Strategic Port Graph Rewriting: An Interactive Modelling and Analysis Framework*

Maribel Fernández
King's College London, Department of Informatics, Strand, London WC2R 2LS, UK
maribel.fernandez@kcl.ac.uk

Hélène Kirchner
Inria, Domaine de Voluceau, Rocquencourt BP 105, 78153 Le Chesnay Cedex, France
helene.kirchner@inria.fr

Bruno Pinaud
Bordeaux University, LaBRI CNRS UMR5800, 33405 Talence Cedex, France
bruno.pinaud@labri.fr

We present strategic port graph rewriting as a basis for the implementation of visual modelling and analysis tools. The goal is to facilitate the specification, analysis and simulation of complex systems, using port graphs. A system is represented by an initial graph and a collection of graph rewriting rules, together with a user-defined strategy to control the application of rules. The strategy language includes constructs to deal with graph traversal and management of rewriting positions in the graph. We give a small-step operational semantics for the language, and describe its implementation in the graph transformation and visualisation tool PORGY.
Keywords: port graph, graph rewriting, strategies, simulation, analysis, visual environment

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 has a graphical interface [24] and an executable specification language (see Fig. 1), where a system is modelled as a port graph together with port graph rewriting rules defining its dynamics (Sect. 2).

Reduction strategies define which (sub)expression(s) should be selected for evaluation and which rule(s) should be applied (see [20, 8] for general definitions). Strategies are present in programming languages such as Clean [25], Curry [18], and Haskell [19] and can be explicitly defined to rewrite terms in languages such as ELAN [7], Stratego [32], Maude [22] or Tom [4]. They are also present in graph transformation tools such as PROGRES [30], AGG [12], Fujaba [23], GROOVE [29], GrGen [15] and GP [28]. PORGY's strategy language draws inspiration from these previous works, but a distinctive feature is that it allows users to define strategies using not only operators to combine graph rewriting rules but also operators to define the location in the target graph where rules should, or should not, apply.

The main contribution of this paper is the definition of a strategic graph program (Sect. 3). It 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),

*Partially supported by the French National Research Agency project EVIDEN (ANR 2010 JCJC 0201 01).

Dragan Bošnački, Stefan Edelkamp, Alberto Lluch Lafuente
& Anton Wijs (Eds.): 3rd Workshop on
GRAPH Inspection and Traversal Engineering (GRAPHITE 2014)
EPTCS 159, 2014, pp. 15–29, doi:10.4204/EPTCS.159.3

© M. Fernández, H. Kirchner and B. Pinaud
This work is licensed under the
Creative Commons Attribution License.

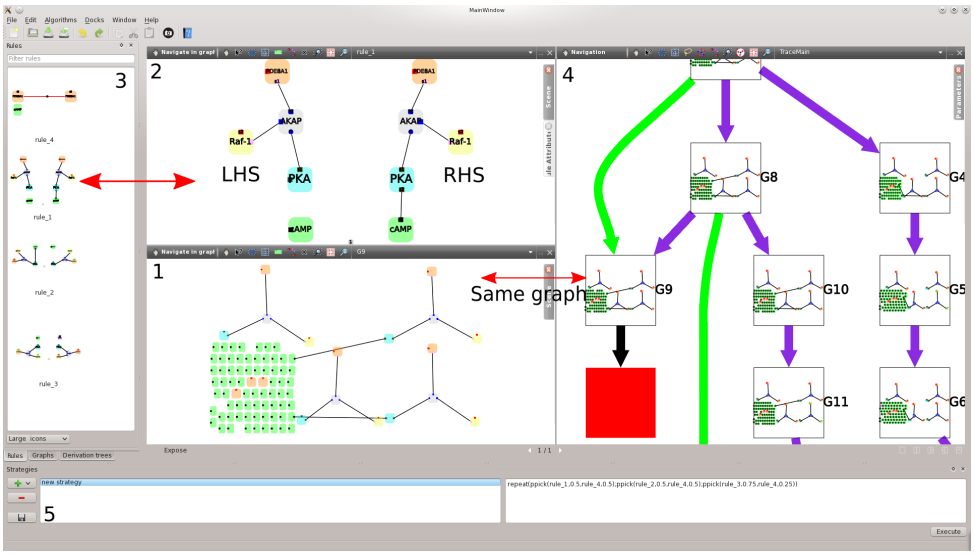

Figure 1: Overview of PORGY: (1) editing one state of the graph being rewritten; (2) editing a rule; (3) all available rewriting rules; (4) portion of the derivation tree, a complete trace of the computing history; (5) the strategy editor.

