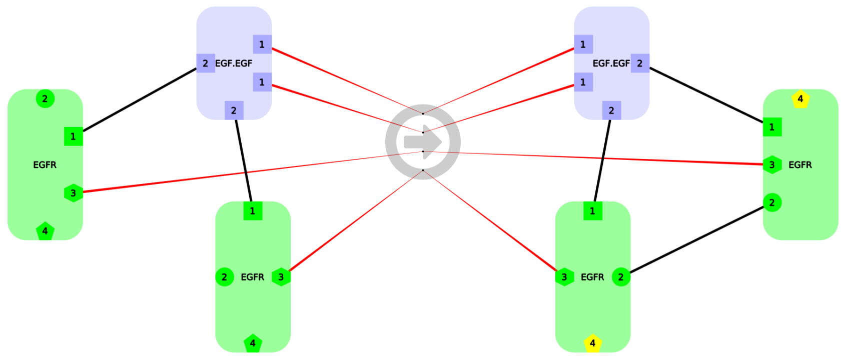


Fig. 2. A close-up on the rule showed in Panel 2 of Fig. 1. This rule adds an edge connecting two green "EGFR" nodes and changes the records of the ports named "4" of the "EGFR" nodes (the Colour attribute changed from green to yellow). Ports named "1" ,"2" and "4" in nodes named "EGFR", as well as both ports labelled "2" in "EGF.EGF" are saturated.

checked during matching. The edges connecting the arrow node with L and R and the *saturated* predicate are used to control the rewiring that occurs during a rewriting step with the rule (see Definition 7), ensuring no dangling edges [24] arise during rewriting (see Property 1 below).

Definition 5 generalises the original definition given in [2], by including case (3), inspired by the notion of rewriting defined for Interaction Nets [47]. This allows us to define more efficient rewrite rules, and additionally Interaction Net rules become a particular case of port graph rewrite rules.

Definition 6 (Match). *Let $L \Rightarrow_C R$ be a port graph rewrite rule and G a port graph without variables (i.e., a* ground *port graph). A* match *$g(L)$ of the left-hand side (also called a* redex*) is found in G if there is a total injective port graph morphism g, called* matching morphism*, from L to G such that if the arrow node has an attribute* Where *with value C, then $g(C)$ is true in $g(L)$. The atom saturated(p) is true if $g(p)$* is not *connected with $G - g(L)$.*

Checking that ports in L that are not connected to the arrow node are saturated (i.e., mapped to ports in $g(L)$ that have no edges connecting them with ports outside the redex) ensures that no edges will be left dangling in rewriting steps (Property 1). In the implementation of Porgy, this condition is checked by ensuring that any port in L not connected to the arrow node has an arity equal to the number of incident edges in the isomorphic port.

Several matching morphisms g from L to G may exist, leading to different rewriting steps; they are computed as solutions of a *matching* problem from L to a subgraph of G.

Definition 7 (Rewriting step). *A* rewriting step *on G using a rule $L \Rightarrow_C R$ (or simply $L \Rightarrow R$) and a matching morphism $g : L \mapsto G$, written $G \rightarrow^g_{L \Rightarrow R} G'$,* *transforms G into a new graph G' obtained from G by performing the following operations in three phases:*

- *In the* build *phase, after a redex $g(L)$ is found in G, a copy $R_c = g(R)$ (i.e., an instantiated copy of the port graph R) is added to G.*
- *The* rewiring *phase then redirects edges from G to R_c as follows:*
  For each port p in the arrow node:
  - *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 $p^i_{R_c}$.
  - *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 $g(p_2)$.*
  - *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.*
- *The* deletion *phase simply deletes $g(L)$. This creates the final graph G'.*

We are now ready to prove that rewriting steps do not leave dangling edges. It is sufficient to show that the result G' of a rewriting step on a port graph G is a port graph.

Property 1. If $G \rightarrow^g_{L \Rightarrow R} G'$ then G' is a port graph (and therefore it cannot have any dangling edges).

Proof. The only nodes in G that are deleted in a rewriting step are the nodes in $g(L)$. We have to prove that all the edges in G connected to ports in $g(L)$ are deleted or redirected to ports in $g(R)$, according to the definition of rewriting step (Definition 7). Let p be a port in $g(L)$. If p is the image of a port in L connected to the arrow node, then all the edges connected to p are dealt with in the definition of rewriting step: the edges are redirected or deleted depending on whether the associated port in the arrow node is a bridge, wire or blackhole; if the port p is the image of a port in L not connected to the arrow node, since the condition *saturated*(p) is checked, there is no edge connecting $G - g(L)$ and $g(p)$ in $g(L)$. The edges in $g(L)$ are deleted in the deletion phase.

Given a finite set $\mathcal{R}$ of rules, a port graph G *rewrites* to G', denoted by $G \rightarrow_{\mathcal{R}} G'$, if there is a rule r in $\mathcal{R}$ and a matching morphism g such that $G \rightarrow^g_r G'$. This induces a reflexive and transitive relation on port graphs, called *the rewriting relation*, denoted by $\rightarrow^*_{\mathcal{R}}$. A port graph on which no rule is applicable is *irreducible*.

2.4 Attributed Port Graph Rewriting and SPO Approach to Graph Transformation

In this section, we show that attributed port graphs, with the notion of rewriting defined in the previous section, can be understood as a single pushout (SPO) graph transformation system.

In the SPO approach, a rule consists of a pair of graphs (the left- and right-hand side of the rule) and a partial graph morphism between them (see [24] for details on the SPO approach for standard graphs). In addition, a matching morphism maps the left-hand side to a subgraph of the graph G to be rewritten. Thus, we have two morphisms with the same source. A rewriting step is then defined by the pushout of this pair of morphisms. One of the main results in this approach is that the pushout of two morphisms with the same source always exists in the category of graphs and partial morphisms. This algebraic approach has been extended to deal with attributed graphs [49, 50].

Below we assume familiarity with basic notions of universal algebra, and briefly recall the notions of *graph structure*, *attributed graph* and corresponding morphism; we refer to [49, 50] for more details.

Definition 8 (Graph Structure). *An algebraic signature $Sig = (S, Op)$ consists of a set S of* sorts *and a set Op of operator symbols. Given an algebraic signature $Sig = (S, Op)$, if A, B are Sig-algebras, a partial Sig-morphism $h : A \mapsto B$ is a total morphism from some sub-algebra A_h of A to B; A_h is called the scope of h.*

A graph structure *is a signature that contains unary operators only.*

*If $GS = (S_1, OP_1)$ is a graph structure, S a subset of S_1 and $SIG = (S_2, OP_2)$ an arbitrary signature, a SIG-*attribution *of GS is an S-indexed family of operator symbols $ATTROP = (ATTROP_s : s \mapsto s2_s)_{s \in S}$ where $s \in S$ and $s2 \in S_2$.*

An attributed graph *is a GS-graph with attributes in SIG, i.e., an algebra with respect to the signature $ATTR = GS + SIG + ATTROP$.*

A morphism $f \colon A \mapsto B$ between GS-graphs A and B having attributes in SIG is a partial GS-morphism $f1 \colon (A)_{GS} \mapsto (B)_{GS}$ together with a total SIG-morphism $f2 \colon (A)_{SIG} \mapsto (B)_{SIG}$ satisfying for all operators $attr \colon s1 \mapsto s2 \in ATTROP$ and all $x \in A(f1)_{s1}, f2(attr^A(x)) = attr^B(f1(x))$.

A rewrite rule is an $ATTR$-morphism r whose SIG-component is an isomorphism $f2 \colon (A)_{SIG} \mapsto (B)_{SIG}$.

As shown in [49], all Sig-algebras and all partial Sig-morphisms form a category $Alg^P(Sig)$. Although $Alg^P(Sig)$ is not closed with respect to pushouts in general, Löwe gave a pushout construction in the case where Sig contains monadic operators only. Thus, pushouts always exist in graph structures. This result also holds for attributed graph structures as shown in [50].

Now let us prove that attributed port graphs are attributed graph structures, and under certain conditions explained below, port graph rewriting corresponds to the SPO transformation of attributed graphs.

Proposition 1. *Attributed port graphs are attributed graph structures.*

Proof. Consider the signature PGS defined by $Sorts = \{node, port, edge, rec\}$ and a set $PGOp$ of operators:

$$\begin{aligned}
&s, t : edge \mapsto port\\
&ports : node \mapsto list[port]\\
&l_V : rec \mapsto node\\
&l_P : rec \mapsto port\\
&l_E : rec \mapsto edge
\end{aligned}$$

Since all the operators in $PGOp$ are unary, PGS is a graph structure.

Now, let us consider the signature $SIG = (\mathcal{S} \cup \{attribute, value, pair, record\}, \mathcal{O}p \cup \{:=, \cdot, \{,\}, \{\}\})$. Here $\mathcal{S}$ is a set of data sorts (subsorts of *value*) and the other sorts in SIG are used to build record structures; Op is the set of operators on data sorts (for example, list constructors, numeric constants, arithmetic operators, etc.) and the other operators in SIG are used to build records:

$$\begin{aligned}
\_ := \_ &: attribute, value \mapsto pair\\
\_ \cdot \_ &: record, attribute \mapsto value\\
\{\_, \_\} &: pair, record \mapsto record\\
\{\} &: \mapsto record
\end{aligned}$$

Let us define $S = \{rec\} \subset Sorts$ and a SIG-attribution of PGS, $ATTROP$, such that $ATTROP_{rec} : rec \mapsto record$. An attributed port graph $G = (V, P, E, D)_{\mathcal{F}}$ can be seen as an algebra on the signature $ATTR = PGS + SIG + ATTROP$ (and therefore as an attributed graph structure, see Definition 8) as follows:

– the sets V, P, E are the carriers of the sorts *node*, *port*, *edge*, respectively, and the sort *rec* is interpreted by a set of pointers, one for each element in V, P, E;
– the functions *Connect*, *Attach* and $\mathcal{L}$ provide interpretations for the operators s, t, *ports*, l_V, l_P, l_E in PGS:

$$\begin{array}{lll}
s, t : edge \mapsto port & \textit{such that} & \\
s(e) := p_1, t(e) := p_2 & \text{iff} & e \in E \wedge Connect(e) = (p_1, p_2)\\
ports : node \mapsto list[port] & \textit{such that} & \\
ports(n) := [p_1 \dots p_n] & \text{iff} & p_1, \dots, p_n \in P \wedge Attach(p_i) = n\\
 & & (1 \le i \le n)\\
l_V : rec \mapsto node & \textit{such that} & \\
l_V(r) := n & \text{iff} & \mathcal{L}(n) = r\\
l_P : rec \mapsto port & \textit{such that} & \\
l_P(r) := p & \text{iff} & \mathcal{L}(p) = r\\
l_E : rec \mapsto edge & \textit{such that} & \\
l_E(r) := e & \text{iff} & \mathcal{L}(e) = r
\end{array}$$

– the sub-algebra corresponding to SIG defines the interpretation of the sort record: it may be either a term algebra (when records are terms in $T(SIG, \mathcal{X})$) or an implementation of records as concrete objects otherwise (i.e. a SIG-algebra).

Let us consider now port graph rewrite rules where the arrow node contains only bridge ports that map one port in the left-hand side to *one* port in the right-hand side. We call this kind of rules *simple port graph rules.* In this case, the arrow node defines a partial PGS-morphism between the left and right-hand side of the rule. We can show that a rewriting step is indeed the pushout defined by the arrow node morphism and the matching morphism.

Proposition 2 (SPO rewriting of port graphs). *Simple port graph rewrite rules are SPO transformation rules and the application of a rule r with a matching morphism m is the pushout of r and m.*

More precisely, consider the signature $ATTR = PGS + SIG + ATTROP$ defined in the proof of Proposition 1, then:

(1) For any port graph morphism f as in Definition 4, there exists a corresponding morphism $f1, f2$ between PGS-graphs with SIG-attribution.
(2) A simple port graph rewrite rule is an $ATTR$-morphism r whose SIG component is an isomorphism.
(3) A match m between the left-hand side of a rule, L, and a redex in a port graph G is an $ATTR$-morphism whose PGS-component is total.
(4) The application of the rule r to G at a redex $m(L)$ is the pushout of r and m.

Proof. We prove each part in turn.

(1) In a port graph morphism $f\colon G \mapsto H$ as in Definition 4, the family of injective functions $\langle f_V\colon V_G \mapsto V_H, f_P\colon P_G \mapsto P_H, f_E\colon E_G \mapsto E_H, f_D\colon D_G \mapsto D_H\rangle$ defines a partial PGS-morphism $f1\colon (G)_{PGS} \mapsto (H)_{GS}$, which coincides with $\langle f_V\colon V_G \mapsto V_H, f_P\colon P_G \mapsto P_H, f_E\colon E_G \mapsto E_H\rangle$ on the carriers of sorts $node, port, edge$. $f1$ preserves operators s, t on edges, and $ports$ on nodes. This is ensured by the two first conditions on *Attach* and *Connect* in Definition 4.
The total SIG-morphism $f2\colon (G)_{SIG} \mapsto (H)_{SIG}$ is the restriction of f on records, i.e. coincides with $f_D\colon D_G \mapsto D_H$. $f2$ must satisfy $\forall attr\colon s \mapsto s' \in ATTROP$, and all $x \in G(f_1)_s$, $f_2(attr^G(x)) = attr^H(f_1(x))$. This is ensured by the condition on $\mathcal{L}$ in Definition 4:
For all $n \in Dom(f)$, $f_D(\mathcal{L}_G(n)) = \mathcal{L}_H(f_V(n))$
For all $p \in Dom(f)$, $f_D(\mathcal{L}_G(p)) = \mathcal{L}_H(f_P(p))$
For all $e \in Dom(f)$, $f_D(\mathcal{L}_G(e)) = \mathcal{L}_H(f_E(e))$.
(2) Since the port graph rewrite rule is simple, the bridge ports in the arrow node map a port from L to at most one port in R, thus defining a partial function r from P_L to P_R (only defined on ports in L that are connected to bridge ports in the arrow node). Hence, r is also a partial PGS-morphism (trivially, since there are no operators in PGS acting on ports). Also, since records in L and R are implemented in the same way, r can be extended as an $ATTR$-morphism where the SIG component is an isomorphism.
(3) This point is a direct consequence of the definition of match (Definition 6), which requires m to be a total morphism, and the definition of morphism in

port graphs (Definition 4), which requires preservation of the graph structure and data values.

(4) To prove the last point, we remark that the construction of the pushout object specified in [50] corresponds to the steps described in Definition 7. The gluing object consists of the set of ports in L that are connected to bridge ports. The pushout object is isomorphic to G where $m(L)$ is replaced with $m(R)$ and the rewriting specified in Definition 7 for bridge ports implements the morphisms on edges, m_r and r_m.

In [49], in order to relate single pushout and double pushout approaches, three conditions are introduced; they actually hold for simple port graph rewrite rules.

