Strategic Port Graph Rewriting: An Interactive Modelling and Analysis 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
[2]Inria, 200 avenue de la Vieille Tour, 33405 Talence, France
[3]Université de Bordeaux, LaBRI CNRS UMR5800, 33405 Talence Cedex, France

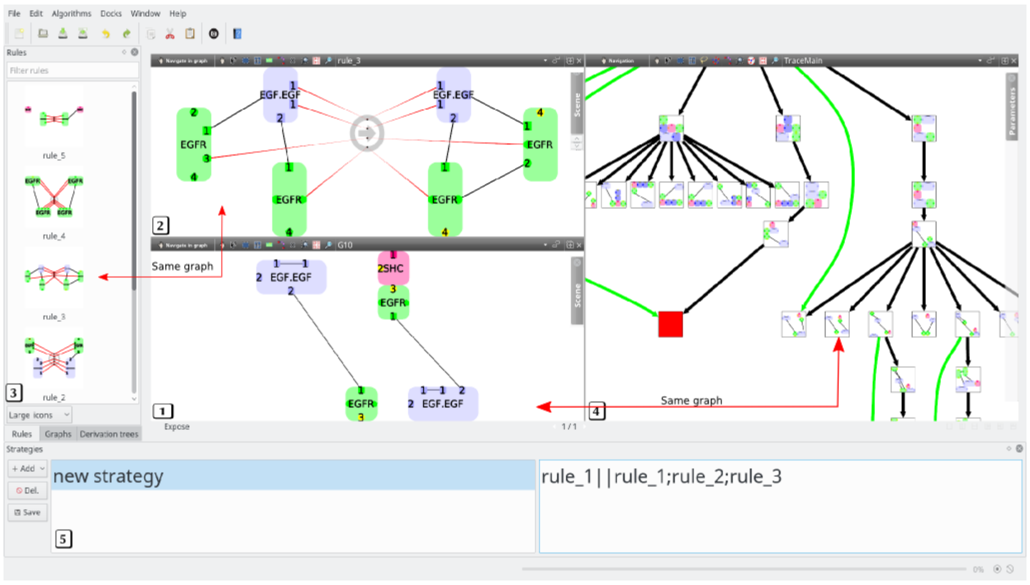
Abstract

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 operational 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 [21, 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



new strategy

rule_1||rule_1;rule_2;rule_3

Figure 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.

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 (see, e.g., [4, 51, 3, 30, 31, 70]).

PORGY [55] 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 provides a *strategy language*. In this paper, we give a formal operational semantics for the language, and show examples of application inside our environment.

Reduction strategies define which (sub)expression(s) should be selected for evaluation and which rule(s) should be applied (see [42, 14, 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.,[72, 68, 48]). Used for a long time in λ-calculus [9], strategies are present in programming languages such as Clean [57], Curry [38], and Haskell [39] and can be explicitly defined to rewrite terms in languages such as ELAN [13], Stratego [71], Maude [49] or Tom [8]. They are also present in graph transformation tools such as PROGRES [65], AGG [27], Fujaba [52], GROOVE [64], GrGen [33] and GP [60, 61]. 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 [24]. 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 port graphs with *attributes* 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 [46]. 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 a term with a rewrite position.

- 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 a visual representation of the derivation tree, together with a user interface that permits to interact with the system. 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 is a revised and expanded version of [3, 30, 31, 70], where PORGY and its strategy language were first presented. Unlike [3, 30], the notion of port graph considered in this paper includes attributes for nodes, ports and also edges, which are formally defined and taken into account in the definition of port graph morphism. 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). Also, in this paper the operational semantics of the language is formally defined.

The paper is organised as follows. In Section 2, we define the concepts of port graph and port graph rewriting. 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 makes a detailed presentation of the Porgy 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, 46, 11, 20, 59, 36]). 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]. We present first the intuitive ideas, followed by the formal definition of port graph rewriting.

2.1 Port graphs

Intuitively, a port graph is a graph where nodes have explicit connection points called *ports*; edges are attached to ports. Nodes, ports and edges are labelled by a set of attributes. For instance, a port may be associated with a state (e.g., active/inactive or principal/auxiliary) and a node may have properties such as colour, shape, etc. Attributes may be used to define both the behaviour of the modelled system and for visualisation purposes (as illustrated in examples later).

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.

Port graphs are transformed by applying port graph rewriting. A *port graph rewrite rule* $L \Rightarrow R$ can itself be seen as a port graph, consisting of two port graphs L and R called the *left-* and *right-hand side* respectively, and one special node $\Rightarrow$, called *arrow node*. The left-hand side of the rule, also called pattern, is used to identify subgraphs in a given graph, which should be replaced by an instance of the right-hand side of the rule. Intuitively, the arrow node describes the way the new subgraph should be linked to the remaining part of the graph, to avoid dangling edges [36, 20] during rewriting.

In order to define formally the notion of 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. A *signature* is used to label the graph.

Definition 1 (Signature)**.** *We define a* signature ∇ *with:*

- $\nabla_{\mathscr{A}}$*, a set of attributes;*
- $\mathcal{X}_{\mathscr{A}}$*, a set of attribute variables;*
- $\nabla_{\mathscr{V}}$*, a set of values;*
- $\mathcal{X}_{\mathscr{V}}$*, a set of value variables.*

We assume that $\nabla_{\mathscr{A}}$*,* $\mathcal{X}_{\mathscr{A}}$*,* $\nabla_{\mathscr{V}}$ *and* $\mathcal{X}_{\mathscr{V}}$ *are pairwise disjoint and there are distinguished elements Name, Arity, Connect, Attach, Interface in* $\nabla_{\mathscr{A}}$*.*

Using the signature ∇, we build *records*, which are sets of attribute-value pairs, sometimes called properties. Records are a central data structure in modern programming languages, and are equally important in software management or computational linguistics, where they are called feature terms.

Definition 2 (Record)**.** *A record r over the signature ∇ is a set $\{(a_1, v_1), \ldots, (a_n, v_n)\}$ of pairs, where $a_i \in \nabla_{\mathscr{A}} \cup \mathcal{X}_{\mathscr{A}}$ and $v_i \in \nabla_{\mathscr{V}} \cup \mathcal{X}_{\mathscr{V}}$ for $1 \leq i \leq n$, each a_i occurs only once in r, and there is one pair where $a_i = Name$. The function Atts applies to records and returns the labels of all the attributes: $Atts(r) = \{a_1, \ldots, a_n\}$ if $r = \{(a_1, v_1), \ldots, (a_n, v_n)\}$. As usual, $r.a_i$ denotes the value v_i of the attribute a_i in r.*

The attribute Name identifies 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 of attributes are assumed to be of basic data types such as *strings*, *int*, *bool*,... We generalise the notion of record to include expressions built using $\nabla_{\mathscr{V}}, \mathcal{X}_{\mathscr{V}}$ and also attributes.

Definition 3 (Record with expressions)**.** *Let $\mathcal{E}_{\nabla}$ denote a set of expressions built using values, attributes and variables in the signature ∇. A* record with expressions *over ∇ is a set $\{(a_1, exp_1), \ldots, (a_n, exp_n)\}$ such that $a_i \in \nabla_{\mathscr{A}} \cup \mathcal{X}_{\mathscr{A}}$, $exp_i \in \mathcal{E}_{\nabla}$ for $1 \leq i \leq n$, each a_i occurs only once and there is one pair where $a_i = Name$.*

For example, $\mathcal{E}_{\nabla}$ may be a set of arithmetic expressions where numbers, variables and attributes are combined using arithmetic operators, or it may be a set of regular expressions built using strings, variables and attributes in ∇, or it may contain both arithmetic and regular expressions.

Attributes, values and expressions will be used in PORGY's strategy language (for example, to build strategies that apply to subgraphs with certain properties). Attributes such as Colour, Shape, etc. may also be used for visualisation purposes; we give examples below.

Records with expressions are records as specified in Definition 2 except that attributes may be associated with expressions instead of just values. We will use records with expressions in right-hand sides of rewrite rules.

In the rest of this paper, we assume that expressions, and more generally records, are well typed: arithmetic expressions use attributes of type *int*, *float*, etc., and regular expressions use attributes of type *string*.

Definition 4 (Port graph)**.** *A* port graph *over a* signature ∇ *is a tuple* $G = (V, P, E, \mathcal{L})$ *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. Edges are undirected and two ports may be connected by more than one edge.*
- $\mathcal{L}$ *is a labelling function that returns, for each element in* $V \cup P \cup E$*, a record such that:*
  - *for each edge* $e \in E$*,* $\mathcal{L}(e)$ *contains an attribute Connect whose value is the pair* $\{p_1, p_2\}$ *of ports connected by* e*.*
  - *for each port* $p \in P$*,* $\mathcal{L}(p)$ *contains an attribute Attach whose value is the node* n *to which the port belongs, and an attribute Arity whose value is the number of edges connected to this port.*
  - *For each node* $n \in V$*,* $\mathcal{L}(n)$ *contains an attribute Interface whose value is the set of names of ports in the node:* $\{\mathcal{L}(p_i).Name \mid \mathcal{L}(p_i).Attach = n\}$*. We assume that* $\mathcal{L}$ *satisfies the following constraint:*

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

By definition 4, nodes with the same name (i.e., the same value for the attribute *Name*) have the same set of port names (i.e., the same interface), with the same attributes but possibly with different values. Variables may be used to denote any value.

Definition 5 (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.

Panel 1 in Figure 1 is an example of a port graph used in a biological case study [3]. It shows 2 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.

2.2 Port Graph Morphism

Let G and H be two port graphs over the same signature ∇. A *port graph morphism* $f : G \to H$ maps nodes, ports and edges of G to those of H such that the attachment of ports and the edge connections are preserved, and all attributes are preserved except for variables in G, which must be instantiated in H. Intuitively, the morphism identifies a subgraph of H that is equal to G except at positions where G has variables (at those positions H could have any value).