and a set of *rewrite rules* describing its dynamic behaviour, controlled by a *strategy*. We formalise the concept of strategic graph program, showing how located graphs generalise the notion of a term with a rewrite position, and provide a small-step operational semantics (Sect. 4) that specifies, for each strategic graph program, a set of rewrite derivations (i.e., a derivation tree) generated by applying the rewrite rules to the initial located graph according to the given strategy.

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 rewriting rules and the position where the rules apply. A rewriting position is a subgraph, which can be interactively selected (in a visual way), or can be specified using a *focusing* expression. Alternatively, rewrite positions could be encoded in the rewrite rules using markers or conditions [28]. We prefer to deal with positions directly, following Dijkstra's separation of concerns principle [11].

PORGY and its strategy language were first presented in [1, 14]. Unlike those papers, the notion of port graph considered in this paper includes attributes for nodes, ports and also edges, which are taken into account in the definition of port graph morphism. In addition, the strategy language includes a sublanguage to deal with properties of graphs, 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 using a transition system that specifies how the derivation tree is computed for each strategic graph program. In this transition system, configurations represent the part of the derivation tree that has already been computed and transitions specify the small-step execution of the commands. Since the language includes non-deterministic and probabilistic constructs, the full transition system is probabilistic. We give the transition rules for the deterministic sublanguage and briefly comment on the probabilistic ones.

2 Port Graph Rewriting

Several definitions of graph rewriting are available, using different kinds of graphs and rewriting rules (see, for instance, [10, 16, 5, 27, 6, 21]). In this paper we consider *port graphs* with *attributes* associated to nodes, ports and edges, generalising the notion of port graph introduced in [2].

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 each one by a name and attributes. For instance, a node or a port may have an attribute "state" (e.g., with possible values active/inactive or principal/auxiliary) or attributes defining some properties such as colour, shape, type, etc. Attributes may be used to define the behaviour of the modelled system and for visualisation purposes (as illustrated in Section 5).

Port Graph with Attributes. A *labelled port graph with attributes* is a tuple $G = (V_G, lv_G, E_G, le_G)$ where:

- V_G is a finite set of nodes.
- lv_G is a function that returns, for each $v \in V_G$ with n ports, a node label N (the node's name), a set $\{p_1, \ldots, p_n\}$ of port labels (each with its own set of attribute labels and values), and a set of attribute labels (each with a value). The node label determines the set of ports and attributes. Thus, we may write $\mathit{Interface}(v) = \mathit{Interface}(N) = \{p_1, \ldots, p_n\}$.
- E_G is a finite set of edges; each edge has two attachment ports $(v_1, p_1), (v_2, p_2)$, where $v_i \in V_G, p_i \in \mathit{Interface}(v_i)$. Edges are undirected, so $\langle (v_1, p_1), (v_2, p_2) \rangle$ is an unordered pair, and two nodes may be connected by more than one edge on the same ports.
- le_G is a labelling function for edges, which returns for each $e \in E_G$ an edge label, its attachment ports $(v_1, p_1), (v_2, p_2)$ and its set of attribute labels, each with an associated value.

Variables may be used as labels for nodes, ports, attributes and values in rewrite rules.
Rewriting is defined using a notion of graph morphism:

Port Graph Morphism. Let G and H be two port graphs, where G may contain variables but H does not. A *port graph morphism* $f : G \to H$ maps nodes, ports, edges with their respective attributes and values from G to H, such that all non-variable labels are preserved, the attachment of edges is preserved and the set of pairs of attributes and values for nodes, ports and edges are also preserved. If G contains variable labels, the morphism must instantiate the variables. Intuitively, the morphism identifies a subgraph of H that is equal to G except for variable occurrences. For more details we refer the reader to [13].