(1) All redexes are *conflict free*.
The notion of conflict free redex ensures that if two elements of L have the same image in the matching morphism, then they map to the same element in R. Since our morphims are injective, this condition is trivially satisfied.
(2) All matching morphims are *d-injective*.
This condition is weaker than the injectivity condition we impose (it is trivially satisfied by our matching morphisms).
(3) All matching morphims are *d-complete*.
In the context of simple port graphs, this condition requires that if an edge in G is connected to the image of a port in L which is not linked to R by a bridge port, then the edge is in the image of L. In other words, if a port is not in the domain of the rule morphism, then it is saturated, which is guaranteed in our matching morphisms.

According to [49], if the port graph G can be transformed to H with a rule $r\colon L \mapsto R$ with a matching morphism $m\colon L \mapsto G$, such that m is d-injective and d-complete, the translation of r to a double pushout rule is applicable to G with m in the double pushout framework and gives the same object H.

Summarising, simple port graph rewrite rules define a rewriting relation that corresponds exactly to the SPO semantics and can be translated to the DPO framework.

More general port graph rewrite rules that include bridge ports with connections 1 to n, blackholes and wire ports in the arrow node are more permissive, but then rewriting steps do not correspond directly to the SPO construction. However, coming back to the conditions of [49], we can note that always redexes are *conflict free* and matching morphims are *d-complete*. Our definition of rule requires elements of G that are going to be deleted to be explicitly specified *in the rule*, in L together with the arrow node. More precisely, if a redex is not d-complete, there exists a port which is not in the domain of the rule morphism; then the port is either connected to a black hole or wire port in the arrow node, or it must be saturated. We use the arrow node and the edges that link it to L and R to provide a full description of the elements that would be automatically deleted according to the SPO semantics. Thus our definition of port graph rewriting still ensures control over the elements deleted during SPO rewriting.

2.5 Derivation tree and strategies

A *derivation*, or computation, is a sequence $G \to_{\mathcal{R}}^{*} G'$ of rewriting steps. Each rewriting step involves the application of a rule at a specific position in the graph. In this section, we formalise the notion of a derivation tree and describe how *strategies* can be used to specify the rewriting steps of interest, selecting branches in the derivation tree.

Definition 9 (Derivation tree). *Given a port graph G and a set of port graph rewrite rules $\mathcal{R}$, the derivation tree of G, written $DT(G, \mathcal{R})$, is a labelled tree such that the root is labelled by the initial port graph G, and its children are all the derivation trees $DT(G_i, \mathcal{R})$ such that $G \to_{\mathcal{R}} G_i$. The edges of the derivation tree are labelled with the rewrite rule and the morphism used in the corresponding rewriting step.*

A derivation tree may be infinite, if there is an infinite reduction sequence out of G.

This notion of derivation tree is a particular instance of the more general notion in *Abstract Reduction System (ARS)* [70]. *Abstract strategies* are defined in [42] as a set of derivations of an abstract reduction system. In a more operational way, an *intentional strategy* is defined in [13] as a partial function that associates to a reduction-in-progress, the possible next steps in the reduction sequence, depending on the current state and the derivation so far. This notion of strategy coincides with the definition of strategy in sequential path-building games and in term rewriting systems. A challenge to address is to define languages for describing those intentional strategies. We propose in the following section such a language in the case of port graph rewriting.

3 Strategic graph programs

In this section, we introduce the concept of *strategic graph program*, consisting of a *located graph* (a port graph with two distinguished subgraphs that specify the locations where rewriting should take place or not), a set of *located rewrite rules*, and a *strategy expression*. We then propose a strategy language to define those strategy expressions. In addition to the well-known constructs to select rewrite rules, the strategy language provides position primitives to select or ban specific positions in the graph for rewriting. The latter is useful to program graph traversals in a concise and natural way, and is a distinctive feature of the language.

3.1 Located graphs and rewrite rules

Definition 10 (Located graph). *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*.*

In a located graph G_P^Q, P represents the subgraph of G where rewriting steps may take place (i.e., P is the focus of the rewriting) and Q represents the subgraph of G where rewriting steps are forbidden. We give a precise definition below; the intuition is that subgraphs of G that overlap with P may be rewritten, if they are outside Q. The subgraph P generalises the notion of rewrite position in a term: if G is the tree representation of a term t then we recover the usual notion of rewrite position p in t by setting P to be the node at position p in the tree G, and Q to be the part of the tree above P (to force the rewriting step to apply from P downwards).

We could restrict P and Q to sets of nodes only, but by allowing edges too we obtain a more expressive formalism, which allows us, for instance, to define located graphs where specific edges should be rewritten (i.e., when they are in the position subgraph).

When applying a port graph rewrite rule, not only the underlying graph G but also the position and banned subgraphs may change. A *located rewrite rule*, defined below, specifies two disjoint subgraphs M and N of the right-hand side R that are used to update the position and banned subgraphs, respectively. If M (resp. N) is not specified, R (resp. the empty graph $\emptyset$) is used as default. Below, we use the operators $\cup, \cap, \setminus$ to denote union, intersection and complement of port graphs. These operators are defined in the natural way on port graphs considered as sets of nodes, ports and edges.

Definition 11 (Located rewrite rule). *A* located rewrite rule *is given by a port graph rewrite rule $L \Rightarrow_C R$, and optionally a subgraph W of L and two disjoint subgraphs M and N of R. It is denoted $L_W \Rightarrow_C R_M^N$ (or simply $L_W \Rightarrow R_M^N$). We write $G_P^Q \rightarrow^g_{L_W \Rightarrow_C R_M^N} {G'}_{P'}^{Q'}$ and say that the located graph G_P^Q* rewrites to ${G'}_{P'}^{Q'}$ using $L_W \Rightarrow_C R_M^N$ at position P avoiding Q, *if $G \rightarrow_{L \Rightarrow R} G'$ with a matching morphism g 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 $Q' = (Q \setminus g(L)) \cup g(N)$; if M (resp. N) are not provided then we assume $M = R$ (resp. $N = \emptyset$).*

In general, for a given located rule $L_W \Rightarrow_C R_M^N$ and located graph G_P^Q, more than one matching morphism g, such that $g(L) \cap P = g(W)$ and $g(L) \cap Q$ is empty, may exist (i.e., several rewriting steps at P avoiding Q may be possible). Thus, the application of the rule at P avoiding Q produces a *set of located graphs.*

3.2 Abstract Syntax for Strategies

To control the application of the rules, we introduce a strategy language with the syntax shown in Table 1. For the sake of simplicity, below, we detail the abstract syntax of our strategy language. The concrete syntax usable by programmers can be found at `http://porgy.labri.fr/strat_examples`.

Strategy expressions are generated by the grammar rules from the non-terminal S. A strategy expression combines applications of located rewrite rules, generated by the non-terminal A, and position updates, generated by the non-terminal U, using *focusing expressions* generated by F.

Some of the strategy constructs are strongly inspired from term rewriting languages such as ELAN [12], Stratego [74] and Tom [8]. Focusing operators are not present in term rewriting languages (although they rely on implicit traversal strategies).

The direct management of positions in strategy expressions, via the distinguished subgraphs P and Q in the target graph and the distinguished graphs M and N in a located port graph rewrite rule are original features of the language and are managed using positioning constructs. The syntax presented here extends the one in [30] by including a language to define subgraphs of a given graph by specifying simple properties, expressed with attributes of nodes, edges and ports.

We start by defining the Rule constructs, which specify how to apply rules, then Position constructs, which allow us to specify subgraphs P and Q in a given located graph. We finally define Composition constructs combining strategies.

Rule Constructs. The simplest transformation is a located rule, which can only be applied to a located graph G_P^Q if a part of the redex is in P, and does not involve Q.

- The syntax $T \parallel T'$ represents simultaneous application of the transformations T and T' on disjoint subgraphs of G; it succeeds if both are possible *concurrently*, and it fails otherwise.
- $\texttt{ppick}(T_1, \ldots, T_n, \Pi)$ picks one of the transformations for application, according to the probability distribution Π. If T and T' have respective probabilities π and π', $T \parallel T'$ has probability $\pi \times \pi'$.
- $\texttt{all}(T)$ denotes all possible applications of the transformation T on the located graph at the current position, creating a new located graph for each application. In the derivation tree, this creates as many children as there are possible applications.
- $\texttt{one}(T)$ computes only one of the possible applications of the transformation and ignores the others; more precisely, it makes a choice between all the possible applications, with equal probabilities.

Position Constructs. The grammar for F generates focusing expressions that are used to define positions for rewriting in a graph, or to define positions where rewriting is not allowed. They denote functions used in strategy expressions to change subgraphs P and Q in the current located graph G_Q^P (e.g., to specify graph traversals).

- Focusing constructs.
  - `crtGraph`, `crtPos` and `crtBan`, applied to a located graph G_P^Q, return respectively the whole graph G, P and Q.

Let L, R be port graphs; M, N subgraphs of R; W a subgraph of L; $n, k \in \mathbb{N}$; $\pi_{i=1\ldots n} \in [0,1]$; $\sum_{i=1}^{n} \pi_i = 1$. Let *attribute* be an attribute; e a valid expression; *function* a computable function; *ComputedProbDist* a probability distribution.
$[x]$ means the item x is optional.

	(Probabilities)	$\Pi ::= \{\pi_1, \ldots, \pi_n\} \mid \mathit{ComputedProbDist}$
Rules	**(Transformations)**	$T ::= L_W \Rightarrow_C R_M^N \mid (T \parallel T)$
		$\mid \texttt{ppick}(T_1, \ldots, T_n, \Pi)$
	(Applications)	$A ::= \texttt{all}(T) \mid \texttt{one}(T)$
Positions	**(Focusing)**	$F ::= \texttt{crtGraph} \mid \texttt{crtPos} \mid \texttt{crtBan}$
		$\mid F \cup F \mid F \cap F \mid F \setminus F \mid (F) \mid \emptyset$
		$\mid \texttt{ppick}(F_1, \ldots, F_n, \Pi)$
		$\mid \texttt{property}(F, \mathit{Elem}[, \mathit{Expr}])$
		$\mid \texttt{ngb}(F, \mathit{Elem}[, \mathit{Expr}])$
	(Determining)	$D ::= \texttt{all}(F) \mid \texttt{one}(F)$
	(Updating)	$U ::= \texttt{setPos}(D) \mid \texttt{setBan}(D)$
		$\mid \texttt{update}(\mathit{function})$
Properties	**(Properties)**	$\mathit{Elem} ::= \texttt{node} \mid \texttt{edge} \mid \texttt{port}$
		$\mathit{Expr} ::= \mathit{attribute}\ \mathit{Relop}\ e \mid \mathit{Expr} \&\& \mathit{Expr}$
		$\mathit{Relop} ::= == \mid \neq \mid > \mid <$
		$\mid \geq \mid \leq \mid =\sim$
Compositions	**(Comparison)**	$C ::= F = F \mid F \neq F \mid F \subset F \mid \texttt{isEmpty}(F)$
		$\mid \texttt{match}(T)$
	(Strategies)	$S ::= \texttt{id} \mid \texttt{fail} \mid A \mid U \mid C \mid S;S$
		$\mid \texttt{if}(S)\texttt{then}(S)[\texttt{else}(S)] \mid (S)\texttt{orelse}(S)$
		$\mid \texttt{repeat}(S)[(k)] \mid \texttt{while}(S)\texttt{do}(S)[(k)]$
		$\mid \texttt{try}(S) \mid \texttt{not}(S) \mid \texttt{ppick}(S_1, \ldots, S_n, \Pi)$

Table 1. Abstract Syntax of the Strategy Language.

- $\texttt{property}(F, \mathit{Elem}[, \mathit{Expr}]\}$, applied to a located graph G_P^Q, is used to select elements of G_P^Q filtered by F that satisfy a certain property, specified by *Expr*. It can be seen as a filtering construct: if the expression F defines a subgraph G then $\texttt{property}(F, \mathit{Elem}, \mathit{Expr})$ returns a subgraph G' of G that satisfy the decidable property *Expr*. Depending on the value of *Elem*, the property is evaluated on nodes, ports, or edges, allowing us for instance to select the red nodes and red edges, or nodes with active ports, as shown in examples below. If *Expr* is not specified, all elements are selected.
- $\texttt{ngb}(F, \mathit{Elem}[, \mathit{Expr}]\}$, applied to a located graph G_P^Q, returns a subset of the neighbours (i.e., adjacent nodes) of F according to *Expr*. When

edge is used (i.e., when we write $\texttt{ngb}(F, \texttt{edge}, Expr)$), it returns all the neighbours of F connected to F via edges which satisfy the expression $Expr$. If $Expr$ is not specified, all neighbours are selected.
  - $\cup$, $\cap$ and $\setminus$ are union, intersection and complement of port graphs; $\emptyset$ denotes the empty graph. We assume the usual priorities (e.g., intersection has priority over union) and operations of the same priority are evaluated left to right.
    We can combine multiple Property operators using intersection $\cap$ to filter multiple times. For example, to select the nodes in the subgraph denoted by Pos that are named $Mult$ and that have at least one port named Aux, we write:
    $$\texttt{all}(\texttt{property}(Pos, \texttt{node}, Name == \text{“}Mult''){\cap}$$
    $$\texttt{property}(Pos, \texttt{port}, Name == \text{“}Aux''))$$
  - $\texttt{ppick}(F_1, \ldots, F_n, \Pi)$ picks one of the positions for application, according to the given probability distribution.
- Determining Constructs.
  $\texttt{one}(F)$ returns one node in F chosen at random and $\texttt{all}(F)$ returns the full F.
- Update Constructs.
  - $\texttt{setPos}(D)$ (resp. $\texttt{setBan}(D)$) sets the position subgraph P (resp. Q) to be the graph resulting from the expression D. It always succeeds (i.e., returns id).
  - $\texttt{update}(function\_name)$ updates attributes and their values in the graph using an external function, with given parameters. This is useful to update global properties of the graph, in order to focus on specific nodes: for example, in social networks, to select a “central” node (see Section 4.2). This is also a way of interfacing with another language (e.g. a Python program or a plugin written outside Porgy).

Composition Constructs. The grammar for S involves, beyond previous constructs, an additional class C of comparison constructs, useful for checking conditions.

- Comparison constructs.
  C includes comparison operators for graphs and a matching construct that checks whether a rule matches the current graph.
  - $F = F'$ returns id if both expressions denote isomorphic port graphs (same sets of nodes, ports and edges), otherwise returns fail. $F \mathrel{!=} F'$ returns id if the expressions do not denote isomorphic port graphs, otherwise returns fail. Similarly $F \subset F'$ checks whether F denotes a subgraph of F'. We have also included an additional operation, which, although derivable from the rest of the language, facilitates the implementation: $\texttt{isEmpty}(F)$ returns id if F denotes the empty graph and fail otherwise. It is defined as $F = \emptyset$.