Definition 6 (Morphism)**.** *Given two port graphs $G = (V_G, P_G, E_G, \mathcal{L}_G)$ and $H = (V_H, P_H, E_H, \mathcal{L}_H)$ over the same signature ∇, a* morphism *f from G to H, denoted $f : G \to H$, is a family* injective functions $\langle f_V : V_G \to V_H, f_P : P_G \to P_H, f_E : E_G \to E_H \rangle$ *and instantiation functions $f_1 : \mathcal{X}_{\mathscr{A}} \to \nabla_{\mathscr{A}}$, $f_2 : \mathcal{X}_{\mathscr{V}} \to \nabla_{\mathscr{V}}$ such that*

- *$f_V : V_G \to V_H$ is a mapping from the set of nodes of G to the set of nodes of H such that if $n \in V_G$ then $f_1(f_2(\mathcal{L}_G(n))) = \mathcal{L}_H(f_V(n))$.*
- *$f_P : P_G \to P_H$ is a mapping from the set of ports of G to the set of ports of H such that if $p \in P_G$ then $f_1(f_2(\mathcal{L}_G(p))) = \mathcal{L}_H(f_P(p))$ and $f_V(\mathcal{L}_G(p).Attach) = \mathcal{L}_H(f_P(p).Attach)$.*
- *$f_E : E_G \to E_H$ is a mapping from the set of edges of G to the set of edges of H such that if $e \in E_G$ then $f_1(f_2(\mathcal{L}_G(e))) = \mathcal{L}_H(f_E(e))$ and $\overline{f_P}(\mathcal{L}_G(e).Connect) = \mathcal{L}_H(f_E(e)).Connect$ where $\overline{f_P}$ is the extension of the function f_P to sets of ports (i.e., the morphism preserves the edge connections).*

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

This definition ensures that each corresponding pair of nodes, ports and edges between G and H have the same set of attribute labels and associated values, except at positions where there are variables. When using this definition to define rewriting, we will only allow the use of variable labels on one of the graphs: G will be the graph on the left-hand side of the rewrite rule, which may include variable labels, and H will be the graph to be rewritten, without variables.

2.3 Rewriting

We see a *port graph rewrite rule* $L \Rightarrow R$ as a port graph consisting of two subgraphs L and R together with a node (called *arrow* node) that encodes the correspondence between the ports of L and the ports of R. More precisely, we assume that the signature ∇ is such that $\nabla_{\mathcal{V}}$ includes a family of names $\Rightarrow_k$ (where k denotes the number of ports attached to the arrow node). Each of the ports attached to the arrow node has an attribute *Type* $\in \nabla_{\mathscr{A}}$, which can have 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. We give details below.

Definition 7 (Port graph rewrite rule)**.** *A port graph rewrite rule is a port graph consisting of:*

- *two port graphs L and R over the signature ∇, called* left-hand side *and* right-hand side*, respectively, such that all the variables in R occur in L, and R may contain records with expressions;*
- *and 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 Name $\Rightarrow_n$: there is indeed a family of arrow nodes, where n is the number of ports in the interface. Each port in the arrow node has an attribute Type, *which can have value bridge, blackhole or wire, satisfying the following conditions:*

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).*

2. *A port of type* blackhole *must have edges connecting it only to L (at least one edge).*

3. *A port of type* wire *must have exactly two edges connecting to L and no edge connecting to R.*

The edges connecting the arrow node with L and R are used to control the rewiring that occurs during a rewriting step with the rule (see Definition 9). This ensures that no dangling edges (see [36, 20]) arise when a rewriting step takes place (see Property 2.3.1 below).

Intuitively, the ports of type bridge in the arrow node form a bridge between both sides of the rewrite rule, and indicate whether the corresponding port in L survives the reduction. A port in L connected to a blackhole port in the arrow node does not survive the reduction; all edges connected to this port in the graph are deleted when the reduction step takes place (as defined later). A port of type wire connected to two ports p_1 and p_2 in the *left-hand side* triggers a particular rewiring, which takes all the ports that are connected to (the image of) p_1 in the redex and creates an edge for each of those ports to each of the ports connected to (the image of) p_2. If a port in L is not connected to the arrow node, we expect that the only edges incident to this port in the redex are the images of the edges in L (thus, if a port in L does not have any incident edges, then it should not have any incident edges in the redex either). This is to avoid dangling edges after rewriting (see the definition of matching below, Definition 8). When the correspondence between ports in the left- and right-hand sides of the rule is graphically obvious, we may omit drawing the ports and edges involving the arrow node.

This definition generalises the original definition given in [2], by including case (3) in Definition 7 above, inspired by the notion of rewriting defined for Interaction Nets [46]. This allows defining more efficient rewrite rules, and additionally Interaction Net rules become a particular case of port graph rewrite rules.

Figure 2 shows four examples of rules. The effect of the rule in the top left is easily read from the diagram: assuming a port graph contains a subgraph with three nodes in the given configuration, an edge connecting two green "EGFR" nodes is added. Moreover, the records in the ports named "4" of the "EGFR" nodes are also modified (the Colour attribute changed from green to yellow). The arrow node (centre of the drawing) and the arrow edges connected to it are easily identifiable.

Definition 8 (Match). *Let $L \Rightarrow R$ be a port graph rewrite rule and G a port graph. We say a* match *$g(L)$ of the left-hand side (also called a* redex*) is found if:*

- *There is a port graph morphism g from L to G; hence $g(L)$ is a subgraph of G.*

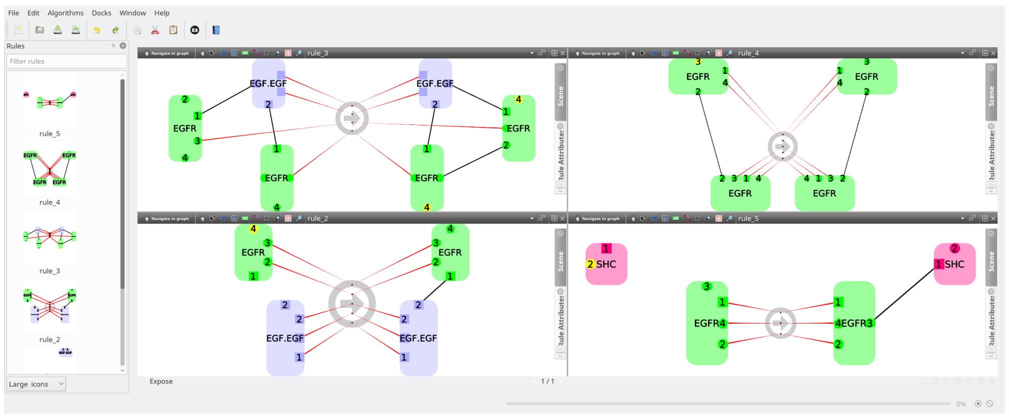


Figure 2: The four rules of the model shown in Fig. 1. The rule at the top left is the rule shown in Panel 2 of Fig. 1.

- *For each port in L that is not connected to the arrow node, its corresponding port in $g(L)$* must not *be an extremity in the set of edges of $G - g(L)$.*

This last point ensures that ports in L that are not connected to the arrow node are mapped to ports in $g(L)$ that have no edges connecting them with ports outside the redex, to avoid dangling edges in rewriting steps (Property 2.3.1). 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 (see Section 6). For ports in L connected to an arrow port there is no check on arities.

Definition 9 (Rewriting step)**.** *A* rewriting step *on G using a rule $L \Rightarrow R$ and a morphism $g : L \to G$, written $G \to^{g}_{L\Rightarrow R} G'$, 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'.*

Arrow nodes in rewrite rules can also have attributes; this is useful in particular to guide the visualisation of rewriting steps (steps with different rules may be displayed differently). For example, in Porgy we use an attribute of the arrow node to select the visualisation algorithm.

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 via a port graph rewrite rule $L \Rightarrow R$ is a port graph.

Property 2.3.1. *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 connected to ports in $g(L)$ are deleted or redirected to ports in R_c, according to the definition of rewriting step (Definition 9). 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, then there is no edge connected to p in $g(L)$ except for the edges in $g(L)$ (by definition of match). The edges in $g(L)$ are deleted in the deletion phase. $\square$

Several injective 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.

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 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 Derivation tree and strategies

A *derivation*, or computation, is a sequence $G \rightarrow^{*}_{\mathcal{R}} G'$ 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 10 (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 $G \rightarrow_{\mathcal{R}} G_i$. 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 of Abstract Reduction System.

An *Abstract Reduction System (ARS)* [67, 42, 14] is a labelled oriented graph $(\mathcal{O}, \mathcal{S})$ with a set of labels $\mathcal{L}$. The nodes in $\mathcal{O}$ are called *objects*. The oriented labelled edges in $\mathcal{S}$ are called *steps*: $a \overset{\phi}{\to} b$ or (a, ϕ, b), with *source* a, *target* b and *label* ϕ. Derivations are composition of steps.

For a given ARS $\mathcal{A}$, a *finite $\mathcal{A}$-derivation* is denoted $\phi : a_0 \overset{\phi_0}{\to} a_1 \overset{\phi_1}{\to} a_2 \ldots \overset{\phi_{n-1}}{\to} a_n$ or $a_0 \overset{\phi}{\to} a_n$, where $n \in \mathbb{N}$. The *source* of ϕ is a_0 and its domain $Dom(\phi) = \{a_0\}$. The *target* of ϕ is a_n and applying ϕ to a_0 gives the singleton set $\{a_n\}$, which is denoted $\phi \bullet a_0 = \{a_n\}$.

Abstract strategies are defined in [42] and in [14] as follows: for a given ARS $\mathcal{A}$, an *abstract strategy* ζ is a subset of the set of all derivations (finite or not) of $\mathcal{A}$. The notions of domain and application are generalised: $Dom(\zeta) = \bigcup_{\phi \in \zeta} Dom(\phi)$ and $\zeta \bullet a = \{b \mid \exists \phi \in \zeta \text{ such that } a \overset{\phi}{\to} b\} = \{\phi \bullet a \mid \phi \in \zeta\}$.