Port Graph Rewrite Rule. Port graphs are transformed by applying *port graph rewrite rules*. Formally, a port graph rewrite rule is a port graph consisting of two port graphs L and R, called the *left-hand side* and *right-hand side*, respectively; an *arrow* node labelled by $\Rightarrow_n$, where n is the number of ports, and each port in the arrow node has an attribute *type* whose value can be *bridge*, *blackhole* or *wire*; and a set of edges that each connect a port of the arrow node to ports in L or R. This set of edges must satisfy 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 incident edges from L and no edges connecting it to R.

The arrow node and arrow-edges are omitted if they are obvious from L and R.

The left-hand side of the rule, also called pattern, is used to identify subgraphs in a given graph, which are then replaced by the right-hand side of the rule. The arrow node describes the way the new subgraph should be linked to the remaining part of the graph, to avoid dangling edges [16, 10] during rewriting.

Derivation. A port graph G *rewrites* to G' using the rule $r = L \Rightarrow R$ and a morphism g from L to G, written $G \to_r^g G'$, if G' is obtained from G by replacing $g(L)$ by $g(R)$ in G and connecting $g(R)$ to the rest of G as specified by r's arrow node. We write $G \to_{\mathscr{R}} G'$ if $G \to_r^g G'$ using $r \in \mathscr{R}$. This induces a reflexive and transitive relation on port graphs, called *the rewriting relation*, denoted by $\to_{\mathscr{R}}^*$. Each *rule application* is a rewriting step and a *derivation*, or computation, is a sequence of rewriting steps.

Derivation Tree. Given a port graph G and a set of port graph rewrite rules $\mathscr{R}$, the *derivation tree* of G, written $DT(G,\mathscr{R})$, is a labelled tree such that the root is labelled by the initial port graph G, and its children are the roots of the derivation trees $DT(G_i,\mathscr{R})$ such that $G \to_{\mathscr{R}} G_i$. The edges of the derivation tree are labelled with the rewrite rule and the morphism used in the corresponding rewrite step. We will use *strategies* to specify the rewrite derivations of interest.

3 Strategic graph programs

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 at p, *i.e.*, downwards from the node P).

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.

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 $G_P^Q \to_{L_W \Rightarrow R_M^N}^g G'^{Q'}_{P'}$ 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 $G \to_{L \Rightarrow R} G'$ 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' in G' 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. $N = \emptyset$).

Let L, R be port graphs; M, N positions; $n \in \mathbb{N}$; $\pi_{i=1\ldots n} \in [0,1]$; $\sum_{i=1}^{n} \pi_i = 1$

(Strategies)	S	::=	$A \mid U \mid S;S \mid \texttt{repeat}(S) \mid \texttt{while}(S)\texttt{do}(S)$
		$\mid$	$(S)\texttt{orelse}(S) \mid \texttt{if}(S)\texttt{then}(S)\texttt{else}(S)$
		$\mid$	$\texttt{ppick}(S_1, \pi_1, \ldots, S_n, \pi_n)$
(Applications)	A	::=	$\mathsf{Id} \mid \mathsf{Fail} \mid \texttt{all}(T) \mid \texttt{one}(T)$
(Transformations)	T	::=	$L_W \Rightarrow R_M^N$
(Position Update)	U	::=	$\texttt{setPos}(F) \mid \texttt{setBan}(F) \mid \texttt{isEmpty}(F)$
(Focusing)	F	::=	$\texttt{CrtGraph} \mid \texttt{CrtPos} \mid \texttt{CrtBan} \mid \texttt{AllNgb}(F)$
		$\mid$	$\texttt{OneNgb}(F) \mid \texttt{NextNgb}(F) \mid \texttt{Property}(\rho, F)$
		$\mid$	$F \cup F \mid F \cap F \mid F \setminus F \mid \emptyset$

Table 1: Syntax of the strategy language.

Let *attribute* be an attribute label; a a valid value for the given attribute label; *function-name* the name of a built-in or user-defined function.