  - `match`(T) returns id if there exists a subgraph isomorphism from the left-hand side of T to the current graph taking into account the current position and banned subgraphs. In other words,
  - `match`(T) can be seen as an abbreviation of `if`(`one`(T))`then`(id)`else`(fail) (see below), but it is directly implemented to improve its efficiency, as explained in Section 6.
- Strategies S are defined with the additional following constructs:
  - id and fail are two atomic strategies that respectively denote success and failure.
  - The expression S;S' represents sequential application of S followed by S'.
  - `if`(S)`then`(S')[`else`(S'')] checks if the application of S on a copy of G_P^Q returns id, in which case S' is applied to the original G_P^Q, otherwise S'' is applied to the original G_P^Q. If S'' is not specified then we consider $S'' =$ id.
  - (S)`orelse`(S') applies S if possible, otherwise applies S'. It fails if both S and S' fail.
  - `repeat`$(S)[(k)]$ simply iterates the application of S until it fails, but, if k is specified between parenthesis, then the number of repetitions cannot exceed k.
  - `while`(S)`do`$(S')[(k)]$ keeps on sequentially applying S' while the expression S succeeds on a copy of the graph. If S fails, then id is returned. If k between parenthesis is specified, then the number of iterations cannot exceed k.
  - `try`(S) behaves like S if S succeeds, but if S fails, it still returns id. It is a derived operation which is defined as (S)`orelse`(id).
  - `not`(S) returns id (resp. fail) if S fails (resp. succeeds). This is also a derivable construct: it is defined as `if`(S)`then`(fail)`else`(id).
  - `ppick`$(S_1, \ldots, S_n, \Pi)$ picks one of the strategies for application, according to the given probability distribution. This construct generalises the probabilistic constructs on rules and positions seen above.

4 Examples

In this section, the expressive power of the language is illustrated through examples taken from three different domains: graph property testing, social network simulation and term rewriting strategies. Working examples can be downloaded from `http://porgy.labri.fr/strat_examples`.

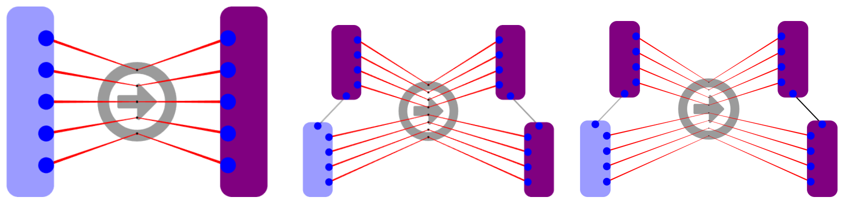
4.1 Graph testing: Connected graph and spanning tree

Strategy 1 below is used to check if a graph is connected (i.e., made of only one connected component). The strategy `pick-one-node` selects a node at random as a starting point and marks it by changing the value of its colour attribute (see Figure 3). Then, rule `walk` is applied as long as possible (`visit-neighbours`).

This rule consists of a pair of nodes linked by an edge, with only one node already visited in the left-hand side and all nodes marked as visited in the right-hand side (the values of colour attributes are changed for both nodes). When the rule `walk` cannot be applied any longer (i.e., there are no connected pairs of nodes where one node has already been visited), the strategy `check-all-nodes-visited` tests if a node can still be chosen with `start` inside the whole graph. If so, the strategy ends on a failure because the graph contains more than one connected component (see Fig. 4-a).

Strategy 1: Strategy to check if a graph is connected.

```
//pick-one-node:
        setPos(all(crtGraph));
        one(start);
//visit-neighbours:
        repeat(one(walk)) ;
//check-all-nodes-visited:
        setPos(all(crtGraph));
        not(one(start))
```



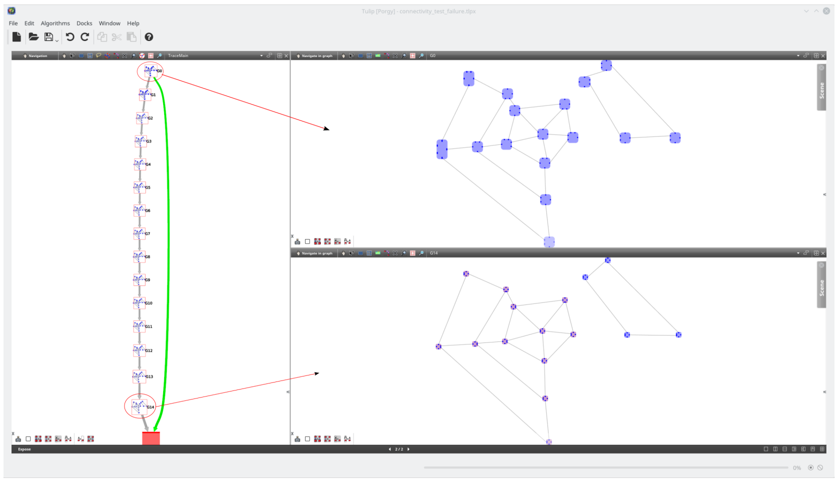
(a) Rule for selecting a starting node.

(b) Rule for computing connected component. The blue node is marked with a change in its colour.

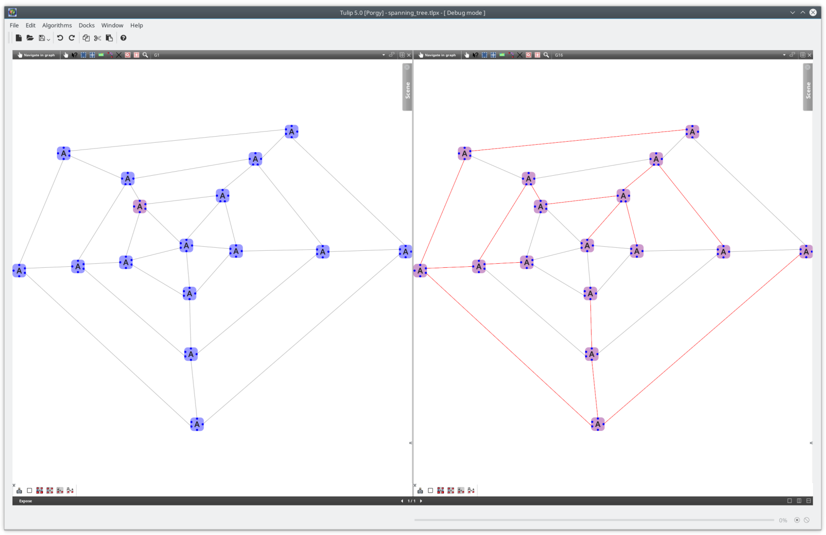
(c) Rule for computing a spanning tree. The difference with (b) is the edge which is also marked.

Fig. 3. Rules used to test if a graph is composed of only one connected component (b) and computing a spanning tree (c) both after choosing a starting node (a). Visited nodes ((b) and (c)) and edges ((c) only) are marked (illustrated by a change in colour).

We can compute a spanning tree from a graph by simply making a small change in the rule `walk`. Instead of changing only the nodes' colour in the right-hand side, we now change the colour of the edge linking the two nodes as well (see Fig. 4-b). To compute spanning trees rooted in each graph node, we simply change `one(start)` to `all(start)` in the `pick-one-node` strategy (see Strategy 2). Fig. 5 shows on another example the obtained derivation tree and a close-up on a branch displayed as a series of small-multiples.



(a) Connected component. One component of the graph is not visited (see the surrounded node at the bottom left and a close-up on the right) so the strategy ends with failure (red node at the bottom of the derivation tree).



(b) Spanning tree. From a node (left), a possible spanning tree (right, red edges).

Fig. 4. A result after running Strategy 1 (a) and the updated version to compute a spanning tree (b).

4.2 Social network behaviour simulation

Social network simulation offers many interesting challenges. We focus here on simulation of acquaintance and influence propagation.

Strategy 2: Computation of a spanning tree from every node of the graph (acting as the root of the tree).

```
//pick-one-node:
          setPos(all(crtGraph));
          all(start);
//visit-neighbours:
          repeat(one(walk)) ;
```

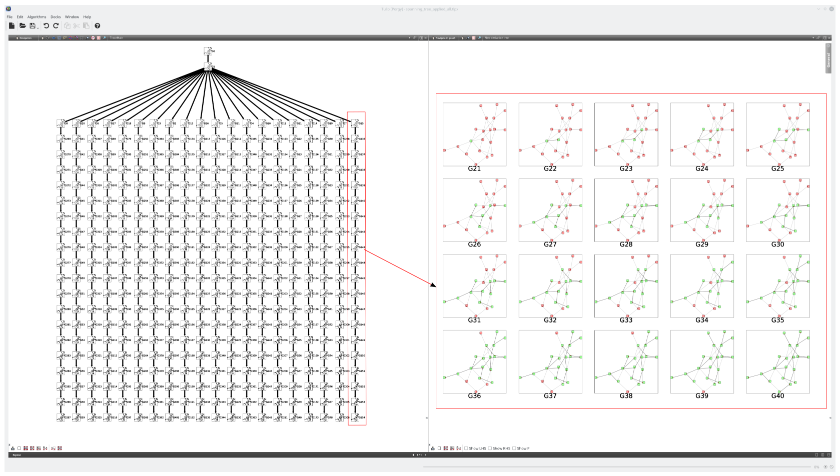


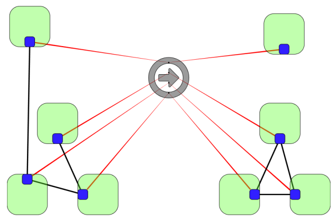
Fig. 5. The left panel shows the whole derivation tree obtained with Strategy 2 on another example. The right panel is a close-up on a branch showed as small-multiples.

Dynamics of acquaintanceship. Rules and strategies to model dynamics in social networks using a probabilistic graph-based approach are presented in [40]. Porgy can be used to formally specify and visualise the dynamics of the model. We illustrate it using the example given in Sect. 3.2 of [40].

This example consists of two rules $R1$ and $R2$ (see Fig. 6). Rule $R1$ shows the evolution to a better acquaintanceship and $R2$ the creation of a stronger relationship by forming triads (three linked vertices). The authors proposed to use a respective application probability of 0.11 and 0.89 for 1,000 iterations (see Strategy 3). From a randomly generated social network (Erdős–Rényi (ER) random graph) with 200 vertices and edges, Fig. 7 shows a possible result.

Strategy 3: Strategy to reproduce the dynamics of acquaintanceship [40].

```
setPos(crtGraph);
repeat(one(ppick(R1, R2, {0.11, 0.89})))(1000)
```


(a) Creation of stronger relationships by forming triads.

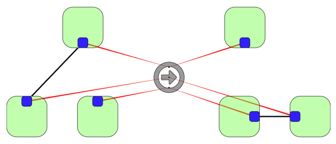

(b) Breaking up of an acquaintanceship for a more advantageous one

Fig. 6. PORGY specification of the two rules presented in [40] to model the dynamics of acquaintanceship in social networks.

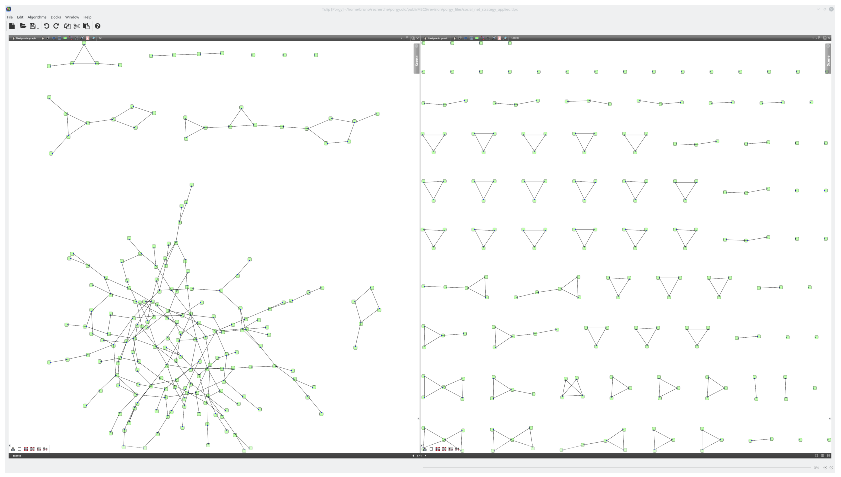
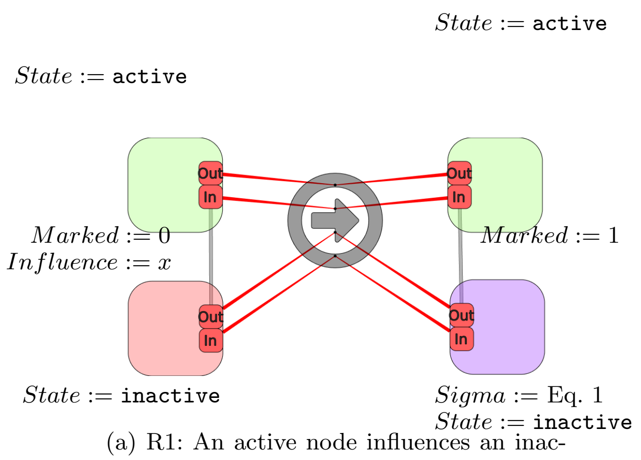


Fig. 7. A random Erdős–Rényi (ER) graph with 200 vertices and edges (left) and a possible result obtained after running Strategy 3 (right).

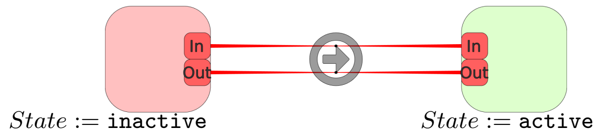
Influence Propagation in Social Networks. This example illustrates how record expressions may be used to compute attribute values and updated through application of rules.

A simplified definition of propagation in a social network is as follows: when individuals perform a specific action like announcing an event, they become active and inform their neighbours of their changing state, thus giving them the possibility to become active if they perform the same action. Such process reiterates as the newly active neighbours share the information with their own neighbours. The activation can thus propagate from peer to peer across the whole network.

A visual approach to compare propagation models in social networks is presented in [73], where graph rewriting is used as a framework to specify and compare several already published propagation models. To express propagation conditions (e.g., a probabilistic model for node activation, or activation after reaching a pre-defined threshold) it is natural to make use of records with expressions, i.e., include specific attributes in rules whose values are numerical expressions.


$State := \texttt{active}$
$State := \texttt{active}$
Out
In
$Marked := 0$
$Influence := x$
$Marked := 1$
Out
In
$State := \texttt{inactive}$
$Sigma :=$ Eq. 1
$State := \texttt{inactive}$

(a) R1: An active node influences an inactive node



(b) R2: an inactive node sufficiently influenced becomes active.