This very general definition of abstract strategies is called *extensional* in [14] in the sense that a strategy is defined explicitly as a set of derivations of an abstract reduction system. The concept is useful to understand and unify reduction systems and deduction systems as explored in [42]. But abstract strategies do not capture another point of view, also frequently adopted in rewriting: a strategy is a partial function that associates to a reduction-in-progress, the possible next steps in the reduction sequence. Here, the strategy as a function depends only on the object and the derivation so far. This notion of strategy coincides with the definition of strategy in sequential path-building games, with applications to planning, verification and synthesis of concurrent systems. This remark leads to the following *intentional* definition given in [14].

A *(memoryless) intentional strategy* for $\mathcal{A} = (\mathcal{O}, \mathcal{S})$ is a partial function λ from $\mathcal{O}$ to $2^{\mathcal{S}}$ such that for every object a, $\lambda(a) \subseteq \{\phi \mid \phi \in \mathcal{S}, Dom(\phi) = a\}$.

As described in [14], an intentional strategy λ naturally generates an abstract strategy, called its *extension*: this is the abstract strategy ζ_λ consisting of the following set of derivations:
$\forall n \in \mathbb{N}, \phi : a_0 \overset{\phi_0}{\to} a_1 \overset{\phi_1}{\to} a_2 \ldots \overset{\phi_{n-1}}{\to} a_n \in \zeta_\lambda$ iff $\forall j \in [0, n], \quad (a_j \overset{\phi_j}{\to} a_{j+1}) \in \lambda(a_j)$.
This extension may obviously contain infinite derivations; in such a case it also contains all the finite derivations that are prefixes of the infinite ones, and so is closed under taking prefixes.

A challenge to address is to define languages for describing intentional strategies. We propose in the following such a language in the case of port graph rewriting.

3 Strategic graph programs

Port graphs are a powerful, versatile and intuitive formalism. The notion of port can be applied to many fields (e.g., to model protein receptors in biochemistry, communication endpoints in computer networking, etc.). Thanks to attributes available for nodes, ports and edges, data can be stored within port graphs, and used with rewrite rules and the right kind of strategy, in order to guide the rewriting process. The inherent concurrence and non-determinism of rewriting are also useful for modelling most of complex systems. Thus, port graphs and port graph rewriting provide a good basis to define a modelling language.

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 is enabled/disabled), a set of rewriting 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

First 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 $G = (V_G, P_G, E_G, \mathcal{L}_G)$ and $H = (V_H, P_H, E_H, \mathcal{L}_H)$ be port graphs over the signature ∇. H is a subgraph of G if $V_H \subseteq V_G$, $P_H \subseteq P_G$, $E_H \subseteq E_G$, $\mathcal{L}_H = \mathcal{L}_G|_{V_H \cup P_H \cup E_H}$, that is, $\mathcal{L}_H$ is the restriction to H of the labelling function of G.

Definition 11 (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 12 (Located rewrite rule)**.** *A* located rewrite rule *is given by a port graph rewrite rule $L \Rightarrow R$, and optionally a subgraph W of L and two disjoint subgraphs M and N of R. It is denoted $L_W \Rightarrow R_M^N$. We write $G_P^Q \rightarrow^g_{L_W \Rightarrow R_M^N} G'^{Q'}_{P'}$ and say that the located graph G_P^Q* rewrites to $G'^{Q'}_{P'}$ using $L_W \Rightarrow R_M^N$ at position P avoiding Q*, if $G \rightarrow_{L \Rightarrow R} G'$ with a 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 $P' = (P \setminus g(L)) \cup g(M)$, $Q' = Q \cup g(N)$; if M (resp. N) are not provided then we assume $M = R$ (resp. $N = \emptyset$).*

In general, for a given located rule $L_W \Rightarrow R_M^N$ and located graph G_P^Q, more than one 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 Strategies

To control the application of the rules, we introduce a strategy language with the syntax shown in Table 1.

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 [13], Stratego [71] and Tom [8]. Focusing operators are not present in term rewriting languages (where the implicit assumption is that the rewrite position is defined by traversing the term from the root downwards). 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 Rules constructs defining how to apply rules, then Positions constructs, which allow us to specify subgraphs in a given located graph. We finally define Compositions constructs combining strategies.

Rules Constructs. The simplest transformation is a located rule, which can only be applied to a located graph G_P^Q if at least 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.

Let L, R be port graphs; M, N subgraphs of R; W a subgraph of L;
$n \in \mathbb{N}$; $\pi_{i=1\ldots n} \in [0,1]$; $\sum_{i=1}^{n} \pi_i = 1$; let *attribute* be an attribute label in $\nabla_{\mathscr{A}}$;
$e \in \mathcal{E}_{\nabla}$ a valid expression without variables;

Rules	**(Transformations)**	T	$::=$	$L_W \Rightarrow R_M^N \mid (T \parallel T)$
			$\mid$	$\texttt{ppick}(T_1, \pi_1, \ldots, T_n, \pi_n)$
	(Applications)	A	$::=$	$\texttt{all}(T) \mid \texttt{one}(T)$
Positions	**(Focusing)**	F	$::=$	$\texttt{crtGraph} \mid \texttt{crtPos} \mid \texttt{crtBan}$
			$\mid$	$F \cup F \mid F \cap F \mid F \setminus F \mid (F) \mid \emptyset$
			$\mid$	$\texttt{ppick}(F_1, \pi_1, \ldots, F_n, \pi_n)$
			$\mid$	$\texttt{property}(F, \rho) \mid \texttt{ngb}(F, \rho)$
	(Determine)	D	$::=$	$\texttt{all}(F) \mid \texttt{one}(F)$
	(Update)	U	$::=$	$\texttt{setPos}(D) \mid \texttt{setBan}(D)$
			$\mid$	$\texttt{update}(\mathit{function}\{\mathit{parameters\_list}\})$
Properties	**(Properties)**	ρ	$:=$	$\mathit{Elem}, \mathit{Expr}$
		Elem	$:=$	$\texttt{node} \mid \texttt{edge} \mid \texttt{port}$
		Expr	$:=$	$\mathit{attribute}\ \mathit{Relop}\ e \mid \texttt{true}$
		Relop	$:=$	$== \mid\ != \mid\ > \mid\ <$
			$\mid$	$>= \mid\ <= \mid\ =\sim$
Compositions	**(Comparison)**	C	$::=$	$F = F \mid F\ !=\ F \mid F \subset F \mid \texttt{isEmpty}(F)$
			$\mid$	$\texttt{match}(T)$
	(Strategies)	S	$::=$	$\mathsf{id} \mid \mathsf{fail} \mid A \mid U \mid C \mid S; S$
			$\mid$	$\texttt{if}(S)\texttt{then}(S)\texttt{else}(S) \mid (S)\texttt{orelse}(S)$
			$\mid$	$\texttt{repeat}(S)[(n)] \mid \texttt{while}(S)[(n)]\texttt{do}(S)$
			$\mid$	$\texttt{try}(S) \mid \texttt{not}(S)$

Table 1: Syntax of the Strategy Language.

When probabilities $\pi_1, \ldots, \pi_n \in [0,1]$ are associated to rules $T_1, \ldots, T_n$ such that $\pi_1 + \ldots + \pi_n = 1$, the strategy $\texttt{ppick}(T_1, \pi_1, \ldots, T_n, \pi_n)$ picks one of the rules for application, according to the given probabilities. If rules 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.

Positions Constructs. The grammar for F (see Table 1) 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 the positions P and Q in the current located graph (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.
  - `property`(F, ρ) is used to select elements of a given graph that satisfy a certain property, specified by ρ. It can be seen as a filtering construct: if the expression F generates a subgraph G' then `property`(F, ρ) returns only the nodes and/or edges from G that satisfy the decidable property $\rho = \mathit{Elem}, \mathit{Expr}$. 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 the examples below. Note that if an edge is selected, the nodes at each extreme are automatically included in the resulting graph.
    - `property`$(F,$ `node`$, Name == \text{“}\mathit{Add}''$) returns all the nodes of the subgraph defined by the expression F that are labelled *Add*.
    - `property`$(F,$ `port`$, \mathit{Active} == \text{“}\mathit{true}''$) returns all the nodes of the subgraph defined by the expression F that have at least one port with an attribute labelled *Active* that is set to the value *true*.
    - `property`$(F,$ `node`$, \mathit{Colour} == Valid)$ returns all the nodes of the subgraph defined by the expression F that have the same values for the attributes Colour and $Valid$.
    - `property`$(F,$ `edge`$, \mathit{State} > 3)$ returns all the edges (including the nodes at their extremities) of the subgraph defined by the expression F that have an attribute named *State* with a value greater than 3.
    - `property`$(F,$ `node`$, Name =\sim \text{“\^{}Num[0-9]\$}''$) returns all the nodes of the subgraph defined by the expression F with a name valid over the regular expression “ˆNum[0-9]$” (the name must start by the string “Num” and terminates by a number). This syntax is inspired by languages such as Perl, Java or the more recent `C++11`.
  - `ngb`(F, ρ) returns a subset of the neighbours (i.e., adjacent nodes) of F according to ρ. When `edge` is used (i.e., when we write `ngb`$(F,$ `edge`$, \mathit{Expr})$, it returns all the neighbours of F connected to F via edges which satisfy the expression *Expr*.

    We give some examples below; in each case we assume that F is an expression that specifies a graph.