(Properties)	ρ	:=	$(Elem, Expr) \mid (\texttt{Function}, \textit{function-name})$
	$Elem$	:=	$\texttt{Node} \mid \texttt{Edge} \mid \texttt{Port}$
	$Expr$	:=	$\texttt{Label} == a \mid \texttt{Label} \mathrel{!}= a \mid \textit{attribute Relop attribute}$
			$\mid \textit{attribute Relop } a$
	$Relop$	:=	$== \mid \mathrel{!}= \mid > \mid < \mid >= \mid <=$

Table 2: Syntax of the Property Language.

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 $g(L) \cap Q = \emptyset$, 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*.

To control the application of rewriting rules, we introduce a strategy language whose syntax is 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 with *focusing expressions* generated by F. The application constructs and some of the strategy constructs are strongly inspired by term rewriting languages such as ELAN [7], Stratego [32] and Tom [4]. 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 syntax presented here extends the one in [14] by including a language to define subgraphs of a given graph by selecting nodes that satisfy some simple properties (see Table 2).

The focusing constructs are a distinctive feature of our language. They 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). We describe them briefly below.

- `CrtGraph`, `CrtPos` and `CrtBan`, applied to a located graph G_P^Q, return respectively G, P and Q.
- `AllNgb`, `OneNgb` and `NextNgb` denote functions that apply to pairs consisting of a located graph G_P^Q and a subgraph G' of G. If *Pos* is an expression denoting a subgraph G' of the current graph G, then `AllNgb`(*Pos*) is the subgraph of G consisting of all immediate successors of the nodes in G', where an immediate successor of a node v is a node that has a port connected to a port of v. `OneNgb`(*Pos*) returns a subgraph of G consisting of one randomly chosen node which is an immediate successor of a node in G'. `NextNgb`(*Pos*) computes all successors of nodes in G' using for each node only the port labelled "next" (so `NextNgb`(*Pos*) returns a subset of the nodes returned by `AllNgb`(*Pos*)).
- `Property`(ρ, F) is used to select a subgraph of a given graph, satisfying a certain property, specified by ρ. It can be seen as a filtering construct: if the focusing expression F generates a subgraph G' then `Property`(ρ, F) returns a subgraph containing only the nodes and edges from G' that satisfy the decidable property ρ. It typically tests a property on nodes, ports, or edges, allowing us for instance to select the subgraph of nodes with active ports: `Property`$((\texttt{Port}, \mathit{State} == \mathit{active}), F)$. It is also possible to specify a function to be used to compute the subgraph: `Property`$((\texttt{Function}, \mathit{Root}), \texttt{CrtGraph})$ uses the built-in (or user-defined) function *Root* to compute a specific subgraph from the current graph.
- $\cup$, $\cap$ and $\setminus$ are union, intersection and complement of port graphs which may be used to combine multiple Property operators; $\emptyset$ denotes the empty graph.

Other operators can be derived from the language constructs. A useful example is the `not` construct:

- `not`$(S) \triangleq$ `if`(S)`then`(Fail)`else`(Id). It fails if S succeeds and succeeds if S fails.

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

4 Semantics of strategic graph programs

Intuitively, a strategic program consists of an initial port graph, together with a set of rules that will be 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 (that is, a set of configurations with a binary relation on configurations), defining a *small step* operational semantics in the style of [26].

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

Given a strategic graph program $\left[S_{\mathscr{R}}, G_P^Q\right]$, we will define sequences of transitions according to the strategy S, starting from the *initial configuration* $\{\left[S, G_P^Q\right]\}$. A configuration is *terminal* if no transitions can be performed.

We will prove that all terminal configurations in our transition system consist of *results*, denoted by V, of the form $[\mathsf{Id}, G_P^Q]$ or $[\mathsf{Fail}, G_P^Q]$. 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.

Below we provide the transition rules for the core sublanguage, that is, the sublanguage that does not include the non-deterministic operators `one()`, () `orelse` (), `ppick()`, `repeat()` and `OneNgb()`.