Fig. 8. Rules used to express a simple propagation model (see [73] for more details). Active nodes are depicted in green and visited, not yet active, nodes in purple. Red nodes are inactive (however, they may have been visited already). Rule R1 indicates when an activated node is connected to an inactive node, it tries to influence it. If it succeeds, R2 makes this node active (Strategy 4). Because social networks are by definition oriented networks, ports are named “In” and ”Out” to make orientation clear.

In the example of Fig. 8.:

– Each node n has an attribute $\mathcal{L}(n).State$ which indicates whether it contributes to the propagation or not. It is coupled with the $\mathcal{L}(n).Colour$ attribute. Nodes also have a $\mathcal{L}(n).Sigma$ attribute that measures the maximum influence withstood from its active neighbours until now.
– An edge that connects two ports p, p' in respective nodes n, n' has an attribute $\mathcal{L}(e).Influence$ which indicates the influence of n' (i.e. $\mathcal{L}(p').Attach =$

n') on n (i.e. $\mathcal{L}(p).Attach = n$). It also has a Boolean attribute $\mathcal{L}(e).Marked$, initially false, which is set to true when n has tried to influence n'.
– The node's attribute $\mathcal{L}(n).Sigma$ is initialised with value 0 and then updated using the formula

$$\mathcal{L}(n).Sigma = \max\left(\frac{\mathcal{L}(e).Influence}{r}, \mathcal{L}(n).Sigma\right) \tag{1}$$

where r is a random value between 0 and 1. The formula is stored in rule (R1) via a dedicated graphical user interface.

When this rule is applied on a pair of nodes active(n)/inactive($\overline{n}$) (green/red):

(a) we generate a random number $r \in]0, 1]$;
(b) we store in the attribute $\mathcal{L}(\overline{n}).Sigma$ the new value of $Sigma$ computed with Formula 1; and,
(c) using the $Marked$ attribute, we mark the edge e linking n to $\overline{n}$ to prevent the selection of this particular pair in the next pattern matching searches. This ensures that the active node n will not be able to try to influence the same node $\overline{n}$ over and over.

Once all pairs of active/inactive neighbours have been tried, if $\overline{n}$ is sufficiently influenced (i.e. $\mathcal{L}(\overline{n}).Sigma \geq 1$), it becomes active with rule (R2). This behaviour is summarised in Strategy 4. This strategy works as a wave. From a starting set of active nodes, a first run of the strategy influences their direct neighbours. The strategy has to be repeated again to allow the newly active nodes to influence their neighbours. The starting set of active nodes can be chosen randomly or, for instance, by calling an external procedure with `update()` to compute the network central nodes.

Strategy 4: Simple influence propagation in social network. This strategy has to be applied after defining a starting set of actives nodes.

```
//Influence a maximum number of nodes:
        repeat(one($R1$));
//select all nodes which should become active:
        setPos(all(property(crtGraph, node, $Sigma \geq 1$)));
//Make them active:
        repeat(one($R2$))
```

4.3 Term rewriting strategies

Using focusing (specifically the `Property` construct), we can easily define concise strategy expressions that implement standard term rewriting strategies. Below we show how to implement outermost and innermost term rewriting with a given rule: to change from innermost to outermost rewriting for instance, we simply change the strategy, not the rewrite rule. This is standard in term-based languages such as Elan [12] or Stratego [16, 74]; in Porgy this idea has been generalised to graphs that are not necessarily trees, via the notion of position graph.

Let us first consider a representation of terms as trees using port graphs. Recall that a tree is a graph in which any two vertices are connected by exactly one path. We can represent terms as port graphs where each node has a name corresponding to its function symbol, a port named *Parent* that connects it to the parent node (if the parent exists, that is, if the node is not the root), and a set of ports named *Child1*, ..., *Childn*, that connect it to the children. We use the *Arity* attribute of ports to identify the root of the tree (where the port *Parent* does not have any incident edges). *Arity* is an attribute of ports that Porgy keeps up to date automatically.

Outermost rewriting on trees. The focusing expression

$$start \triangleq \texttt{property}(\texttt{crtGraph}, \texttt{port}, Name == \text{“}Parent''\&\&Arity == 0)$$

selects the subgraph containing the root of the tree (the root is the only node where the parent port has arity 0, that is, the only node without a parent node).

The strategy for outermost rewriting with a rule R is presented in Strategy 5.

Strategy 5: Outermost rewriting on trees.

```
setPos(all($start$));
while(not(isEmpty(crtPos)))do(
        if(match($R$))then(
            one($R$); setPos(all($start$))
        )else(
            setPos(all(ngb(crtPos, port, $Name =\sim$ “$\hat{}Child[1-9]\$''$)))
        )
)
```

The strategy starts by focusing on the root node, using $\texttt{setPos}(\texttt{all}(start))$ which in this case is equivalent to $\texttt{setPos}(\texttt{one}(start))$. If R applies, the position is set back to the root of the new term. Otherwise, $\texttt{setPos}(\texttt{all}(\texttt{ngb}(\texttt{crtPos}, \texttt{port}, Name =\sim$ “$\hat{}Child[0-9]\$''$))) goes one level down into the tree, taking all children of nodes in the current position as the new current position.

Innermost rewriting on trees. We define the focusing expressions *start* and *NonLeaf* to select the leaves and the rest of the tree, respectively.

$$NonLeaf \triangleq \texttt{property}(\texttt{crtGraph}, \texttt{port}, Name =\sim \text{“}\hat{}Child[1-9]\$'')$$
$$start \triangleq \texttt{crtGraph} \setminus NonLeaf$$

The strategy for innermost rewriting with a rule R is presented in Strategy 6.

Strategy 6: Innermost rewriting on trees.

```
setPos(all($start$)); setBan(all($NonLeaf$));
while(not(isEmpty(crtPos)))do(
        if(match($R$))then(
            one($R$); setPos(all($start$)); setBan(all($NonLeaf$))
        )else(
            setPos(all(ngb(crtPos, port, $Name ==$ “$Parent$”)));
            setBan(all(crtBan \ crtPos))
        )
)
```

The initial position contains the leaves of the tree. Thus, if R can be applied then we apply it and set the position back to the leaves of the tree and put all other elements of the tree into the banned subgraph. Otherwise, we move one level up in the tree with setPos(all(ngb(crtPos, port, $Name ==$ “$Parent$”))) and the banned subgraph is updated again to all remaining elements of the tree (with setBan(all(crtBan $\setminus$ crtPos))).

5 Semantics of strategic graph programs

We are now ready to formally define strategic graph programs and their semantics.

Definition 12 (Strategic graph program). *A* strategic graph program *consists of a finite set of located rewrite rules $\mathcal{R}$, a strategy expression $S_{\mathcal{R}}$ (built from $\mathcal{R}$ using the grammar in Table 1) and a located graph G_P^Q. We denote it $\left[S_{\mathcal{R}}, G_P^Q\right]$, or simply $\left[S, G_P^Q\right]$ when $\mathcal{R}$ is clear from the context, or $[S, G]$ when positions are implicit.*

Intuitively, a strategic program consists of an initial port graph, together with a set of rules that is used to reduce it, following the given strategy.

Formally, the semantics of a strategic graph program $\left[S, G_P^Q\right]$ is specified using a transition system, defining a *small step* operational semantics in SOS style [61]. The idea is to use the strategy expression S to decide which rewriting steps should be performed on G. In general, there may be more than one way of rewriting a port graph according to S, so we have a set of possible rewritings at each step.

In order to keep track of the various rewriting alternatives, we introduce the notion of *configuration.* A configuration $\{O_1, \ldots, O_k, \ldots, O_n\}$ is a multiset of graph programs corresponding to nodes in the derivation tree generated from the initial graph program.

Definition 13. *A* configuration C *is a multiset* $\{O_1, \ldots, O_n\}$ *where each* O_i *is a strategic graph program.*

Definition 14 (Transitions). *The transition relation* $\longmapsto$ *is a binary relation on configurations defined as follows:*

$$\{O_1, \ldots, O_{k-1}, O_k, O_{k+1}, \ldots, O_n\} \longmapsto \{O_1, \ldots, O_{k-1}, O'_{k_1}, \ldots, O'_{k_m}, O_{k+1}, \ldots, O_n\}$$

if $O_k \rightarrowtail \{O'_{k_1}, \ldots, O'_{k_m}\}$ *(*$1 \leq k \leq n$*).*

The transition relation $\rightarrowtail$ is defined below in Section 5.1 using axioms and rules. It is extended to take into account probabilistic strategies in Section 5.2.

Given a configuration $\{O_1, \ldots, O_k, \ldots, O_n\}$, there may be several values of k such that a $\rightarrowtail$-transition can be applied to the strategic program O_k, so the relation $\longmapsto$ on configurations could have been defined as a parallel reduction relation (performing reductions in parallel at all the positions in the configuration where a $\rightarrowtail$-transition is possible). However, we prefer to define independent transitions for each graph program in the configuration, reducing the O_i one-by-one (see Definition 14), because we associate each graph program with a node in the derivation tree. Intuitively, starting with a configuration $[S, G]$, the transition relation builds configurations which embed the derivation tree of G. One can recover it by projecting a strategic program $O = [S, G]$ on its second component G and by associating to a $\rightarrowtail$-step $O_k \rightarrowtail \{O'_{k_1}, \ldots, O'_{k_m}\}$, for $1 \leq k \leq n$, a set of m reduction steps $G_k \rightarrow_{\mathcal{R}} G'_{k_i}$ for $1 \leq i \leq m$. This is how PORGY builds and displays a derivation tree.

In our implementation the choice of k, that is the object where the transition applies, is done either by the user interactively, by clicking on a node of the derivation tree, or automatically (left to right, as described in [29]).

We give the transition rules below, first for the core sublanguage, and then for the whole language including probabilistic constructs. We type variables in transition rules by naming them as the initial symbol of the corresponding grammar with an index number if needed (for example: F_2 represents a focusing expression; S_3 represents a strategy expression).

5.1 Core sublanguage

We start by considering the strategy sublanguage that does not include one nor ppick.

The transition relation $\rightarrowtail$ on individual strategic graph programs is defined by induction, for each construct of the strategy language (see Table 1).

Rule Constructs. Let us consider the strategies `all`(T) and `all`$(T||T)$. In order to formally define the operator all, we first define the set of *legal reducts* for a rule on a located graph.

Definition 15. *The* set of legal reducts of G_P^Q for $L_W \Rightarrow R_M^N$, *or* legal set *for short, denoted* $LS_{L_W \Rightarrow R_M^N}(G_P^Q)$*, contains all the located graphs* ${G_i}_{P_i}^{Q_i}$ *(*$1 \leq i \leq k$*) such that* $G_P^Q \rightarrow^{g_i}_{L_W \Rightarrow R_M^N} {G_i}_{P_i}^{Q_i}$ *and* $g_1, \ldots, g_k$ *are pairwise different.*

The transition rules for rewrite rule application are then as follows:

$$\frac{}{[\mathtt{all}(L_W \Rightarrow R^N_M), G^Q_P] \rightarrowtail \{[\mathsf{id}, {G_1}^{Q_1}_{P_1}], \ldots, [\mathsf{id}, {G_k}^{Q_k}_{P_k}]\}}\ {\scriptstyle if\ LS_{L_W \Rightarrow R^N_M}(G^Q_P) = \{{G_1}^{Q_1}_{P_1}, \ldots, {G_k}^{Q_k}_{P_k}\}}$$

$$\frac{}{[\mathtt{all}(L_W \Rightarrow R^N_M), G^Q_P] \rightarrowtail \{[\mathsf{fail}, G^Q_P]\}}\ {\scriptstyle if\ LS_{L_W \Rightarrow R^N_M}(G^Q_P) = \emptyset}$$

As the name of the operator indicates, all possible applications of the rule are considered. The strategy fails if the rule is not applicable.

Parallel application of two rewrite rules is achieved through the operator $||$, which works on rules only (not on general strategies). To define the semantics of $\mathtt{all}({L_1}_{W_1} \Rightarrow {R_1}^{N_1}_{M_1} || \ldots || {L_k}_{W_k} \Rightarrow {R_k}^{N_k}_{M_k})$, we define a new rule $(L_1 \cup \cdots \cup L_k)_{W_1 \cup \ldots \cup W_k} \Rightarrow (R_1 \cup \cdots \cup R_k))^{N_1 \cup \cdots \cup N_k}_{M_1 \cup \cdots \cup M_k}$, where the arrow node contains all the ports and edges of the arrow nodes of individual rules ${L_i}_{W_i} \Rightarrow {R_i}^{N_i}_{M_i}$ (for $1 \leq i \leq k$) and the conjunction of their conditions. Note that the union operator works on graphs, so even if two nodes have the same name, if they are different nodes, then the union contains both nodes. We need to generalise the notion of legal reduct to ensure simultaneous application of rules at disjoint redexes that superpose with P.

The *set of legal parallel reducts* of G^Q_P for

$$ {L_1}_{W_1} \Rightarrow {R_1}^{N_1}_{M_1} || \ldots || {L_k}_{W_k} \Rightarrow {R_k}^{N_k}_{M_k}$$

or *legal parallel set*, denoted $LPS_{{L_1}_{W_1} \Rightarrow {R_1}^{N_1}_{M_1} || \ldots || {L_k}_{W_k} \Rightarrow {R_k}^{N_k}_{M_k}}(G^Q_P)$, contains all the located graphs ${G_i}^{Q_i}_{P_i}$ $(1 \leq i \leq n)$ such that $G^Q_P \rightarrow^{g_i}_{(L_1 \cup \cdots \cup L_k)_{W_1 \cup \ldots \cup W_k} \Rightarrow (R_1 \cup \cdots \cup R_k))^{N_1 \cup \cdots \cup N_k}_{M_1 \cup \cdots \cup M_k}} {G_i}^{Q_i}_{P_i}$, $g_1, \ldots, g_n$ are pairwise different, and each g_i $(1 \leq i \leq n)$ is defined as the union of k morphisms g_{ij} $(1 \leq j \leq k)$ such that g_{ij} has as domain L_j and $g_{ij}(L_j) \cap P = g_{ij}(W_j)$. Now we can define the axioms for $\mathtt{all}({L_1}_{W_1} \Rightarrow {R_1}^{N_1}_{M_1} || \ldots || {L_k}_{W_k} \Rightarrow {R_k}^{N_k}_{M_k})$.

$$\frac{}{[\mathtt{all}({L_1}_{W_1} \Rightarrow {R_1}^{N_1}_{M_1} || \ldots || {L_k}_{W_k} \Rightarrow {R_k}^{N_k}_{M_k}), G^Q_P] \rightarrowtail \{[\mathsf{id}, {G_1}^{Q_1}_{P_1}], \ldots, [\mathsf{id}, {G_k}^{Q_k}_{P_k}]\}}\ (\star)$$

$(\star)$ if $LPS_{{L_1}_{W_1} \Rightarrow {R_1}^{N_1}_{M_1} || \ldots || {L_k}_{W_k} \Rightarrow {R_k}^{N_k}_{M_k}}(G^Q_P) = \{{G_1}^{Q_1}_{P_1}, \ldots, {G_k}^{Q_k}_{P_k}\}$

$$\frac{}{[\mathtt{all}({L_1}_{W_1} \Rightarrow {R_1}^{N_1}_{M_1} || \ldots || {L_k}_{W_k} \Rightarrow {R_k}^{N_k}_{M_k}), G^Q_P] \rightarrowtail \{[\mathsf{fail}, G^Q_P]\}}\ (\star\star)$$

$(\star\star)$ if $LPS_{{L_1}_{W_1} \Rightarrow {R_1}^{N_1}_{M_1} || \ldots || {L_k}_{W_k} \Rightarrow {R_k}^{N_k}_{M_k}}(G^Q_P) = \emptyset$.