  * $\texttt{ngb}(F, \texttt{node}, \mathit{Name} == \mathit{Add})$, returns all the nodes that are adjacent to nodes labelled *Add* in F but are not in F themselves (i.e., it returns the neighbours of the nodes in F labelled *Add*).
  * $\texttt{ngb}(F, \texttt{port}, \mathit{Active} == \mathit{true})$ returns all the nodes not already in F that are adjacent to nodes that have a port with an attribute labelled *Active* set to the value *true* .
  * $\texttt{ngb}(F, \texttt{edge}, \mathit{State} > 3)$ returns the nodes (not already in F) at the other extremity of edges connected to nodes in F, where the edge has an attribute *State* with a value greater than 3.
  * $\texttt{ngb}(F, \texttt{edge}, \mathit{true})$ returns all the nodes adjacent to nodes in F and not already in F.
- $\cup$, $\cap$ 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, if we want to focus on active principal ports, we can specify

$$\texttt{all}(\texttt{property}(F, \texttt{Port}, \mathit{Name} == \mathit{Principal}) \cap \texttt{property}(F, \texttt{port}, \mathit{Active} == \mathit{true}))$$

  (the latter selects nodes with ports that have an attribute labelled *Active* that is set to the value *true*). $\texttt{property}(\mathit{Pos}, \texttt{node}, \mathit{Name} == \mathit{Mult}) \cap \texttt{property}(\mathit{Pos}, \texttt{Port}, \mathit{Name} == \mathit{Aux})$ returns all the nodes in the subgraph denoted by *Pos* that are labelled *Mult and* that have a port labelled *Aux*. We give more examples in Section 4.
- When probabilities $\pi_1, \ldots, \pi_n \in [0, 1]$ are associated to subgraphs denoting positions $F_1, \ldots, F_n$ such that $\pi_1 + \ldots + \pi_n = 1$, the strategy

$$\texttt{ppick}(F_1, \pi_1, \ldots, F_n, \pi_n)$$

  picks one of the positions for application, according to the given probabilities.

- Determine Constructs.
  $\texttt{one}(F)$ returns one node in F chosen at random and $\texttt{all}(F)$ returns the full F. When not specified, F stands for $\texttt{all}(F)$.

- 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). $\texttt{update}(\mathit{function\_name}\{\mathit{parameters\_list}\})$ 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, selecting 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).

Compositions 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 the same port graph, otherwise returns fail. F != F returns id if both expressions do not denote the same port graph, 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: `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 the strategy `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 eventually 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.

  – (S)`orelse`(S') applies S if possible, otherwise applies S'. It fails if both S and S' fail.

  – `repeat`(S)[max n] simply iterates the application of S until it fails, but, if max n is specified, then the number of repetitions cannot exceed n.

  – `while`(S)`do`(S')[max n] keeps on sequentially applying S' while the expression S succeeds on a copy of the graph. If S fails, then id is returned. If max n is specified, then the number of iterations cannot exceed n.

  – `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).

Note that conjunction and disjunction of comparisons can be simulated with other constructs: (C)`and`(C') by C;C', (C)`or`(C') by (C)`orelse`(C').

4 Examples

In this section, the expressivity of the language is illustrated through examples taken in three different domains: graph property testing, social network simulation, term rewriting strategies. The examples can be downloaded from `http://tulip.labri.fr/Porgy/MSCS/`.

4.1 Graph testing: Connected graph and spanning tree

The first Strategy 1 below is used to check whether a graph is connected (i.e., made of only one connected component). If not, the strategy ends with a failure. 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(crtGraph);
        one(start);
visit-neighbours:
        repeat(one(walk)) ;
check-all-nodes-visited:
        setPos(crtGraph);
        not(one(start))
```

Note that this example can also be used to compute a spanning tree from a graph by simply making a small change to 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 a spanning tree from each graph node as the root of the tree, one has to change `one(start)` to `all(start)` in the `pick-one-node`

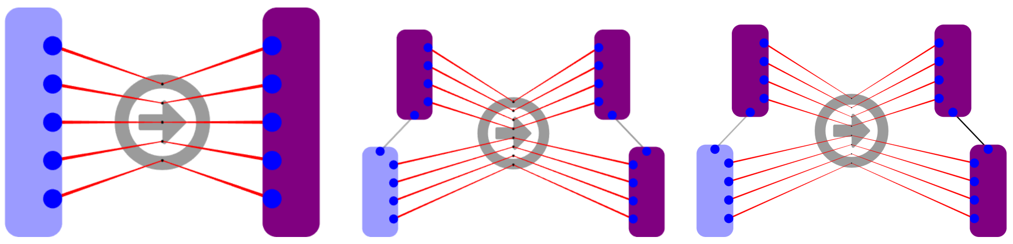

(a) Rule for selecting a starting node. (b) Rule for computing connected component. (c) Rule for computing a spanning tree. Only the edge in the right-hand side is modified.

Figure 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).

strategy (See Strat. 2). Fig. 5 shows on another example the obtained derivation tree and a close-up on a branch shows as a series of small-multiples.

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

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

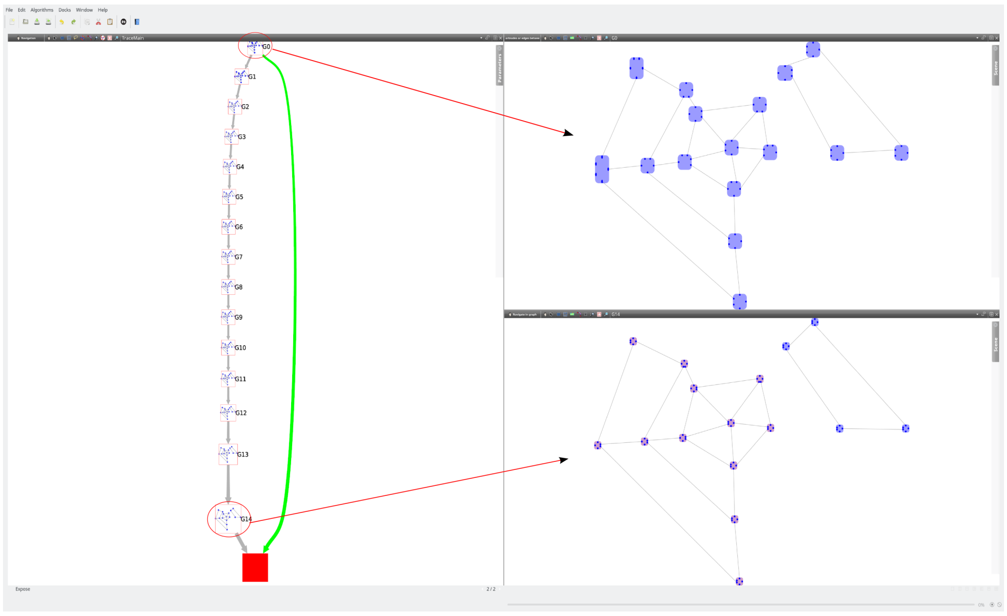
4.2 Social network behaviour simulation

Social networks simulation offers many interesting questions. We focus here on simulation of acquaintance and influence propagation.

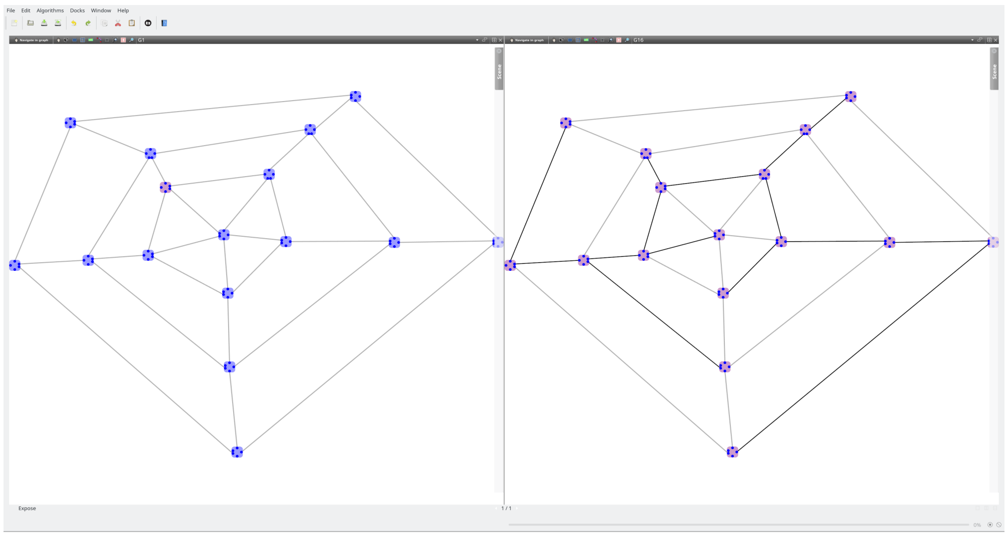
4.2.1 Dynamics of acquaintanceship.

[40] present rules and strategies to model dynamics in social networks using a probabilistic graph-based approach. 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 the paper cited above.

This example consits 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.



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



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

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

4.2.2 Influence Propagation in Social Networks.

A simplified definition of propagation in a network is as follows: when an individual performs a specific action like announcing an event, she/he becomes active and informs

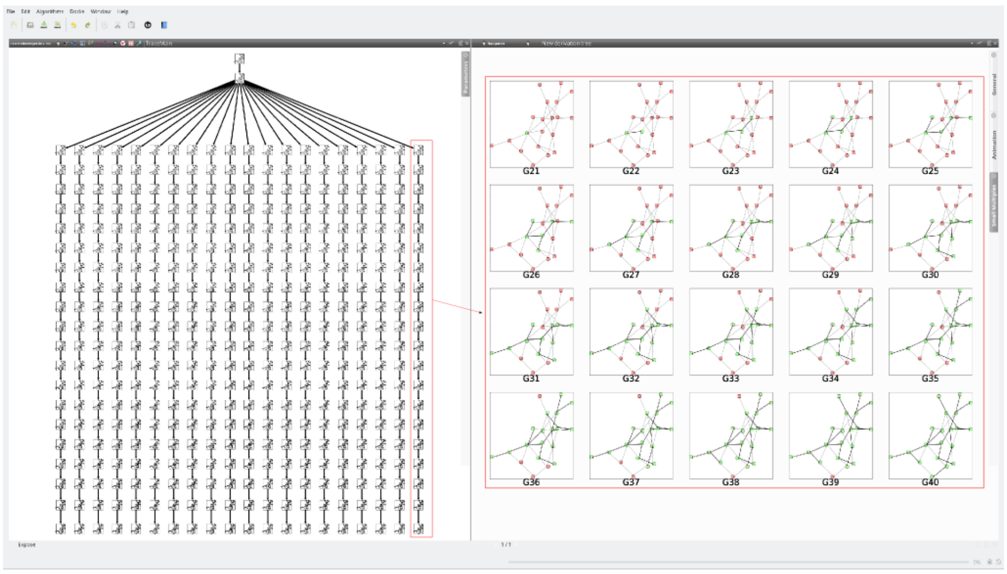


Figure 5: The left panel shows the whole derivation tree obtained with Strategy 2 on another example. The right panel is a close-up on a branch showed as small-multiples.

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

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

her/his neighbours of its 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. In [70], we present a visual approach to compare propagation models in social networks; graph rewriting is used as a common 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. We give an example in Figure 8.

In this example,

- Each node n has an attribute *Active* which indicates whether it contributes to the propagation or not. It is coupled with the *Colour* attribute which takes accordingly green or red values. The node n has also a *Sigma* attribute that measures the maximum influence withstood by n from its active neighbours until now.

- An edge e that connects two ports p, p' in respective nodes n, n' has an attribute $Influence$ which indicates the influence of n' (i.e. $\mathcal{L}(p').Attach$) on n (i.e. $\mathcal{L}(p).Attach$). The edge e has also a Boolean attribute $Marked$ that is initially false and becames true

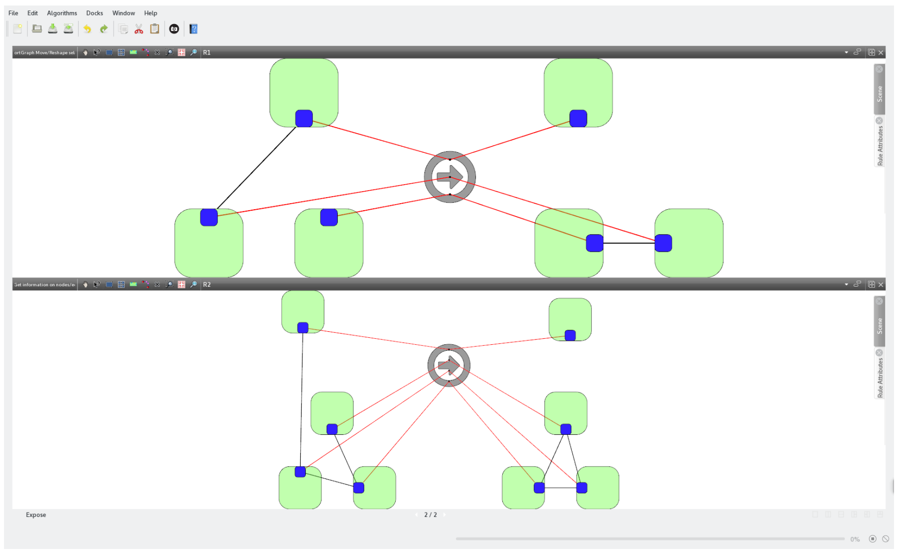


Figure 6: PORGY specification of the two rules presented in [40] to model the dynamics of acquaintanceship in social networks. The top rule ($R1$) shows breaking up of an acquaintanceship for a more advantageous one and the bottom rule ($R2$) shows the creation of stronger relationships by forming triads.

when n is active, n' is inactive and 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

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

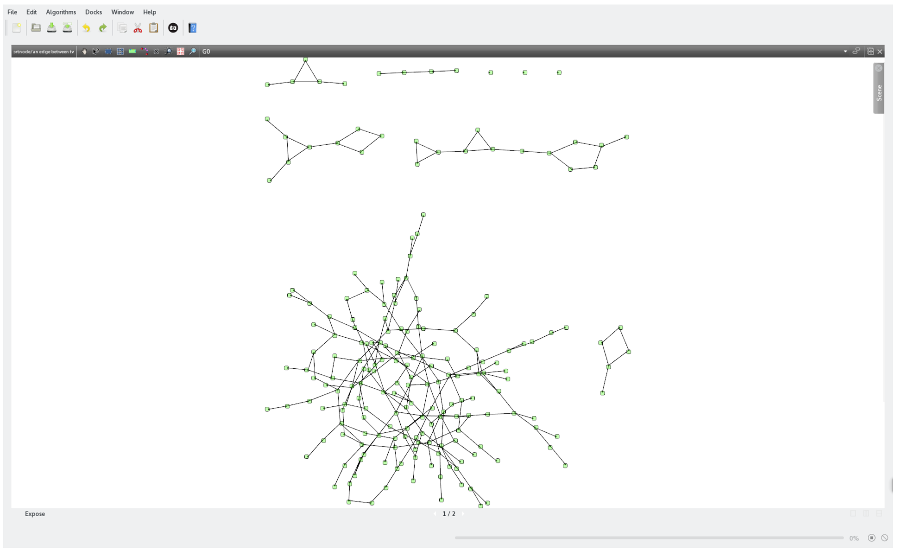
where r is a random value between 0 and 1. The formula is stored in the the rule (R1) in Fig. 8 (on the left).

When this rule is applied on a pair of nodes active(n)/non active($\overline{n}$) (green/red): *a)* we generate a random number $r\ \in]0,1]$; *b)* we store in the attribute $\mathcal{L}(n).Sigma$ the new value of $Sigma$ computed with the given formula; and *c)* using the *Marked* attribute, we mark the edge e linking n to $\overline{n}$ to prevent the selection of this particular pair configuration 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.

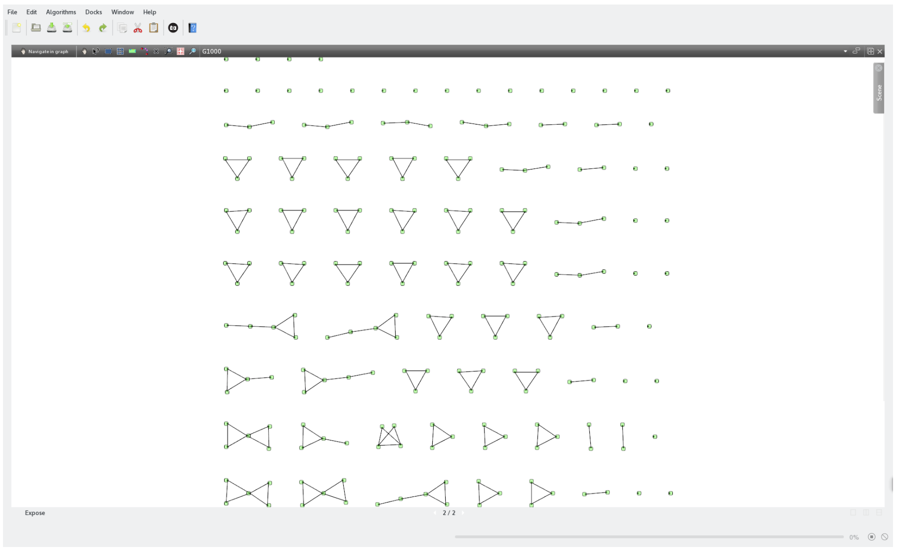
Once every pair on active/inactive neighbours has been tried, if n is sufficiently influenced (i.e. $\mathcal{L}(n).Sigma > 1$), it becomes active with the second rule (R2) given on the right of Fig. 8. This behaviour is summarised in Strategy 4.

Strategy 4: Influence propagation in social network.