Transitions The transition relation $\longrightarrow$ is a binary relation on configurations, defined as follows:

$$\{O_1, \ldots, O_k, V_1, \ldots, V_j\} \longrightarrow \{O'_{11}, \ldots, O'_{1m_1}, \ldots, O'_{km_k}, V_1, \ldots, V_j\}$$

if $O_i \to \{O'_{i1}, \ldots, O'_{im_i}\}$, for $1 \leq i \leq k$, where $k \geq 1$ and where some of the O'_{ij} might be results.

The auxiliary relation $\to$ is defined below using axioms and rules.

A configuration $\{O_1, \ldots, O_k, V_1, \ldots, V_j\}$ is a multiset of graph programs, representing a partially computed derivation tree. Each element in the configuration represents a node in the derivation tree associated to the initial graph program. Some of the elements may already be results. The transition relation performs reductions in parallel at all the positions in the derivation tree where there is a reducible graph program O_i.

Definition The transition relation $\to$ on individual strategic graph programs is defined by induction.

There are no axioms/rules defining transitions for a program where the strategy is Id or Fail (these are terminal).

Axioms for the operator `all`*:*

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

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

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

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

Position Update and Focusing. Next we give the semantics of the commands that are used to specify and update positions *via* focusing constructs. The focusing expressions generated by the grammar for the non terminal F in Tab. 1 have a functional semantics (see below). In other words, an expression F denotes a function that applies to the current located graph, and computes a subgraph of G. Since there is no ambiguity, the function denoted by the expression F is also called F.

$$\frac{}{[\mathtt{setPos}(F), G_P^Q] \to \{[\mathsf{Id}, G_{P'}^Q]\}} \; F(G_P^Q) = P' \qquad \frac{}{[\mathtt{setBan}(F), G_P^Q] \to \{[\mathsf{Id}, G_P^{Q'}]\}} \; F(G_P^Q) = Q'$$

$$\frac{}{[\mathtt{isEmpty}(F), G_P^Q] \to \{[\mathsf{Id}, G_P^Q]\}} \; F(G_P^Q) = \emptyset$$

$$\frac{}{[\mathtt{isEmpty}(F), G_P^Q] \to \{[\mathsf{Fail}, G_P^Q]\}} \; F(G_P^Q) \neq \emptyset$$

$$\begin{array}{lcll}
\texttt{CrtGraph}(G_P^Q) & = & G & \texttt{CrtPos}(G_P^Q) = P \qquad \texttt{CrtBan}(G_P^Q) = Q \\
\texttt{AllNgb}(F)(G_P^Q) & = & G' & \text{where } G' \text{ consists of all immediate successors of} \\
 & & & \text{nodes in } F(G_P^Q) \\
\texttt{NextNgb}(F)(G_P^Q) & = & G' & \text{where } G' \text{ consists of the immediate successors,} \\
 & & & \textit{via} \text{ ports labelled "next", of nodes in } F(G_P^Q) \\
\texttt{Property}(\rho,F)(G_P^Q) & = & G' & \text{where } G' \text{ consists of all nodes in } F(G_P^Q) \text{ satisfying } \rho \\
(F_1\ op\ F_2)(G_P^Q) & = & F_1(G_P^Q)\ op\ F_2(G_P^Q) & \text{where } op \text{ is } \cup, \cap, \setminus
\end{array}$$

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.

Sequence. The semantics of sequential application, written $S_1;S_2$, is defined by two axioms and a rule:

$$\frac{}{[\mathsf{Id};S,G_P^Q] \to \{[S,G_P^Q]\}} \qquad \frac{}{[\mathsf{Fail};S,G_P^Q] \to \{[\mathsf{Fail},G_P^Q]\}}$$

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

The rule for sequences ensures that S_1 is applied first.