If all the sets W_i in the rules participating in the parallel construct are non empty, then

$$[\mathtt{all}({L_1}_{W_1} \Rightarrow {R_1}^{N_1}_{M_1} || \ldots || {L_k}_{W_k} \Rightarrow {R_k}^{N_k}_{M_k}), G^Q_P] \rightarrowtail C$$

if $[\mathtt{all}((L_1 \cup \cdots \cup L_k)_{W_1 \cup \ldots \cup W_k} \Rightarrow (R_1 \cup \cdots \cup R_k))^{N_1 \cup \cdots \cup N_k}_{M_1 \cup \cdots \cup M_k}, G^Q_P] \rightarrowtail C$. However, if some or all of the W_i are empty, the computation of the set of legal parallel reducts cannot be avoided, since it is necessary to check that all the left-hand sides have a non-empty intersection with P.

Position Constructs. The commands that are used to specify and update positions are `setPos, setBan`, and `update`. The first two rely on focusing expressions, generated by the grammar for the non terminal F (see Tab. 1), which have a functional semantics. In other words, an expression F denotes a function that applies to the current located graph G^Q_P, and computes a subgraph of G. Since there is no ambiguity, the function denoted by the expression F will also be called F. We define it below.

$\mathtt{crtGraph}(G^Q_P)$	$= G$	
$\mathtt{crtPos}(G^Q_P)$	$= P$	
$\mathtt{crtBan}(G^Q_P)$	$= Q$	
$\mathtt{property}(F, Elem[, Expr])(G^Q_P)$	$= G'$	where G' consists of all nodes in $F(G^Q_P)$ satisfying Expr (if not given, *Expr* is considered to be always true)
$\mathtt{ngb}(F, Elem[, Expr])(G^Q_P)$	$= G'$	where G' consists of a subset of the nodes adjacent to nodes in $F(G^Q_P)$: *Expr* is evaluated on the elements *Elem* of $F(G^Q_P)$ to compute the adjacent nodes. If *Expr* is not given, it is considered to be always true
$(F_1\ op\ F_2)(G^Q_P)$	$= F_1(G^Q_P)\ op\ F_2(G^Q_P)$ where op is $\cup, \cap, \setminus$	

The scope constructs $\mathtt{all}(F)$ and $\mathtt{one}(F)$ return respectively the whole subgraph computed by F and one randomly chosen node in the subgraph computed by F. Below, we give the semantics of $\mathtt{setPos}(D)$, $\mathtt{setBan}(D)$ and $\mathtt{update}(function\_name)$. Since we are dealing with the deterministic sublanguage, we assume the commands use $\mathtt{all}(F)$ (we treat non-deterministic commands in the next section).

$$\frac{}{[\mathtt{setPos}(\mathtt{all}(F)), G^Q_P] \rightarrowtail \{[\mathsf{id}, G^Q_{P'}]\}}\ \textit{if } F(G^Q_P) = P'$$

$$\frac{}{[\mathtt{setBan}(\mathtt{all}(F)), G^Q_P] \rightarrowtail \{[\mathsf{id}, G^{Q'}_P]\}}\ \textit{if } F(G^Q_P) = Q'$$

$$\frac{}{[\mathtt{update}(f), G^Q_P] \rightarrowtail \{[\mathsf{id}, \overline{G}^{Q'}_{P'}]\}}\ \textit{if } f(G^Q_P) = \overline{G}^{Q'}_{P'}$$

The located graph $\overline{G}^{Q'}_{P'}$ has the same structure as G^Q_P, but attributes and their values may have changed, as well as position and banned subgraphs.

Note that with the semantics given above for `setPos()` and `setBan()`, it is possible for P and Q to have a non-empty intersection. A rewrite rule can still apply if the redex overlaps P but not Q.

Composition constructs.

- **Comparison constructs.** For every construct C of the form F `op` F' (see the grammar for C in Table 1), there are two rules:

$$\frac{F(G_P^Q)\ \texttt{op}\ F'(G_P^Q) = true}{[C, G_P^Q] \rightarrowtail \{[\mathsf{id}, G_P^Q]\}} \quad \frac{F(G_P^Q)\ \texttt{op}\ F'(G_P^Q) = false}{[C, G_P^Q] \rightarrowtail \{[\mathsf{fail}, G_P^Q]\}}$$

  As mentioned earlier, $\texttt{isEmpty}(F)$ is defined as $F = \emptyset$ and $\texttt{match}(T)$ has the same semantics as $\texttt{if}(\texttt{one}(T))\texttt{then}(\mathsf{id})\texttt{else}(\mathsf{fail})$, defined below.
- There are no axioms/rules defining transitions for a program where the strategy is id or fail (these are terminal).
- **Sequence.** The semantics of sequential application, written $S_1; S_2$, is defined by two axioms and a rule:

$$\frac{}{[\mathsf{id}; S, G_P^Q] \rightarrowtail \{[S, G_P^Q]\}} \quad \frac{}{[\mathsf{fail}; S, G_P^Q] \rightarrowtail \{[\mathsf{fail}, G_P^Q]\}}$$

$$\frac{[S_1, G_P^Q] \rightarrowtail \{[S_1^1, G_1{}_{P_1}^{Q_1}], \ldots, [S_1^k, G_k{}_{P_k}^{Q_k}]\}}{[S_1; S_2, G_P^Q] \rightarrowtail \{[S_1^1; S_2, G_1{}_{P_1}^{Q_1}], \ldots, [S_1^k; S_2, G_k{}_{P_k}^{Q_k}]\}}$$

  The rule for sequences ensures that S_1 is applied first.
- **Conditional.** The behaviour of the strategy $\texttt{if}(S_1)\texttt{then}(S_2)[\texttt{else}(S_3)]$ depends on the result of the strategy S_1. If S_1 succeeds on a copy of the current located graph, then S_2 is applied to the current located graph, otherwise S_3 is applied. If S_3 is not specified, we consider $S_3 = \mathsf{id}$.

$$\frac{\exists G', M\ s.t.\ \{[S_1, G_P^Q]\} \longmapsto^* \{[\mathsf{id}, G'], M\}}{[\texttt{if}(S_1)\texttt{then}(S_2)\texttt{else}(S_3), G_P^Q] \rightarrowtail \{[S_2, G_P^Q]\}}$$

$$\frac{\nexists G', M\ s.t.\ \{[S_1, G_P^Q]\} \longmapsto^* \{[\mathsf{id}, G'], M\}}{[\texttt{if}(S_1)\texttt{then}(S_2)\texttt{else}(S_3), G_P^Q] \rightarrowtail \{[S_3, G_P^Q]\}}$$

  Note that S_1 may produce more than one result, and some results could be successes and others failures. The first rule above states that if there is a success, then S_2 is applied, otherwise S_3 is applied. To be able to decide which transition to use, the strategy S_1 should terminate. We will present in Section 5.5 a class Cond of strategies that are terminating.
- **Priority choice.** $(S_1)\texttt{orelse}(S_2)$ applies S_1 if possible, otherwise applies S_2. It fails if both S_1 and S_2 fail. To be able to decide which transition rule to use, the strategy S_1 should terminate.

$$\frac{\{[S_1, G_P^Q]\} \longmapsto^* \{[\mathsf{id}, G_1'], \ldots, [\mathsf{id}, G_k'], M\}}{[(S_1)\texttt{orelse}(S_2), G_P^Q] \rightarrowtail \{[\mathsf{id}, G_1'], \ldots, [\mathsf{id}, G_k']\}}$$

$$\frac{\{[S_1, G_P^Q]\} \longmapsto^* \{[\mathsf{fail}, G_1'], \ldots, [\mathsf{fail}, G_n']\}}{[(S_1)\texttt{orelse}(S_2), G_P^Q] \rightarrowtail \{[S_2, G_P^Q]\}}$$

where M is either empty or consists of pairs of the form $[\mathsf{fail}, G]$ only.
We chose to define $(S_1)\mathtt{orelse}(S_2)$ as a primitive operator instead of encoding it as $\mathtt{if}(S_1)\mathtt{then}(S_1)\mathtt{else}(S_2)$ since the language has probabilistic operators: evaluating S_1 in the condition and afterwards in the "then" branch of the if-then-else could yield different results. The semantics given above ensures that if S_1 can succeed then it can be successfully applied.

– **While.** Iteration is defined using a conditional as follows:

$$\frac{}{[\mathtt{while}(S_1)\mathtt{do}(S_2), G_P^Q] \rightarrowtail \{[\mathtt{if}(S_1)\mathtt{then}(S_2;\mathtt{while}(S_1)\mathtt{do}(S_2)), G_P^Q]\}}$$

If a maximum number of iterations is specified, then the semantics is defined by:

$$\frac{}{[\mathtt{while}(S_1)\mathtt{do}(S_2)(0), G_P^Q] \rightarrowtail \{[\mathsf{id}, G_P^Q]\}}$$

$$\frac{}{[\mathtt{while}(S_1)\mathtt{do}(S_2)(n+1), G_P^Q] \rightarrowtail \{[\mathtt{if}(S_1)\mathtt{then}(S_2;\mathtt{while}(S_1)\mathtt{do}(S_2)(n)), G_P^Q]\}}$$

– **Repeat.** The construction $\mathtt{repeat}(S)$ iterates the strategy S while it succeeds. It always returns id. Here S may be successful on different branches in the derivation tree. To be able to decide which semantic rule to use, the strategy S should terminate.

$$\frac{\{[S, G_P^Q]\} \longmapsto^* \{[\mathsf{id}, G'_1], \ldots, [\mathsf{id}, G'_k], M\}}{[\mathtt{repeat}(S), G_P^Q] \rightarrowtail \{[\mathtt{repeat}(S), G'_1], \ldots, [\mathtt{repeat}(S), G'_k]\}}\ (\star)$$

$(\star)$where M is either empty or consists of pairs of the form $[\mathsf{fail}, G]$ only.

$$\frac{\{[S, G_P^Q]\} \longmapsto^* \{[\mathsf{fail}, G'_1], \ldots, [\mathsf{fail}, G'_k]\}}{[\mathtt{repeat}(S), G_P^Q] \rightarrowtail \{[\mathsf{id}, G_P^Q]\}}$$

If a maximum number of repetitions is specified, then the semantics is defined by:

$$\frac{}{[\mathtt{repeat}(S)(0), G_P^Q] \rightarrowtail \{[\mathsf{id}, G_P^Q]\}}$$

$$\frac{\{[S, G_P^Q]\} \longmapsto^* \{[\mathsf{id}, G'_1], \ldots, [\mathsf{id}, G'_k], M\}}{[\mathtt{repeat}(S)(n+1), G_P^Q] \rightarrowtail \{[\mathtt{repeat}(S)(n), G'_1], \ldots, [\mathtt{repeat}(S)(n), G'_k]\}}\ (\star)$$

$(\star)$ where M is either empty or consists of pairs of the form $[\mathsf{fail}, G]$ only.

$$\frac{\{[S, G_P^Q]\} \longmapsto^* \{[\mathsf{fail}, G'_1], \ldots, [\mathsf{fail}, G'_k]\}}{[\mathtt{repeat}(S)(n+1), G_P^Q] \rightarrowtail \{[\mathsf{id}, G_P^Q]\}}$$

5.2 Probabilistic extension

To define the semantics of the probabilistic constructs in the language we now generalise the transition relation.

We denote by $\rightarrowtail_\pi$ a transition step with probability π. The relation $\rightarrowtail$ defined in the previous section can be seen as a particular case where $\pi = 1$, that is, $\rightarrowtail$ corresponds to $\rightarrowtail_1$.

The relation $\longmapsto$ also becomes probabilistic:

$$\{O_1, \dots, O_{k-1}, O_k, O_{k+1}, \dots, O_n\} \longmapsto_\pi \{O_1, \dots, O_{k-1}, O'_{k_1}, \dots, O'_{k_m}, O_{k+1}, \dots, O_n\}$$

if $O_k \rightarrowtail_\pi \{O'_{k_1}, \dots, O'_{k_m}\}$.

We are now ready to define the semantics of the remaining constructs in the strategy language.

Equiprobabilistic Choice of Reduct. The non-deterministic $\texttt{one}(T)$ operator takes as argument a rule or several rules in parallel (in the latter case, we create a new rule, as explained above). It selects only one amongst the set of legal reducts $LS_{L_W \Rightarrow R^N_M}(G^Q_P)$. Since all of them have the same probability of being selected, in the axiom below $\pi = 1/|LS_{L_W \Rightarrow R^N_M}(G^Q_P)|$.

$$\frac{}{[\texttt{one}(L_W \Rightarrow R^N_M), G^Q_P] \rightarrowtail_\pi \{[\mathsf{id}, G'^{Q'}_{P'}]\}} \quad \textit{if } G'^{Q'}_{P'} \in LS_{L_W \Rightarrow R^N_M}(G^Q_P)$$

$$\frac{}{[\texttt{one}(L_W \Rightarrow R^N_M), G^Q_P] \rightarrowtail_1 \{[\mathsf{fail}, G^Q_P]\}} \quad \textit{if } LS_{L_W \Rightarrow R^N_M}(G^Q_P) = \emptyset$$

Similarly, in the case of parallelism, the $\texttt{one}$ operator selects one amongst the set of parallel legal reducts, equiprobabilistically.

$$\frac{}{[\texttt{one}(L_{1W_1} \Rightarrow R_1{}^{N_1}_{M_1} || \dots || L_{kW_k} \Rightarrow R_k{}^{N_k}_{M_k}), G^Q_P] \rightarrowtail_\pi \{[\mathsf{id}, G'^{Q'}_{P'}]\}} \, (\star)$$

$(\star)$ if $G'^{Q'}_{P'} \in LPS_{L_{1W_1} \Rightarrow R_1{}^{N_1}_{M_1} || \dots || L_{kW_k} \Rightarrow R_k{}^{N_k}_{M_k}}(G^Q_P)$

$(\star)$ where $\pi = 1/|LPS_{L_{1W_1} \Rightarrow R_1{}^{N_1}_{M_1} || \dots || L_{kW_k} \Rightarrow R_k{}^{N_k}_{M_k}}(G^Q_P)|$

$$\frac{}{[\texttt{one}(L_{1W_1} \Rightarrow R_1{}^{N_1}_{M_1} || \dots || L_{kW_k} \Rightarrow R_k{}^{N_k}_{M_k}), G^Q_P] \rightarrowtail_1 \{[\mathsf{fail}, G^Q_P]\}} \, (\star)$$

$(\star)$ if $LPS_{L_{1W_1} \Rightarrow R_1{}^{N_1}_{M_1} || \dots || L_{kW_k} \Rightarrow R_k{}^{N_k}_{M_k}}(G^Q_P) = \emptyset$

Equiprobabilistic focusing. The commands $\texttt{setPos}(D)$ and $\texttt{setBan}(D)$ are probabilistic if the expression D is probabilistic. The operator $\texttt{one}(F)$ introduces non-determinism. The axioms are similar to the ones we gave in the previous section, but now the transitions are indexed by a probability:

$$\frac{}{[\mathtt{setPos}(\mathtt{one}(F)), G_P^Q] \rightarrowtail_\pi \{[\mathsf{id}, G_{P'}^Q]\}} \quad \textit{if } \mathtt{one}(F(G_P^Q)) =_\pi P'$$

$$\frac{}{[\mathtt{setBan}(\mathtt{one}(F)), G_P^Q] \rightarrowtail_\pi \{[\mathsf{id}, G_P^{Q'}]\}} \quad \textit{if } \mathtt{one}(F(G_P^Q)) =_\pi Q'$$

where $\mathtt{one}(F(G_P^Q)) =_{1/|V_{G'}|} F_1$ if $F(G_P^Q) = G'$ and $F_1 \in V_{G'}$. In other words, if $F(G_P^Q) = G'$, $\mathtt{one}(F(G_P^Q))$ returns one node in G' randomly chosen with an equiprobability $1/|V_{G'}|$ where $|V_{G'}|$ is the number of nodes in $V_{G'}$.