```
repeat($R1$);
setPos(property(crtGraph, node, $Sigma \geq$ “1″));
repeat($R2$)
```



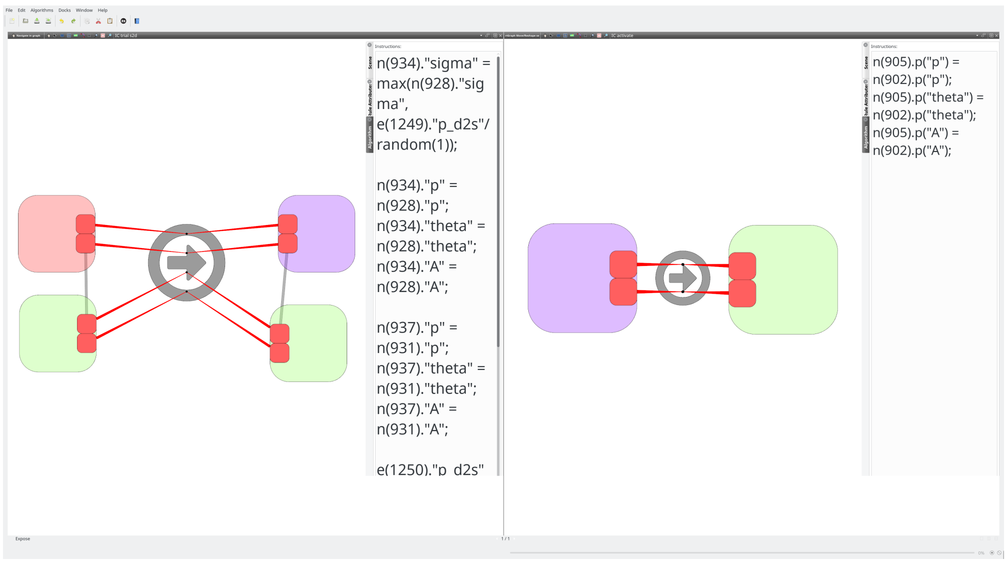
(a) A random ER graph with 200 vertices and edges.



(b) Simulation results.

Figure 7: A possible result obtained after running Strategy 3.

The starting set of active nodes can be chosen randomly or, for instance, by calling an external procedure with the `update()` construct to compute the network central nodes. One can think that the propagation will be faster from the central nodes. This example illustrates how records' expressions may be used to compute attribute values and updated through application of rules.



```
n(934)."sigma" =
max(n(928)."sig
ma",
e(1249)."p_d2s"/
random(1));

n(934)."p" =
n(928)."p";
n(934)."theta" =
n(928)."theta";
n(934)."A" =
n(928)."A";

n(937)."p" =
n(931)."p";
n(937)."theta" =
n(931)."theta";
n(937)."A" =
n(931)."A";

e(1250)."p_d2s"
```

```
n(905).p("p") =
n(902).p("p");
n(905).p("theta") =
n(902).p("theta");
n(905).p("A") =
n(902).p("A");
```

Figure 8: Rules used to express a propagation model and the complete expressions associated to the right-hand side (see [70] for more details about the expressions). Active nodes are depicted in green and visited nodes in purple. Red nodes are in an inactive state (however, they may have been visited already). The rule on the left (R1) indicates that when an activated node is connected to an inactived node, it tries to influence it. If it succeeds, the second rule (R2) makes this node active.

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 [13] or Stratego [71][17]; in PORGY this idea is generalised to graphs that are not necessarily trees.

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 label 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. The arity of the function symbol that labels the node must be equal to the number of Child ports in the node; function symbols of arity 0 are leaves in the tree (they have no Child port).

Outermost rewriting on trees. The focusing expression

$$start \triangleq \texttt{crtGraph} \setminus \texttt{property}(\texttt{crtGraph}, \texttt{port}, Name == \text{``}Parent''\text{)}$$

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($start$);
while(not(isEmpty(crtPos)))do(
        if(match($R$))then(
            one($R$); setPos($start$)
        )else(
            setPos(ngb(crtPos, port, $Name =\sim$ "^$Child[1-9]\$$"))
        )
)
```

The strategy starts by focusing on the root node, using setPos(*start*). We then apply R as close to the root as possible: if R can be applied at the root, then we apply it and set the position back to the root of the tree. Otherwise,

$$\texttt{setPos}(\texttt{ngb}(\texttt{crtPos}, \texttt{port}, Name =\sim \text{``\^{}}Child[0-9]\$''))$$

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.