Probabilistic Choice of Rules, Positions and Strategies. Let Π be a probability distribution given by a list $\pi_1, \ldots, \pi_n \in [0,1]$ respectively associated to rules $T_1, \ldots, T_n$ such that $\pi_1 + \ldots + \pi_n = 1$, or, more generally, specified by an external function.

The probabilistic construct $\mathtt{ppick}(T_1, \ldots, T_n, \Pi)$ must be used with constructs $\mathtt{one}()$ or $\mathtt{all}()$, with the following semantics:

$$\frac{}{[\mathtt{one}(\mathtt{ppick}(T_1, \ldots, T_n, \Pi)), G_P^Q] \rightarrowtail_{\pi_j} \{[\mathtt{one}(T_j), G_P^Q]\}}$$

$$\frac{}{[\mathtt{all}(\mathtt{ppick}(T_1, \ldots, T_n, \Pi)), G_P^Q] \rightarrowtail_{\pi_j} \{[\mathtt{all}(T_j), G_P^Q]\}}$$

Like the probabilistic choice of rules, the probabilistic choice of positions

$$\mathtt{ppick}(F_1, \ldots, F_n, \Pi)$$

must be used with constructs $\mathtt{one}()$ or $\mathtt{all}()$ with the following rules:

$$\frac{}{[\mathtt{one}(\mathtt{ppick}(F_1, \ldots, F_n, \Pi)), G_P^Q] \rightarrowtail_{\pi_j} \{[\mathtt{one}(F_j), G_P^Q]\}}$$

$$\frac{}{[\mathtt{all}(\mathtt{ppick}(F_1, \ldots, F_n, \Pi)), G_P^Q] \rightarrowtail_{\pi_j} \{[\mathtt{all}(F_j), G_P^Q]\}}$$

Finally, probabilistic choice of strategies is defined by:

$$\frac{}{[\mathtt{ppick}(S_1, \ldots, S_n, \Pi), G_P^Q] \rightarrowtail_{\pi_j} \{[S_j, G_P^Q]\}}$$

5.3 Correctness and Completeness of the language

Given a strategic graph program $\left[S_{\mathcal{R}}, G_P^Q\right]$, we define sequences of $\longmapsto$ transitions according to the strategy $S_{\mathcal{R}}$, starting from the *initial configuration* $\left\{\left[S_{\mathcal{R}}, G_P^Q\right]\right\}$.

For a given configuration $C = \{O_1, \ldots, O_k, \ldots, O_n\}$, where each O_i is a strategic graph program $\left[S_i, {G_i}_{P_i}^{Q_i}\right]$, let $Reach(C) = \{G_1, \ldots, G_k, \ldots, G_n\}$ be the multiset of associated reachable graphs (that is, the projection of $O_1, \ldots, O_k, \ldots, O_n$ on the second component, forgetting about the position and banned subgraphs). For a sequence of configurations $T = C_0 \longmapsto \ldots \longmapsto C_n$, let $Reach(T) = \bigcup_{C_k, 0 \leq k \leq n} Reach(C_k)$ be the set of associated reachable graphs.

As presented in [53], it is expected from a strategy language to satisfy the properties of correctness and completeness with respect to rewriting derivations (in our case, port graph rewriting, see Definition 7).

In order to state these properties, we need to restrict the PORGY strategy language by excluding external functions (command `update`$(f\{list\})$) since those can change the attributes and their values in a graph in an arbitrary way. Such changes can have an impact on correctness and completeness of rewriting. Let us call this restricted strategy language PORGY-Light, abbreviated $Light(\mathcal{R})$.

Property 2 (Correctness). PORGY-Light is correct w.r.t. port graph rewriting. More precisely, for all G and $S \in Light(\mathcal{R})$, if T is the sequence of configurations $C_0 = \{[S, G]\} \longmapsto \ldots \longmapsto C_k = \{\ldots [S_k, G_k] \ldots\}$ and if $G' \in Reach(T)$, then $G \rightarrow^*_{\mathcal{R}} G'$.

Proof. If $G' \in Reach(T)$, G' is introduced at some step n of the derivation: $C_0 = \{[S, G]\} \longmapsto C_1 \ldots C_{n-1} \longmapsto C_n = \{\ldots [S', G'] \ldots\}$. Let us prove the result by induction on n and on the size $|S|$ of the strategy expression. If $n = 0$, this is trivial since $G' = G$. Assume the property holds for the derivation up to step $n-1$ and consider the step $C_{n-1} \longmapsto C_n$ with $[S_{n-1}, G_{n-1}] \in C_{n-1}$, $[S', G'] \in C_n$ such that $[S_{n-1}, G_{n-1}] \rightarrowtail \{\ldots, [S', G'], \ldots\}$. G_{n-1} has been introduced at some step $k < n$ and by induction $G \rightarrow^*_{\mathcal{R}} G_{n-1}$. Let us prove that $G_{n-1} \rightarrow^*_{\mathcal{R}} G'$ by case analysis on the different strategy constructs, applied in the $\rightarrowtail$ transition. For `id` or `fail`, there is no further step, so $G' = G_{n-1}$ and we are done. For Rules constructs, G' is obtained from G_{n-1} by rewriting and then $G \rightarrow^*_{\mathcal{R}} G_{n-1}$; or $G' = G_{n-1}$ in case of failure of rewriting. For positioning constructs, $G' = G_{n-1}$ since only the position/banned subgraphs may change. For sequential application $S_1; S_2$, G' is one of the graphs that occurs in the premise of the transition rule, and is obtained from G_{n-1} with S_1 such that $|S_1| < |S_1; S_2|$. By induction on the size of the strategy expression, $G_{n-1} \rightarrow^*_{\mathcal{R}} G'$. For all other compound strategy constructs, either $G' = G_{n-1}$ or G' is one of the graphs that occurs in the premise of the transition rule. In both cases the property holds with the same argument as for sequential application.

Property 3 (Completeness). PORGY-Light is complete w.r.t. (located) port graph rewriting. More precisely, if $G \rightarrow^*_{\mathcal{R}} G'$ ($G_P^Q \rightarrow^*_{\mathcal{R}} {G'}_{P'}^{Q'}$), there exists $S \in Light(\mathcal{R})$ and a sequence of configurations T of the form $C_0 = \{[S, G]\} \longmapsto \ldots \longmapsto C_k = \{\ldots [S'_k, G'_k] \ldots\}$ such that $G' \in Reach(T)$.

Proof. By induction on the derivation. For each rewriting step, one can find one or several $\longmapsto$-steps mimicking the choice of position and rule application, using positioning and rewriting constructs. The strategy S is then the sequence of strategies for every step.

5.4 Result sets

A configuration is *terminal* if no $\rightarrowtail$ transition can be performed. We prove in this section that all terminal configurations consist of results defined below in Definition 16. In other words, there are no blocked programs: the transition system ensures that, for any configuration, either there are transitions to perform, or we have reached results.

Definition 16 (Result). *Strategic graph programs are called* results *if they are of the form* $[\mathsf{id}, G_P^Q]$ *or* $[\mathsf{fail}, G_P^Q]$.

Terminal configurations contain only results, thanks to the following property.

Property 4 (Progress: Characterisation of Terminal Configurations). For every strategic graph program $[S, G_P^Q]$ that is not a result (i.e., $S \neq \mathsf{id}$ and $S \neq \mathsf{fail}$), there exists a configuration C such that $[S, G_P^Q] \rightarrowtail C$.

Proof. By induction on S. According to Definition 14 (see the axioms and rules given in Sections 5.1 and 5.2), for every strategic graph program $[S, G_P^Q]$ different from $[\mathsf{id}, G_P^Q]$ or $[\mathsf{fail}, G_P^Q]$ there is an axiom or rule that applies (it suffices to check all cases in the grammar for S).

Given a strategic graph program $\left[S, G_P^Q\right]$, we can associate a set of results to a finite sequence T out of the initial configuration $\{\left[S, G_P^Q\right]\}$: the *result set* associated to T is the set of results in the last configuration.

Definition 17 (Result set). *For a given configuration* $C = \{O_1, \ldots, O_i, \ldots, O_n\}$, *where each* O_i *is a strategic program* $\left[S_i, G_{i\,P_i}^{\,Q_i}\right]$, *let* $Results(C)$ *be the subset of* C *that are results.*
If T *is the derivation* $C_0 = \{[S_0, G_0]\} \longmapsto \ldots \longmapsto C_k = \{\ldots\left[S_k, G_{k\,P_k}^{\,Q_k}\right]\ldots\}$, $Results(T) = Results(C_k)$.

The result set associated to a configuration or a derivation can be empty, if there are no results in the configurations of the sequence, which can be the case for non-terminating programs.

If the sequence of $\longmapsto$ transitions out of the initial configuration $\{\left[S, G_P^Q\right]\}$ ends in a terminal configuration, then the result set is a *program result.* If a strategic graph program does not reach a terminal configuration (in case of non-termination) then the program result is undefined ($\perp$).

The full strategy language contains probabilistic operators `one`() and `ppick`(). They may produce different results if they are applied twice on the same strategic graph program and in this sense are non-deterministic. Indeed in that case, other operators (e.g., `orelse`, `if-then-else`, `repeat`) inherit this non-deterministic behaviour.

For the core language defined in Section 5.1, we have the property:

Property 5 (Determinism). Each strategic graph program in the core language (i.e., excluding `one` and `ppick`), always returns the same program result in every execution.

Proof. If we restrict the strategy language to the core constructs, the $\longmapsto$ transition relation is deterministic, and gives all possible cases.

In the full language, if $\left[S, G_P^Q\right]$ is a strategic graph program without `ppick`, its execution gives a (non-strict) subset of results that would be obtained by replacing in the strategy S, the constructs `one`, by `all`.

5.5 Termination

Definition 18. *A strategic graph program* $\left[S, G_P^Q\right]$ *is* strongly terminating *(or just terminating) if there is no infinite* $\longmapsto$ *transition sequence out of the initial configuration* $\{\left[S, G_P^Q\right]\}$*, otherwise it is non-terminating. It is* weakly terminating *if there exists a configuration having at least one result.*

Graph programs are not terminating in general, however we can identify a terminating sublanguage (i.e. a sublanguage for which the transition relation is terminating) and we can characterise the strategic graph programs that yield terminal configurations. The proof of termination needs a preliminary lemma.

Lemma 1. *If* $\left[S_1, G_P^Q\right]$ *is strongly terminating and* S_2 *is such that* $\left[S_2, G'^{Q'}_{P'}\right]$ *is strongly terminating for any* $G'^{Q'}_{P'}$*, then* $\left[S_1; S_2, G_P^Q\right]$ *is strongly terminating.*

Proof. By induction on the maximal length of reductions out of $\left[S_1, G_P^Q\right]$. Let us consider the transition rules for sequence. The base cases in the induction correspond to $S_1 = \mathsf{id}$ or $S_1 = \mathsf{fail}$. Then the property holds by assumption, since S_2 is strongly terminating for any graph.

Assume the property holds when the maximal length of reductions out of $\left[S_1, G_P^Q\right]$ is n (Induction Hypothesis). Since $\left[S_1, G_P^Q\right] \rightarrowtail \{\left[S_1^1, G1_{P_1}^{Q_1}\right], \ldots, \left[S_1^k, G_k{}_{P_k}^{Q_k}\right]\}$ (rule premise), we can apply the induction hypothesis to deduce that each of

$$\left[S_1^1; S_2, G1_{P_1}^{Q_1}\right], \ldots, \left[S_1^k; S_2, G_k{}_{P_k}^{Q_k}\right]$$

is strongly terminating.

Property 6 (Termination). The sublanguage that excludes the while and repeat constructs is strongly terminating.

Proof. We prove, by induction on the structure of S, that for all S and for all G_P^Q, $\left[S, G_P^Q\right]$ is strongly terminating if S does not contain while and repeat. Let $[S,G] \rightarrowtail \{\ldots[S_k,G_k]\ldots\}$ be a transition step, it is easy to check that either S_k is a sub-expression of S, then the result follows by induction hypothesis, or $S = S_1; S_2$, in which case we conclude using Lemma 1.

Corollary 1. *Let Cond be the language obtained from the core language by excluding the* while *and* repeat *strategies. Cond is terminating and deterministic.*

5.6 Expressivity

With respect to the computation power of the language, it is easy to state, as in [37], the Turing completeness property since the language includes sequence and iteration.

Property 7 (Computational Completeness). The set of all strategic graph programs $\left[S_{\mathcal{R}}, G_P^Q\right]$ is Turing complete, i.e. can simulate any Turing machine.

6 Implementation

Porgy is implemented on top of the visualisation framework Tulip [6] as a set of C++11 Tulip plugins. 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.

Tulip is an information visualisation framework dedicated to the analysis and visualisation of relational data. It provides a complete Python and C++ API supporting the design of interactive information visualisation applications easily customisable to address a wide range of visualisation problems. The plugin architecture of Tulip enables to change a component of Porgy (i.e., replace the matching algorithm) without modifying other parts of the software and permits to add new features, such as importing or exporting data from/to other software programs.