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

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

Strategy 6: Innermost rewriting on trees.

```
setPos($start$); setBan($NonLeaf$);
while(not(isEmpty(crtPos)))do(
      if(match($R$))then(
         one($R$); setPos($start$); setBan($NonLeaf$)
      )else(
         setPos(ngb(crtPos, port, $Name ==$ “$Parent$”));
         setBan(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(ngb(crtPos, port,` $Name ==$ “$Parent$”`))` and the banned subgraph is updated again to all remaining elements of the tree (with `setBan(crtBan \ crtPos)`).

5 Semantics of strategic graph programs

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

Definition 13 (Strategic graph program)**.** *A* strategic graph program *consists of a finite set of located rewrite rules $\mathcal{R}$, a strategy expression S (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 [58]. The idea is to use the strategy expression S to decide which rewrite 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 14. *A* configuration *C is a multiset $\{O_1, \dots, O_n\}$ where each O_i is a graph program.*

Definition 15 (Transitions)**.** *The transition relation $\longmapsto$ is a binary relation on configurations defined as follows:*

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

if $O_k \mapsto \{O'_{k_1}, \dots, O'_{k_m}\}$ $(1 \leq k \leq n)$.

The transition relation $\mapsto$ 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 $\{O_1, \dots, O_k, \dots, O_n\}$, there may be several values of k such that a $\mapsto$-transition can be applied to the strategic programs 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 $\mapsto$-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 15), 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 $\mapsto$-step $O_k \mapsto \{O'_{k_1}, \dots, O'_{k_m}\}$, 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 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 $\mapsto$ on individual strategic graph programs is defined by induction, for each construct of the strategy language.

Rules 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 16. *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 $g_1, \dots, g_k$ are pairwise different.*

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

$$\frac{}{[\mathtt{all}(L_W \Rightarrow R_M^N), G_P^Q] \mapsto \{[\mathsf{id}, G_1{}_{P_1}^{Q_1}], \ldots, [\mathsf{id}, G_k{}_{P_k}^{Q_k}]\}}\; LS_{L_W \Rightarrow R_M^N}(G_P^Q) = \{G_1{}_{P_1}^{Q_1}, \ldots, G_k{}_{P_k}^{Q_k}\}$$

$$\frac{}{[\mathtt{all}(L_W \Rightarrow R_M^N), G_P^Q] \mapsto \{[\mathsf{fail}, G_P^Q]\}}\; LS_{L_W \Rightarrow R_M^N}(G_P^Q) = \emptyset$$

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

Concurrent application of two rewrite rules is achieved through the operator $||$, which works on rules only (not on general strategies). To define the semantics of $\mathtt{all}(L_1{}_{W_1} \Rightarrow R_1{}_{M_1}^{N_1} || \ldots || L_k{}_{W_k} \Rightarrow R_k{}_{M_k}^{N_k})$, we define a new rule $(L_1 \cup \cdots \cup L_k)_{W_1 \cup \ldots \cup W_k} \Rightarrow (R_1 \cup \cdots \cup R_k))_{M_1 \cup \cdots \cup M_k}^{N_1 \cup \cdots \cup N_k}$, where the arrow node contains all the ports and edges of the arrow nodes of individual rules $L_i{}_{W_i} \Rightarrow R_i{}_{M_i}^{N_i}$ (for $1 \leq i \leq k$). Note that the union operator works on graphs, so even if two nodes have the same label, 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_P^Q for

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

or *legal parallel set*, denoted $LPS_{L_1{}_{W_1} \Rightarrow R_1{}_{M_1}^{N_1} || \ldots || L_k{}_{W_k} \Rightarrow R_k{}_{M_k}^{N_k}}(G_P^Q)$, contains all the located graphs $G_i{}_{P_i}^{Q_i}$ $(1 \leq i \leq n)$ such that $G_P^Q \rightarrow^{g_i}_{(L_1 \cup \cdots \cup L_k)_{W_1 \cup \ldots \cup W_k} \Rightarrow (R_1 \cup \cdots \cup R_k))_{M_1 \cup \cdots \cup M_k}^{N_1 \cup \cdots \cup N_k}} G_i{}_{P_i}^{Q_i}$, $g_1, \ldots, g_n$ 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 $\mathtt{all}(L_1{}_{W_1} \Rightarrow R_1{}_{M_1}^{N_1} || \ldots || L_k{}_{W_k} \Rightarrow R_k{}_{M_k}^{N_k})$.

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

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

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

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

If all the sets W_i in the rules participating in the parallel construct are non empty, then $[\mathtt{all}(L_1{}_{W_1} \Rightarrow R_1{}_{M_1}^{N_1} || \ldots || L_k{}_{W_k} \Rightarrow R_k{}_{M_k}^{N_k}), G_P^Q] \mapsto C$ if

$$[\mathtt{all}((L_1 \cup \cdots \cup L_k)_{W_1 \cup \ldots \cup W_k} \Rightarrow (R_1 \cup \cdots \cup R_k))_{M_1 \cup \cdots \cup M_k}^{N_1 \cup \cdots \cup N_k}, G_P^Q] \mapsto C$$

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.

Positions Constructs. The commands that are used to specify and update positions are `setPos`(), `setBan`(), `update`(*function_name*{*parameters_list*}). The first two rely on focusing expressions, generated by the grammar for the non terminal F in Fig. 1, which have a functional semantics. In other words, an expression F denotes a function that applies to the current located graph G_P^Q, 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.

$$\begin{array}{lcll}
\texttt{crtGraph}(G_P^Q) & = & G & \texttt{crtPos}(G_P^Q) \;=\; P \qquad \texttt{crtBan}(G_P^Q) \;=\; Q \\
\texttt{property}(F,\rho)(G_P^Q) & = & G' & \text{where } G' \text{ consists of all nodes in } F(G_P^Q) \\
 & & & \text{satisfying } \rho \\
\texttt{ngb}(F,\rho)(G_P^Q) & = & G' & \text{where } G' \text{ consists of a subset of the nodes} \\
 & & & \text{adjacent to nodes in } F(G_P^Q)\text{: } \rho \text{ is evaluated on} \\
 & & & F(G_P^Q) \text{ to compute the adjacent nodes} \\
(F_1 \; op \; F_2)(G_P^Q) & = & \multicolumn{2}{l}{F_1(G_P^Q) \; op \; F_2(G_P^Q) \text{ where } op \text{ is } \cup, \cap, \setminus}
\end{array}$$

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

$$\frac{}{[\texttt{setPos}(\texttt{all}(F)), G_P^Q] \mapsto \{[\mathsf{id}, G_{P'}^Q]\}} \; F(G_P^Q) = P'$$

$$\frac{}{[\texttt{setBan}(\texttt{all}(F)), G_P^Q] \mapsto \{[\mathsf{id}, G_P^{Q'}]\}} \; F(G_P^Q) = Q'$$

$$\frac{}{[\texttt{update}(f\{list\}), G_P^Q] \mapsto \{[\mathsf{id}, \overline{G}_{P'}^{Q'}]\}} \; f(list, G_P^Q) = \overline{G}_{P'}^{Q'}$$

The located graph $\overline{G}_{P'}^{Q'}$ has the same structure as G_P^Q, 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.

Compositions constructs.

- **Comparison constructs.** Those are expressions of the form $F = F'$, $F! = F'$, $F \subset F'$. Recall that an expression F denotes a function that applies to the current located graph G_P^Q, and computes a subgraph of G. As mentioned above, the function denoted by the expression F is also called F.

For every construct C of the form F `op` F' in this class, there are two rules:

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

As mentioned earlier, $\mathtt{isEmpty}(F)$ is defined as $F = \emptyset$ and $\mathtt{match}(T)$ has the same semantics as $\mathtt{if}(\mathtt{one}(T))\mathtt{then}(\mathsf{id})\mathtt{else}(\mathsf{fail})$.

- 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:

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

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

The rule for sequences ensures that S_1 is applied first.

- **Conditional.** The behaviour of the strategy $\mathtt{if}(S_1)\mathtt{then}(S_2)\mathtt{else}(S_3)$ 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 graph, otherwise S_3 is applied.

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

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

Note that S_1 may produce more than one result, and some results could be success and others failure. 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.** $(S_1)\mathtt{orelse}(S_2)$ 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.

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

$$\frac{[S_1, G_P^Q] \longmapsto \{[\mathsf{fail}, G'_1], \ldots, [\mathsf{fail}, G'_n]\}}{[(S_1)\texttt{orelse}(S_2), G_P^Q] \mapsto \{[S_2, G_P^Q]\}}$$

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

We chose to define $(S_1)\texttt{orelse}(S_2)$ as a primitive operator instead of encoding it as $\texttt{if}(S_1)\texttt{then}(S_1)\texttt{else}(S_2)$ 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:

$$\frac{}{[\texttt{while}(S_1)\texttt{do}(S_2), G_P^Q] \mapsto \{[\texttt{if}(S_1)\texttt{then}(S_2; \texttt{while}(S_1)\texttt{do}(S_2))\texttt{else}(\mathsf{id}), G_P^Q]\}}$$

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

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

$$\frac{}{[\texttt{while}(S_1)\texttt{do}(S_2)(n), G_P^Q] \mapsto \{[\texttt{if}(S_1)\texttt{then}(S_2; \texttt{while}(S_1)\texttt{do}(S_2)(m))\texttt{else}(\mathsf{id}), G_P^Q]\}}$$

where $m = n - 1$.

- **Repeat.** The construction $\texttt{repeat}(S)$ 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.

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

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

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

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

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

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

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

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

5.2 Probabilistic extension

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

We denote by $\mapsto_\pi$ a transition step with probability π. The relation $\mapsto$ defined in the previous section can be seen as a particular case where $\pi = 1$, that is, $\mapsto$ corresponds to $\mapsto_1$.

The relation $\longmapsto$ also becomes probabilistic:

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

if $O_k \mapsto_\pi \{O'_{k_1}, \ldots, O'_{k_m}\}$.

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

Equiprobabilistic Choice of Reduct. The non-deterministic $\mathtt{one}(T)$ 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 $LS_{L_W \Rightarrow R_M^N}(G_P^Q)$. Since all of them have the same probability of being selected, in the axiom below $\pi = 1/|LS_{L_W \Rightarrow R_M^N}(G_P^Q)|$.

$$\frac{}{[\mathtt{one}(L_W \Rightarrow R_M^N), G_P^Q] \mapsto_\pi \{[\mathsf{id}, G'^{Q'}_{P'}]\}}\ G'^{Q'}_{P'} \in LS_{L_W \Rightarrow R_M^N}(G_P^Q)$$

$$\frac{}{[\mathtt{one}(L_W \Rightarrow R_M^N), G_P^Q] \mapsto_1 \{[\mathsf{fail}, G_P^Q]\}}\ LS_{L_W \Rightarrow R_M^N}(G_P^Q) = \emptyset$$

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

$$\pi = \frac{1}{|LPS_{{L_1}_{W_1} \Rightarrow {R_1}_{M_1}^{N_1} || \ldots || {L_k}_{W_k} \Rightarrow {R_k}_{M_k}^{N_k}}(G_P^Q)|}$$

$$\frac{}{[\mathtt{one}({L_1}_{W_1} \Rightarrow {R_1}_{M_1}^{N_1} || \ldots || {L_k}_{W_k} \Rightarrow {R_k}_{M_k}^{N_k}), G_P^Q] \mapsto_\pi \{[\mathsf{id}, G'^{Q'}_{P'}]\}}\ (\star)$$

($\star$) if $G'^{Q'}_{P'} \in LPS_{{L_1}_{W_1} \Rightarrow {R_1}_{M_1}^{N_1} || \ldots || {L_k}_{W_k} \Rightarrow {R_k}_{M_k}^{N_k}}(G_P^Q)$

$$\frac{}{[\mathtt{one}({L_1}_{W_1} \Rightarrow {R_1}_{M_1}^{N_1} || \ldots || {L_k}_{W_k} \Rightarrow {R_k}_{M_k}^{N_k}), G_P^Q] \mapsto_1 \{[\mathsf{fail}, G_P^Q]\}}\ (\star)$$

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

Probabilistic Choice of Rules. The probabilistic construct $\texttt{ppick}(T_1, \pi_1, \ldots, T_n, \pi_n)$ must be used with constructs `one()` or `all()` with the following rules:

$$\frac{}{[\texttt{one}(\texttt{ppick}(T_1, \pi_1, \ldots, T_n, \pi_n)), G_P^Q] \mapsto_{\pi_j} \{[\texttt{one}(T_j), G_P^Q]\}}$$

$$\frac{}{[\texttt{all}(\texttt{ppick}(T_1, \pi_1, \ldots, T_n, \pi_n)), G_P^Q] \mapsto_{\pi_j} \{[\texttt{all}(T_j), G_P^Q]\}}$$

Probabilistic position update and focusing. $\texttt{setPos}(D)$ 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:

$$\frac{}{[\texttt{setPos}(\texttt{one}(F)), G_P^Q] \mapsto_{\pi} \{[\mathsf{id}, G_{P'}^Q]\}} \; \texttt{one}(F(G_P^Q)) =_{\pi} P'$$

$$\frac{}{[\texttt{setBan}(\texttt{one}(F)), G_P^Q] \mapsto_{\pi} \{[\mathsf{id}, G_P^{Q'}]\}} \; \texttt{one}(F(G_P^Q)) =_{\pi} Q'$$

where $\texttt{one}(F(G_P^Q)) =_{1/|V_{G'}|} F_1$ if $F(G_P^Q) = G'$ and $F_1 \in V_{G'}$. In other words, if $F(G_P^Q) = G'$, $\texttt{one}(F(G_P^Q))$ 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 Positions.

$$\texttt{ppick}(F_1, \pi_1, \ldots, F_n, \pi_n)(G_P^Q) =_{\pi_j} F_j(G_P^Q)$$

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 transitions according to the strategy $S_{\mathcal{R}}$, starting from the *initial configuration* $\{\left[S_{\mathcal{R}}, G_P^Q\right]\}$.

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 $\mathit{Reach}(C) = \{G_1, \ldots, G_k, \ldots, G_n\}$ 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 derivation $T = C_0 \longmapsto \ldots \longmapsto C_n$, let $\mathit{Reach}(T) = \bigcup_{C_k, 0 \le k \le n} \mathit{Reach}(C_k)$ be the set of associated reachable graphs.

As presented in [50], 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 9).

In order to state these properties, we need to restrict the Porgy strategy language by excluding external functions (command $\texttt{update}(f\{\mathit{list}\})$) 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 5.3.1 (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 derivation $C_0 = \{[S, G]\} \longmapsto \ldots \longmapsto C_k = \{\ldots[S_k, G_k]\ldots\}$ and if $G' \in Reach(T)$, then $G \to^*_{\mathcal{R}} G'$.*

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 $[S_{n-1}, G_{n-1}] \mapsto \{\ldots, [S', G'], \ldots\}$. G_{n-1} has been introduced at some step $k < n$ and by induction $G \to^*_{\mathcal{R}} G_{n-1}$. Let us prove that $G_{n-1} \to^*_{\mathcal{R}} G'$ by case analysis on the different strategy constructs, applied in the $\mapsto$ 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 $G \to^*_{\mathcal{R}} G_{n-1}$; 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, $G_{n-1} \to^*_{\mathcal{R}} G'$. 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 5.3.2 (Completeness)**.** PORGY-*Light is complete w.r.t. (located) port graph rewriting. More precisely, if $G \to^*_{\mathcal{R}} G'$ ($G^Q_P \to^*_{\mathcal{R}} G'^{Q'}_{P'}$), there exists $S \in \mathcal{L}(\mathcal{R})$ and a derivation 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 $\mapsto$ transition can be performed. We prove in this section that all terminal configurations consist of results (see Definition 17 below). 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 17 (Result)**.** *Strategic graph programs are called* results *if they are of the form* $[\mathsf{id}, G^Q_P]$ *or* $[\mathsf{fail}, G^Q_P]$.

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

Property 5.4.1 (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 $[S, G_P^Q] \mapsto C$.*

Proof. By induction on S. According to Definition 15 (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 derivation out of the initial configuration $\{\left[S, G_P^Q\right]\}$: the *derivation result set* associated to a sequence of transitions is the set of results in the configurations in the sequence.

Definition 18 (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 $C_0 = \{[S_0, G_0]\} \longmapsto \ldots \longmapsto C_k = \{\ldots \left[S_k, G_{k P_k}^{Q_k}\right] \ldots\}$, let $Results(T) = \bigcup_{C_k, 0 \leq k \leq n} Results(C_k)$