Graph Data Structure. 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. This model is close to our notion of records (see Def. 1). Other Tulip features we need are described below. We refer the reader to [58] for more details about the interactive features of Porgy and how they are implemented.

We made an abstraction layer on top of the Tulip graph data structure to handle port graphs. Thus, a port graph node is composed of a set of Tulip nodes (the ports), each connected with an edge to a centre node. Taking advantage of Tulip graph data structure with properties, we implement the attributed port graph functions *Attach, Connect* and *Ports* with Tulip properties. For a rule, the distinction between the left-hand side and the right-hand side is also implemented with Tulip properties. Other attributes related to the graph structure, like *Self*, which stores the physical identity of any graph element, or *Arity*, which stores the degree of a port, are also always present in records, since they are heavily used as seen in the description of the matching process. *Data attributes* are used to store information relevant to the model being constructed.

Rewriting Core Engine. When applying a rule $L_W \Rightarrow_C R_N^M$ on a located graph G_P^Q, the first operation is to find a matching morphism from L to G, then given a morphism, the image of L is replaced in G by R as indicated in Definition 7. As a consequence, we have implemented rule applications via two Tulip plugins, one for each step.

The morphism problem, known as graph-subgraph isomorphism, still receives great attention from the community. We use Ullman's original algorithm [72] because its implementation is straightforward, it is used as a reference in many papers and many existing algorithms are only small variations of Ullman's work for specific graph classes. It takes as input a left-hand side of a rule and a graph; its output is a list of morphisms from L to the graph. In the systems we have considered so far, the left-hand side of rules are always small compared to the size of the graph they apply on. This reduces heavily the complexity of the graph-subgraph isomorphism. Moreover, when looking for images of nodes of the left-hand side, we first check nodes of the graph that are not banned before checking adjacency. We follow a classical strategy which is to reduce the search tree as much as possible and as early as possible.

The second plugin is an implementation of Def. 7. To avoid a high memory consumption, we use the graph hierarchy features of Tulip. Every produced graph (G, G', . . .) has a direct common ancestor, *i.e.*, they are subgraphs of the same graph. In Tulip, a subgraph can be represented using very little memory because it is only a filter on its direct ancestor in the hierarchy: only nodes and edges modified by a rewriting step are created. The rest are left untouched. They are just marked as present in the newly created graph.

Strategies. The operational semantics defined in Section 5 is also a Tulip plugin. It works like an interpreter (see [29] for more details). From a string describing a strategy to run, the plugin calls the rule application plugins, updates the derivation tree and makes the necessary computation for the banned and position subgraphs. If an error is detected (wrong syntax, non-existing rules/attributes, ...) inside the strategy, it is not computed. An error message is shown to the user.

This plugin is developed with the Spirit Library from Boost (see `http://www.boost.org/libs/spirit`). Spirit enables to directly implement the small-step operational semantics given above. Below, we give some implementation details for some operators.

There are two ways to implement the construct `one`(T): either by taking the first computed redex, or by randomly choosing one among several. In both cases, it acts as a cut mechanism in logic programming. We chose to take the first computed redex as our implementation of Ullman's algorithm does a random iteration on the nodes of the graph when looking for a morphism.

As we have divided rule application into two steps, the implementation of `match`(T) is straightforward. We just apply the subgraph isomorphism plugin. If it finds at least one morphism, `match`(T) returns `id`, otherwise it returns `fail`.

The regular expression capabilities (when using $=\sim$ with `Property` or `Ngb`) directly use the regular expression capabilities of C++11.

Visualisation and interaction features. In order to support the various tasks involved in the study of a port graph rewriting system, Porgy provides functionalities such as:

– Different synchronised views on each component of the rewriting system. For instance, the selection of some nodes in a port graph triggers the selection of the corresponding nodes in the whole hierarchy. They are visible inside each node of the derivation tree.
– Drag-and-drop mechanisms to apply rules and strategies on any node of the derivation tree. While in this paper a derivation tree is defined for one strategy, the interactive and visualisation features of Porgy allow us to easily apply any strategy on any node of the derivation tree.
– Navigation in the tree, for instance, backtracking and exploring different branches. We can track reductions throughout the whole derivation tree.
– Plot of evolution of a chosen parameter (a specific element in the port graph structure) along a derivation. Such a mechanism helps to analyse or debug a rule system, by tracking properties of the output graph along the rewriting process.
– Identification of isomorphic nodes in the derivation tree, grouping them to show cycles or possible shortcuts in the derivation tree.

The interested reader can refer to [58] for more details.

7 Related Work

Graph grammars were introduced in [56] to represent pictures and geometrical problems. Bunke [17] proposes the use of *attributed graphs* and *graph transformations* to interpret diagrams and flowcharts. This work gave rise to numerous applications in a variety of domains: recognition of music notation, implementation of programming languages, visual programming, modelling of biological processes, software development environments. In all these works, graph transformations are usually specified by means of rules [19,25].

To formally define the transformation (i.e., rewriting) relation generated by the rules, it is necessary to give a formal semantics for rules and for their application. The most well-known approaches are algebraic (that is, based on an algebraic construction, as in the double pushout [26] or single pushout [41,49,66] semantics) or algorithmic (that is, the application of rules is described as a sequence of atomic operations, as we have done in this paper). Although algorithmic, our rewriting semantics follows the single pushout approach, as shown in Section 2.4.

Nowadays, graph rewriting is implemented in a variety of tools. In AGG [27], application of rules can be controlled by defining *layers* and then iterating through and across layers. PROGRES [68] allows users to define the way rules are applied and includes non-deterministic constructs, sequence, conditional and loops. The Fujaba Tool Suite [55] offers a basic strategy language, including conditionals, sequence and method calls, but no concurrency. GROOVE [67] permits to control the application of rules, via a control language with sequence, loop, random choice, try()else() and simple function calls. In GReAT [7] the pattern-matching algorithm always starts from specific nodes called "pivot nodes"; rule execution is sequential and there are conditional and looping structures. GrGen.NET [33] uses the concept of search plans to represent different matching strategies. GP [63] is a 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, and uses a Prolog-like backtracking technique to explore the derivation tree, unlike PORGY where the derivation tree is displayed and 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, the PORGY 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 [28], RuleBender [69], Mosbie [76] 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 ours and BioNetGen uses port graphs.

The strategy language defined in this paper is strongly inspired by the work on GP and PROGRES, and by strategy languages developed for term rewriting, such as ELAN [12], Tom [8] and Stratego [74]. It can be applied to terms as a particular case; then the constructs dealing with applications and strategies are similar to those found in ELAN or Stratego. The Positions constructs sublanguage on the other hand can be seen as a lower level version of these languages. Instead of providing built-in (predefined) traversal mechanisms, PORGY's language allows users to program traversals by using Positions constructs (see, for example, the definition of outermost and innermost rewriting in Section 4.3).

The probabilistic primitives in the strategy language allow users to model basic dynamic behaviour in non-deterministic and probabilistic systems. Probabilistic or stochastic features are widely used to deal with uncertainties and huge volumes of data, and there has been extensive research on models, logics and verification of probabilistic systems, including probabilistic programming languages, probabilistic Petri nets, probabilistic algebra approaches, Continuous

Stochastic logic (CSL), Probabilistic Computational Logic,... For some of these logics, tools have been developed to support model checking specifications (e.g., PRISM [46]).

Several approaches to combine rewriting with probabilities have been explored in previous work. For example:

- Probabilistic (real-time) rewrite theories have been defined and implemented in

  PMaude [1] and extended in [11]. Based on probabilistic rewrite theories, [45] provides a general semantic framework that unifies several existing models of probabilistic systems mentioned above.
- To perform stochastic simulation in biological signalling pathways [21], the κ language [22] and the BioNetGen system [28] provide for each state of the system and each rule, a rate law used to determine the probability that a reaction occurs within a given fixed time step. How to compute this probability is detailed for instance in [18].
- Probabilistic functional programming languages such as Church [34,35] and IBAL [57] extend well-known deterministic languages (LISP and ML, respectively) with primitive constructs for random choice. The abstract semantics of these probabilistic languages can be defined as a map from programs to distributions over executions. For example, Church allows programmers to include as a parameter the probability distribution to be used, and in addition it provides a language construct called *mem*, which memorises its input function and is useful for describing persistent random properties.
- Probabilistic graph transformation systems [44] combine probabilistic and nondeterministic behaviour following the double-pushout approach. If there are several rules with the same left-hand side, a probability distribution can be specified to choose a right-hand side when a matching morphism is found. If several matching morphisms exist, one of them is chosen nondeterministically.

In contrast to these approaches, we define a probabilistic choice operator which works on rules, positions or strategies. Our approach is close to the one followed in [14,15] that studies the definition and consequences of a probabilistic choice operator for strategies in the context of the ρ-calculus and of the rewriting logic. A similar behaviour as [44] can be obtained in Porgy using the `one` construct (non-deterministic redex selection) and the `ppick` construct (probabilistic rule selection), however, in Porgy the latter is not restricted to rules with the same left-hand side. But understanding the possible semantic relations between the different approaches mentioned above and their respective implications on the properties of the modelled systems needs more work.

The Porgy system has been tailored to various application domains, mainly biological systems, interaction nets, graph theory, social networks. Its strategy language has evolved to reflect this progression. Previous versions of Porgy's strategy language were presented in [3,29,30,54]. The main changes with respect to the previous versions are in the specification of graph morphisms (here we take into account the attributes of nodes, ports and edges and the conditions specified in rewrite rules), in the Position constructs to deal with rewriting positions (the Property and Ngb constructs defined in this paper are new), and in the `ppick` constructs. The small-step style of operational semantics is also new: the core language was defined using semantic rules in [30, 54] and with an abstract machine in [29]. Here we have chosen to give a small-step operational semantics because it provides an intermediate level of abstraction between the semantic rules and the abstract machine. Similar approaches based on reduction rules and abstract machines have been used to specify the semantics of interaction nets [32, 48, 59], but note that the strategy is built into the semantics in these works, whereas in PORGY it is part of the program. In this sense, the style of operational semantics used in this paper is closer to the one used to provide an operational semantics for GP in [65].

8 Conclusion

This paper presents strategic port graphs programs and their implementation in the PORGY system, an environment for visual modelling of complex systems through port graphs and port graph rewrite rules. PORGY provides tools to build port graphs from scratch, with nodes, ports, edges and associated attributes. It offers also means to visualise traces of rewriting as a derivation tree. The strategy language is used in particular to guide the construction of this derivation tree. PORGY also emphasises visualisation and scale, thanks to the TULIP back-end which can handle large graphs with millions of elements and comes with powerful visualisation and interaction features.

Many improvements are yet possible both at theoretical and practical levels.

At the strategy level, it would be interesting to define strategies with memory, where the next rewriting steps are decided depending on the history of the derivation, i.e. on previous states. It should be noted that, since PORGY gives access to the derivation tree of a strategic graph program, a mechanism to get the history of a state should be easy to implement. From a conceptual point of view, we need to introduce a more general notion of strategic graph program to include the history. Having the derivation tree as a first-class component of the system allows us not only to define strategies with memory, but also to easily perform analyses such as detecting cycles (repetition of nodes in the derivation tree within a path), and identifying the rewrite rules responsible for such cycles (which helps users in the specification and debugging phases).

Efficiency of port graph rewriting remains an important issue when huge graphs with complex records are considered. A first idea is to include nodes in the rule morphism (via the arrow node) to perform less copying, another one is to explore the approximate matching approach to find redexes.

As in any programming context, debugging and verification are crucial. Many questions have been addressed in the rewriting community that could be worth adapting to our port graph setting: termination analysis, confluence analysis, conflicting rules detection, cycle detection, fairness analysis. With program debugging and verification in mind, we can also mention reachability proof, detection of unwanted patterns, error detection and correction. Indeed these questions are related to the design of an ambitious debugging, verifying and certifying environment for strategic graph programs, which is a long term goal.

Last but not least, modelling and analysing dynamic systems on massive data formalised through graph data bases provide interesting opportunities and challenges for a system like PORGY, that are worth exploring. In this context, due to the frequent imprecision of data, stochastic and probabilistic reasoning is often necessary. Clarifying the relation between our semantics and probabilistic transition systems is thus a question to address in the future. This is indeed necessary in order to develop adequate verification and debugging tools.

Acknowledgements. 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.

References

1. Gul A. Agha, José Meseguer, and Koushik Sen. PMaude: Rewrite-based specification language for probabilistic object systems. *Electronic Notes in Theoretical Computer Science*, 153(2):213–239, 2006.
2. Oana Andrei. *A Rewriting Calculus for Graphs: Applications to Biology and Autonomous Systems.* PhD thesis, Institut National Polytechnique de Lorraine, 2008.
3. Oana Andrei, Maribel Fernández, Hélène Kirchner, Guy Melançon, Olivier Namet, and Bruno Pinaud. PORGY: Strategy-Driven Interactive Transformation of Graphs. In Rachid Echahed, editor, 6^{th} *Int. Workshop on Computing with Terms and Graphs*, volume 48 of *Electronic Proceedings in Theoretical Computer Science*, pages 54–68, 2011.
4. Oana Andrei and Hélène Kirchner. A Rewriting Calculus for Multigraphs with Ports. In *Proceedings of RULE'07*, volume 219 of *Electronic Notes in Theoretical Computer Science*, pages 67–82, 2008.
5. Oana Andrei and Hélène Kirchner. A Higher-Order Graph Calculus for Autonomic Computing. In *Graph Theory, Computational Intelligence and Thought. Golumbic Festschrift*, volume 5420 of *Lecture Notes in Computer Science*, pages 15–26. Springer, 2009.
6. David Auber, Romain Bourqui, Maylis Delest, Antoine Lambert, Patrick Mary, Guy Melançon, Bruno Pinaud, Benjamin Renoust, and Jason Vallet. TULIP 4. Research report, LaBRI - Laboratoire Bordelais de Recherche en Informatique, September 2016.
7. Daniel Balasubramanian, Anantha Narayanan, Christopher P. van Buskirk, and Gabor Karsai. The Graph Rewriting and Transformation Language: GReAT. *Electronic Communications of the European Association for the Study of Science and Technology*, 1, 2006.

8. Emilie Balland, Paul Brauner, Radu Kopetz, Pierre-Etienne Moreau, and Antoine Reilles. Tom: Piggybacking Rewriting on Java. In Franz Baader, editor, *Rewriting Techniques and Applications (RTA)*, volume 4533 of *Lecture Notes in Computer Science*, pages 36–47. Springer, 2007.
9. Henk Barendregt. *The Lambda Calculus, Its Syntax and Semantics.* North Holland, 1981.
10. H.P. Barendregt, M.C.J.D. van Eekelen, J.R.W. Glauert, J. R. Kennaway, M.J. Plasmeijer, and M.R. Sleep. Term graph rewriting. In *Proceedings of PARLE, Parallel Architectures and Languages Europe*, number 259-II in Lecture Notes in Computer Science, pages 141–158, Eindhoven, The Netherlands, 1987. Springer-Verlag.
11. Lucian Bentea and Peter Csaba Ölveczky. A probabilistic strategy language for probabilistic rewrite theories and its application to cloud computing. In *Recent Trends in Algebraic Development Techniques, 21st International Workshop, WADT 2012, Salamanca, Spain, June 7-10, 2012, Revised Selected Papers*, pages 77–94, 2012.
12. Peter Borovanský, Claude Kirchner, Hélène Kirchner, Pierre-Etienne Moreau, and Christophe Ringeissen. An overview of ELAN. *ENTCS*, 15, 1998.
13. Tony Bourdier, Horatiu Cirstea, Daniel J. Dougherty, and Hélène Kirchner. Extensional and intensional strategies. In *Proceedings Ninth International Workshop on Reduction Strategies in Rewriting and Programming*, volume 15 of *Electronic Proceedings in Theoretical Computer Science*, pages 1–19, 2009.
14. Olivier Bournez and Mathieu Hoyrup. Rewriting logic and probabilities. In Robert Nieuwenhuis, editor, *Rewriting Techniques and Applications, 14th International Conference, RTA 2003, Valencia, Spain, June 9-11, 2003, Proceedings*, volume 2706 of *Lecture Notes in Computer Science*, pages 61–75. Springer, 2003.
15. Olivier Bournez and Claude Kirchner. Probabilistic rewrite strategies. applications to ELAN. In Sophie Tison, editor, *Proceedings of the 13th Rewriting Techniques and Applications (RTA) conference*, volume 2378 of *Lecture Notes in Computer Science*, pages 252–266. Springer, July 2002.
16. Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/XT 0.17. A language and toolset for program transformation. *Science of Computer Programming*, 72:52–70, 2008. Special issue on Experimental Systems and Tools.
17. Horst Bunke. Attributed programmed graph grammars and their application to schematic diagram interpretation. *IEEE Trans. Pattern Anal. Mach. Intell.*, 4(6):574–582, 1982.
18. Joshua Colvin, Michael I Monine, James R Faeder, William S Hlavacek, Daniel D Von Hoff, and Richard G Posner. Simulation of large-scale rule-based models. *Bioinformatics*, 25(7):910–917, 04 2009.
19. Andrea Corradini, Ugo Montanari, Francesca Rossi, Hartmut Ehrig, Reiko Heckel, and Michael Löwe. Algebraic approaches to graph transformation - part I: Basic concepts and double pushout approach. In *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations, Chapter 3*, pages 163–246. World Scientific, 1997.
20. Bruno Courcelle. Graph Rewriting: An Algebraic and Logic Approach. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, pages 193–242. Elsevier and MIT Press, 1990.
21. Vincent Danos, Jérôme Feret, Walter Fontana, Russell Harmer, and Jean Krivine. Rule-based modelling of cellular signalling. In Luis Caires and Vasco Vasconcelos,

editors, *CONCUR 2007 - Concurrency Theory*, volume 4703 of *Lecture Notes in Computer Science*, pages 17–41. Springer Berlin Heidelberg, 2007.

22. Vincent Danos and Cosimo Laneve. Formal Molecular Biology. *Theoretical Computer Science*, 325(1):69–110, 2004.
23. Edsger W. Dijkstra. *Selected writings on computing - a personal perspective.* Texts and monographs in computer science. Springer, 1982.
24. 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 *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations, Chapter 4*, pages 247–312. World Scientific, 1997.
25. Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1-3.* World Scientific, 1997.
26. Hartmut Ehrig, Michael Pfender, and Hans Jürgen Schneider. Graph-grammars: An algebraic approach. In *14th Annual Symposium on Switching and Automata Theory, Iowa City, Iowa, USA, October 15-17, 1973*, pages 167–180, 1973.
27. Claudia Ermel, Michael Rudolf, and Gabriele Taentzer. The AGG approach: Language and environment. In Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 2: Applications, Languages, and Tools*, pages 551–603. World Scientific, 1997.
28. James Faeder, Michael Blinov, and William Hlavacek. Rule-based modeling of biochemical systems with bionetgen. In Ivan V. Maly, editor, *Systems Biology*, volume 500 of *Methods in Molecular Biology*, pages 113–167. Humana Press, 2009.
29. Maribel Fernández, Hélène Kirchner, Ian Mackie, and Bruno Pinaud. Visual Modelling of Complex Systems: Towards an Abstract Machine for PORGY. In Arnold Beckmann, Erzsébet Csuhaj-Varjú, and Klaus Meer, editors, *Computability In Europe*, volume 8493 of *Language, Life, Limits*, pages 183–193, Budapest, Hungary, June 2014. Springer International Publishing.
30. Maribel Fernández, Hélène Kirchner, and Olivier Namet. A strategy language for graph rewriting. In Germán Vidal, editor, *Logic-Based Program Synthesis and Transformation*, volume 7225 of *Lecture Notes in Computer Science*, pages 173–188. Springer Berlin Heidelberg, 2012.
31. Maribel Fernández, Hélène Kirchner, and Bruno Pinaud. Strategic port graph rewriting: An interactive modelling and analysis framework. In Dragan Bosnacki, Stefan Edelkamp, Alberto Lluch-Lafuente, and Anton Wijs, editors, *Proceedings 3rd Workshop on GRAPH Inspection and Traversal Engineering, GRAPHITE 2014, Grenoble, France, 5th April 2014.*, volume 159 of *Electronic Proceedings in Theoretical Computer Science*, pages 15–29, 2014.
32. Maribel Fernández and Ian Mackie. A calculus for interaction nets. In *Proceedings of PPDP'99, Paris*, number 1702 in Lecture Notes in Computer Science. Springer, 1999.
33. Rubino Geiß, Gernot Veit Batz, Daniel Grund, Sebastian Hack, and Adam Szalkowski. GrGen: A Fast SPO-Based Graph Rewriting Tool. In *Proc. of ICGT*, volume 4178 of *Lecture Notes in Computer Science*, pages 383–397. Springer, 2006.
34. Noah D. Goodman. The principles and practice of probabilistic programming. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, pages 399–402, 2013.

35. Noah D. Goodman, Vikash K. Mansinghka, Daniel M. Roy, Keith Bonawitz, and Joshua B. Tenenbaum. Church: a language for generative models. In *UAI 2008, Proceedings of the 24th Conference in Uncertainty in Artificial Intelligence, Helsinki, Finland, July 9-12, 2008*, pages 220–229, 2008.
36. Annegret Habel, Jürgen Müller, and Detlef Plump. Double-pushout graph transformation revisited. *Mathematical Structures in Computer Science*, 11(5):637–688, 2001.
37. Annegret Habel and Detlef Plump. Computational completeness of programming languages based on graph transformation. In *Foundations of Software Science and Computation Structures, 4th International Conference, FOSSACS 2001, Proceedings*, volume 2030 of *Lecture Notes in Computer Science*, pages 230–245. Springer, 2001.
38. Michael Hanus. Curry: A multi-paradigm declarative language (system description). In *Twelfth Workshop Logic Programming, WLP'97, Munich*, 1997.
39. Simon L. Peyton Jones. *Haskell 98 language and libraries: the revised report.* Cambridge University Press, 2003.
40. N. Kejẑar, Z. Nikoloski, and V. Batagelj. Probabilistic inductive classes of graphs. *The Journal of Mathematical Sociology*, 32(2):85–109, 2008.
41. Richard Kennaway. On "on graph rewritings". *Theor. Comput. Sci.*, 52(1–2):37–58, 1987.
42. Claude Kirchner, Florent Kirchner, and Hélène Kirchner. Strategic computations and deductions. In *Reasoning in Simple Type Theory. Studies in Logic and the Foundations of Mathematics, vol.17*, pages 339–364. College Publications, 2008.
43. Hélène Kirchner. 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. Springer, 2015.
44. Christian Krause and Holger Giese. Probabilistic graph transformation systems. In Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *Graph Transformations - 6th International Conference, ICGT 2012, Bremen, Germany, September 24-29, 2012. Proceedings*, volume 7562 of *Lecture Notes in Computer Science*, pages 311–325. Springer, 2012.
45. Nirman Kumar, Koushik Sen, Jos?e Meseguer, and Gul Agha. Probabilistic rewrite theories: Unifyoing models, logics and tools. Technical Report UIUCDCS-R-2003-2347, Dept of Computer Science, Univeristy of Illinois at Urbana Champaign, 2003.
46. M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In G. Gopalakrishnan and S. Qadeer, editors, *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)*, volume 6806 of *Lecture Notes in Computer Science*, pages 585–591. Springer, 2011.
47. Yves Lafont. Interaction nets. In *Proceedings of the 17th ACM Symposium on Principles of Programming Languages (POPL'90)*, pages 95–108. ACM Press, 1990.
48. Sylvain Lippi. in2: A graphical interpreter for interaction nets. In *Proceedings of the 13th International Conference on Rewriting Techniques and Applications (RTA '02)*, pages 380–386, London, UK, 2002. Springer-Verlag.
49. Michael Löwe. Algebraic approach to single-pushout graph transformation. *Theoretical Computer Science*, 109:181–224, 1993.
50. Michael Löwe, Martin Korff, and Annika 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*, pages 185–199. John Wiley and Sons Ltd., Chichester, UK, 1993.
51. Salvador Lucas. Strategies in programming languages today. *Electronic Notes in Theoretical Computer Science*, 124(2):113–118, 2005.

52. Narciso Martí-Oliet, José Meseguer, and Alberto Verdejo. Towards a strategy language for Maude. *Electronic Notes in Theoretical Computer Science*, 117:417–441, 2005.
53. Narciso Martí-Oliet, José Meseguer, and Alberto Verdejo. A rewriting semantics for Maude strategies. *Electronic Notes in Theoretical Computer Science*, 238(3):227–247, 2008.
54. Olivier Namet. *Strategic Modelling with Graph Rewriting Tools*. PhD thesis, King's College London, 2011.
55. Ulrich Nickel, Jörg Niere, and Albert Zündorf. The FUJABA environment. In *Proccedings of International Conference on Software Engineering-ICSE*, pages 742–745, 2000.
56. John L. Pfaltz and Azriel Rosenfeld. Web grammars. In Donald E. Walker and Lewis M. Norton, editors, *Proceedings of the 1st International Joint Conference on Artificial Intelligence, Washington, DC, May 1969*, pages 609–620. William Kaufmann, 1969.
57. Avi Pfeffer. IBAL: A probabilistic rational programming language. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence, IJCAI 2001, Seattle, Washington, USA, August 4-10, 2001*, pages 733–740, 2001.
58. Bruno Pinaud, Guy Melançon, and Jonathan Dubois. PORGY: A Visual Graph Rewriting Environment for Complex Systems. *Computer Graphics Forum*, 31(3):1265–1274, 2012.
59. Jorge Sousa Pinto. Sequential and concurrent abstract machines for interaction nets. In J. Tiuryn, editor, *Proceedings of Foundations of Software Science and Computation Structures (FOSSACS)*, volume 1784 of *Lecture Notes in Computer Science*, pages 267–282. Springer-Verlag, 2000.
60. M. J. Plasmeijer and M. C. J. D. van Eekelen. *Functional Programming and Parallel Graph Rewriting*. Addison-Wesley, 1993.
61. Gordon D. Plotkin. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming*, 60-61:17–139, 2004.
62. Detlef Plump. Term graph rewriting. In Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 2: Applications, Languages, and Tools*, pages 3–61. World Scientific, 1998.
63. Detlef Plump. The Graph Programming Language GP. In Symeon Bozapalidis and George Rahonis, editors, *Algebraic Informatics CAI*, volume 5725 of *Lecture Notes in Computer Science*, pages 99–122. Springer, 2009.
64. Detlef Plump. The design of GP 2. In Santiago Escobar, editor, *Proceedings 10th International Workshop on Reduction Strategies in Rewriting and Programming, WRS 2011, Novi Sad, Serbia, 29 May 2011.*, volume 82 of *Electronic Proceedings in Theoretical Computer Science*, pages 1–16, 2011.
65. Detlef Plump and Sandra Steinert. The semantics of graph programs. In *Proceedings Tenth International Workshop on Rule-Based Programming, RULE 2009, Brasília, Brazil, 28th June 2009.*, pages 27–38, 2009.
66. Jean-Claude Raoult. On graph rewritings. *Theor. Comput. Sci.*, 32:1–24, 1984.
67. Arend Rensink. The GROOVE Simulator: A Tool for State Space Generation. In *Applications of Graph Transformations with Industrial Relevance (AGTIVE)*, volume 3062 of *Lecture Notes in Computer Science*, pages 479–485. Springer, 2003.
68. Andy Schürr, Andreas J. Winter, and Albert Zündorf. The PROGRES Approach: Language and Environment. In Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *Handbook of Graph Grammars and Com-*

puting by Graph Transformations, Volume 2: Applications, Languages, and Tools, pages 479–546. World Scientific, 1997.

69. Adam M. Smith, Wen Xu, Yao Sun, James R. Faeder, and G.Elisabeta Marai. Rulebender: integrated modeling, simulation and visualization for rule-based intracellular biochemistry. *BMC Bioinformatics*, 13(8), 2012.
70. Terese. *Term Rewriting Systems*. Cambridge University Press, 2003. M. Bezem, J. W. Klop and R. de Vrijer, eds.
71. René Thiemann, Christian Sternagel, Jürgen Giesl, and Peter Schneider-Kamp. Loops under strategies ... continued. In *Proceedings International Workshop on Strategies in Rewriting, Proving, and Programming*, volume 44 of *Electronic Proceedings in Theoretical Computer Science*, pages 51–65, 2010.
72. J.R. Ullman. An algorithm for subgraph isomorphism. *Journal of the ACM*, 23(1):31–42, 1976.
73. Jason Vallet, Hélène Kirchner, Bruno Pinaud, and Guy Melançon. A visual analytics approach to compare propagation models in social networks. In Arend Rensink and Eduardo Zambon, editors, *Proceedings Graphs as Models, GaM 2015, London, UK, 11-12 April 2015.*, volume 181 of *Electronic Proceedings in Theoretical Computer Science*, pages 65–79, 2015.
74. Eelco Visser. Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In *Proceedings of the International Conference on Rewriting Techniques and Applications (RTA'01)*, volume 2051 of *Lecture Notes in Computer Science*, pages 357–361. Springer-Verlag, 2001.
75. Eelco Visser. A survey of strategies in rule-based program transformation systems. *Journal of Symbolic Computation*, 40(1):831–873, 2005.
76. John E.Jr. Wenskovitch, Leonard A. Harris, Jose-Juan Tapia, James R. Faeder, and G. Elisabeta Marai. Mosbie: a tool for comparison and analysis of rule-based biochemical models. *BMC Bioinformatics*, 15(1), 2014.