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

If the sequence of transitions out of the the initial configuration $\{\left[S, G_P^Q\right]\}$ ends in a terminal configuration then the derivation 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 set is undefined ($\perp$).

The full strategy language of PORGY contains probabilistic operators `one`() and `ppick`(), for Rules and Positions constructs. 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 above, we have the property:

Property 5.4.2 (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 transition system 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 19. *A strategic graph program* $\left[S, G_P^Q\right]$ *is* strongly terminating *(or just terminating) if there is no infinite transition sequence out of the initial configuration* $\{\left[S, G_P^Q\right]\}$*, otherwise it is non-terminating. It is* weakly terminating *if we can reach 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. In the proof of termination, we need a lemma.

Lemma 20. *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 $\left[S_1, G_P^Q\right] \mapsto \{\left[S_1^1, G1_{P_1}^{Q_1}\right], \ldots, \left[S_1^k, Gk_{P_k}^{Q_k}\right]\}$ (rule premise), we can apply the induction hypothesis to deduce that each of

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

is strongly terminating. □

Property 5.5.1 (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. Looking at each transition step $[S, G] \longmapsto \{\ldots [S_k, G_k] \ldots\}$, it is easy to check that S_k is a sub-expression of S except for $S = S_1; S_2$, in which case we conclude using Lemma 20. □

Corollary 5.5.1. *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 5.6.1 (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 Tulip plugins. The latest version of Porgy (including Tulip) can be downloaded from `http://tulip.labri.fr/TulipDrupal/?q=porgy` either as source code or binaries for MacOS and Windows machines.

We use the Tulip library because it natively supports many features we need. First of all, 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. Tulip properties can be seen as records (see Def. 2). Other Tulip features we need are described below. We refer the reader to [55] for more details about the interactive features of Porgy and how they are implemented.

Graph Data Structure.

We made an abstraction layer on the Tulip graph data structure to handle port graphs. Thus, a port graph node is composed of a set of Tulip nodes (the ports) connected by edges to a centre node to form a star structure. To be able to use traditional graph layout algorithms (like FMˆ3 or GEM which are included in Tulip) without modifying them, a port graph layout is divided in two steps: first, all ports are removed (edges are kept), a standard layout plugin is applied, then ports are added and their placement is optimised to reduce edge length. This results in a readable and understandable drawing.

Rewriting Core Engine.

When applying a rule $L_W \Rightarrow R_N^M$ on a located graph G_P^Q, the first operation is to find a morphism from L to G, then given a morphism, the image of L is replaced in G by R. 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 have implemented Ullman's original algorithm [69] 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.

The second plugin is an implementation of Def. 9. 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 the nodes/edges modified by a rewriting step are created. The rest are left untouched. They are just marked as present in the newly created graph.

Strategy language.

The operational semantics defined in Section 5 is implemented in another 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 and makes the necessary computation for the Banned and Position subgraphs.

This plugin is developed with the recent C++11 along 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.

There are two ways to implement the construct `one`(T): either by taking the first computed redex, or by randomly choosing one among several ones. In both cases, it acts as a cut mechanism in logic programming. We chose to take the first computed redex as our implementation of the 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.

In the construct `update(`function_name{parameters_list}`)`, the `function_name` parameter must be the name of a Tulip plugin written in C++ or Python compatible with Porgy.

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 facilities 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 visualization 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 obviously helps to track 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 [55] for more details.

7 Related Work

Graph grammars were introduced in [53] to represent pictures and geometrical problems. Bunke [18] 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 [25, 20].

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 Push-Out [26] or Single Push-Out [41, 63] semantics) or algorithmic (that is, the application of rules is described as a sequence of atomic operations, as we have done in this paper). In the Double Push-Out approach, rules consist of a left-hand side, a right-hand side, and an "interface", which is a subgraph of both the left- and right-hand side, and remains untouched by the transformation. The Single Push-Out approach defines rules simply as a pair of graphs (without specifying a fixed subgraph). Although algorithmic, our rewriting semantics is in the spirit of the Single Push-Out approach. Moving to the Double-Push Out may be beneficial if graphs carry large quantities of data in their attributes, but it hasn't been necessary so far in PORGY.

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 [65] allows users to define the way rules are applied and includes non-deterministic constructs, sequence, conditional and loops. The Fujaba Tool Suite [52] offers a basic strategy language, including conditionals, sequence and method calls, but no concurrency. GROOVE [64] 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 [60] 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 development of specific redexes, etc. None of the languages above has Positions 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 [66], Mosbie [73] address the problem of modelling huge graphs. They integrate visualisation with modelling and simulation of rule-based intracellular biochemistry. The emphasis is put on visual global/local model exploration and integrated execution of simulations. Those systems are dedicated to biochemistry and do not provide a strategy language. However the rules are quite similar to ours and BioNetGen explicitely 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 [13] and Stratego [71]. 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 [45]).

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 [12]. Based on probabilistic rewrite theories, [44] provides a general semantic framework that unifies several existing models of probabilistic systems mentioned above.

- To perform stochastic simulation in biological signalling pathways [22], the κ language [23] 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 [19].

- Probabilistic functional programming languages such as Church [35, 34] and IBAL [54] 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 memoises its input function and is useful for describing persistent random properties.

In contrast to these approaches, we define a probabilistic choice operator which is associated to a subset of rules; the probabilities associated to each rule are assumed given and do not depend on the current state of the system. More precisely, although in PORGY it is not possible to associate a probability distribution with a strategy expression, the `ppick( )`construct allows us to provide probabilities for rule selection. Our approach is thus closer to the one followed in [16, 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. 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 improved along exploration of different application domains, mainly biological systems, interaction nets, graph theory, social networks. Its strategy language evolution reflects this progression. Previous versions of PORGY's strategy language were presented in [3, 51, 30, 29]. 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 in the Positions constructs to deal with rewriting positions: the Property and Ngb constructs defined in this paper are new. The small-step style of operational semantics is also new: the core language was defined using semantic rules in [51, 30] 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, 47, 56], 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 [62].

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 interesting improvements are yet possible both at theoretical and practical levels. Let us mention some of them that came out along PORGY's development.

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 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 possible idea is to use the double pushout method to perform less copying, another one is to explore 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 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. *Electr. Notes Theor. Comput. Sci.*, 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, 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, Daniel Archambault, Romain Bourqui, Maylis Delest, Jonathan Dubois, Bruno Pinaud, Antoine Lambert, Patrick Mary, Morgan Mathiaut, and Guy Melancon. Tulip III. In *Encyclopedia of Social Network Analysis and Mining.* Springer-Verlag New York, 2014.

[7] Daniel Balasubramanian, Anantha Narayanan, Christopher P. van Buskirk, and Gabor Karsai. The Graph Rewriting and Transformation Language: GReAT. *ECE-ASST*, 1, 2006.

[8] Emilie Balland, Paul Brauner, Radu Kopetz, Pierre-Etienne Moreau, and Antoine Reilles. Tom: Piggybacking Rewriting on Java. In Franz Baader, editor, *RTA*, volume 4533 of *LNCS*, 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 LNCS, pages 141–158, Eindhoven, The Netherlands, 1987. Springer-Verlag.

[11] Klaus Barthelmann. How to construct a hyperedge replacement system for a context-free set of hypergraphs. Technical report, Universität Mainz, Institut für Informatik, 1996.

[12] 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.

[13] Peter Borovanský, Claude Kirchner, Hélène Kirchner, Pierre-Etienne Moreau, and Christophe Ringeissen. An overview of ELAN. *ENTCS*, 15, 1998.

[14] 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 *EPTCS*, pages 1–19, 2009.

[15] 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.

[16] Olivier Bournez and Claude Kirchner. Probabilistic rewrite strategies. applications to ELAN. In Sophie Tison, editor, *Proceedings of the 13th RTA conference*, volume 2378 of *Lecture Notes in Computer Science*, pages 252–266, Copenhagen, July 2002. 13th International Conference, RTA 2002 Copenhagen, Denmark, July 22–24, 2002 Proceedings, Springer.

[17] 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*, 2008. Special issue on Experimental Systems and Tools.

[18] Horst Bunke. Attributed programmed graph grammars and their application to schematic diagram interpretation. *IEEE Trans. Pattern Anal. Mach. Intell.*, 4(6):574–582, 1982.

[19] 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.

[20] 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*, pages 163–246. World Scientific, 1997.

[21] 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.

[22] 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.

[23] Vincent Danos and Cosimo Laneve. Formal Molecular Biology. *Theoretical Computer Science*, 325(1):69–110, 2004.

[24] Edsger W. Dijkstra. *Selected writings on computing - a personal perspective.* Texts and monographs in computer science. Springer, 1982.

[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 *EPTCS*, 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 *LNCS*, 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] 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.

[45] 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.

[46] Yves Lafont. Interaction nets. In *Proceedings of the 17th ACM Symposium on Principles of Programming Languages (POPL'90)*, pages 95–108. ACM Press, 1990.

[47] Sylvain Lippi. in2: A graphical interpreter for interaction nets. In *RTA '02: Proceedings of the 13th International Conference on Rewriting Techniques and Applications*, pages 380–386, London, UK, 2002. Springer-Verlag.

[48] Salvador Lucas. Strategies in programming languages today. *Electr. Notes Theor. Comput. Sci.*, 124(2):113–118, 2005.

[49] Narciso Martí-Oliet, José Meseguer, and Alberto Verdejo. Towards a strategy language for Maude. *Electr. Notes Theor. Comput. Sci.*, 117:417–441, 2005.

[50] Narciso Martí-Oliet, José Meseguer, and Alberto Verdejo. A rewriting semantics for Maude strategies. *Electr. Notes Theor. Comput. Sci.*, 238(3):227–247, 2008.

[51] Olivier Namet. *Strategic Modelling with Graph Rewriting Tools.* PhD thesis, King's College London, 2011.

[52] Ulrich Nickel, Jörg Niere, and Albert Zündorf. The FUJABA environment. In *ICSE*, pages 742–745, 2000.

[53] 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.

[54] 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.

[55] Bruno Pinaud, Guy Melançon, and Jonathan Dubois. PORGY: A Visual Graph Rewriting Environment for Complex Systems. *Computer Graphics Forum*, 31(3):1265–1274, 2012.

[56] 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.

[57] M. J. Plasmeijer and M. C. J. D. van Eekelen. *Functional Programming and Parallel Graph Rewriting.* Addison-Wesley, 1993.

[58] Gordon D. Plotkin. A structural approach to operational semantics. *J. Log. Algebr. Program.*, 60-61:17–139, 2004.

[59] 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.

[60] Detlef Plump. The Graph Programming Language GP. In Symeon Bozapalidis and George Rahonis, editors, *CAI*, volume 5725 of *LNCS*, pages 99–122. Springer, 2009.

[61] 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 *EPTCS*, pages 1–16, 2011.

[62] 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.

[63] Jean-Claude Raoult. On graph rewritings. *Theor. Comput. Sci.*, 32:1–24, 1984.

[64] Arend Rensink. The GROOVE Simulator: A Tool for State Space Generation. In *AGTIVE*, volume 3062 of *LNCS*, pages 479–485. Springer, 2003.

[65] 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 Computing by Graph Transformations, Volume 2: Applications, Languages, and Tools*, pages 479–546. World Scientific, 1997.

[66] 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.

[67] Terese. *Term Rewriting Systems.* Cambridge University Press, 2003. M. Bezem, J. W. Klop and R. de Vrijer, eds.

[68] 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 *EPTCS*, pages 51–65, 2010.

[69] J.R. Ullman. An algorithm for subgraph isomorphism. *Journal of the ACM*, 23(1):31–42, 1976.

[70] 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 *EPTCS*, pages 65–79, 2015.

[71] Eelco Visser. Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In *Proc. of RTA'01*, volume 2051 of *Lecture Notes in Computer Science*, pages 357–361. Springer-Verlag, 2001.

[72] Eelco Visser. A survey of strategies in rule-based program transformation systems. *J. Symb. Comput.*, 40(1):831–873, 2005.

[73] 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.