Conditional. The behaviour of the strategy $\texttt{if}(S_1)\texttt{then}(S_2)\texttt{else}(S_3)$ depends on the result of the strategy S_1. If S_1 succeeds on (a copy of) the current located graph, then S_2 is applied to the current graph, otherwise S_3 is applied.

$$\frac{\{[S_1,G_P^Q]\} \longrightarrow^* M \ s.t.\ [\mathsf{Id},G'] \in M}{[\texttt{if}(S_1)\texttt{then}(S_2)\texttt{else}(S_3),G_P^Q] \to \{[S_2,G_P^Q]\}}$$

$$\frac{\{[S_1,G_P^Q]\} \longrightarrow^* \{[\mathsf{Fail},G_1],\ldots,[\mathsf{Fail},G_n]\}}{[\texttt{if}(S_1)\texttt{then}(S_2)\texttt{else}(S_3),G_P^Q] \to \{[S_3,G_P^Q]\}}$$

While loop. Iteration is defined using a conditional as follows:

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

Note that S_1 used as a condition in the two constructs above may produce some successes and some failure results. Also, in general the strategy S_1 could be non-deterministic and/or non-terminating. To avoid non-deterministic conditions in if and while commands, the class *Cond* of strategies generated by the following grammar should be used:

$$\mathit{Cond} ::= \mathit{Cond};\mathit{Cond} \mid \mathsf{Id} \mid \mathsf{Fail} \mid \texttt{all}(T) \mid \texttt{isEmpty}(F) \mid \texttt{not}(\mathit{Cond})$$

where F should also be deterministic:

$$F ::= \texttt{AllNgb}(F) \mid \texttt{NextNgb}(F) \mid \texttt{Property}(\rho,F) \mid \cup \mid \cap \mid \setminus \mid \emptyset$$

However, using non-deterministic constructs in the condition is not necessarily unsafe: if R is a located rule, we could, for instance, write $\texttt{if}(\texttt{one}(R))\texttt{then}(S_2)\texttt{else}(S_3)$ to perform either S_2 or S_3,

depending on whether R is applicable at the current position or not. Also note that although the strategy $\texttt{one}(R)$ is non-deterministic, the strategy $\texttt{not}(\texttt{one}(R))$ is deterministic (we are simply testing whether R can be applied or not).

We finish this section by giving the intuition for the semantics of the remaining constructs.

To define the semantics of the non-deterministic and probabilistic constructs in the language, we generalise the transition relation. Let us denote by $\rightarrow_\pi$ a transition step with probability π. The relation $\rightarrow$ defined above can be seen as a particular case where $\pi = 1$, that is, $\rightarrow$ corresponds to $\rightarrow_1$. The relation $\longrightarrow$ on configurations also becomes probabilistic:

$$\{O_1,\ldots,O_k,V_1,\ldots,V_j\} \longrightarrow_\pi \{O'_{11},\ldots,O'_{1m_1},\ldots,O'_{km_k},V_1,\ldots,V_j\}$$

if $O_i \rightarrow_{\pi_i} \{O'_{i1},\ldots,O'_{im_i}\}$, for $1 \leq i \leq k$ (where $k \geq 1$ and some of the O'_{ij} might be results) and $\pi = \pi_1 \times \cdots \times \pi_k$.

We write $M \longrightarrow^*_\pi M'$ if there is a sequence of transitions $\longrightarrow_{\pi_i}$ from configuration M to M', such that the product of probabilities is π.

We can define transition rules for the remaining constructs in the strategy language as follows.

Probabilistic Choice of Strategy:

$$\frac{}{[\texttt{ppick}(S_1,\pi_1,\ldots,S_n,\pi_n),G^Q_P] \rightarrow_{\pi_j} \{[S_j,G^Q_P]\}}$$

Non-deterministic Choice of Reduct: The non-deterministic $\texttt{one}()$ operator takes as argument a rule. It randomly selects only one amongst the set of legal reducts $LS_{L_W \Rightarrow R^N_M}(G^Q_P)$. Since all of them have the same probability of being selected, in the axiom below $\pi = 1/|LS_{L_W \Rightarrow R^N_M}(G^Q_P)|$.

$$\frac{G'^{Q'}_{P'} \in LS_{L_W \Rightarrow R^N_M}(G^Q_P)}{[\texttt{one}(L_W \Rightarrow R^N_M),G^Q_P] \rightarrow_\pi \{[\mathsf{Id},G'^{Q'}_{P'}]\}}$$

$$\frac{LS_{L_W \Rightarrow R^N_M}(G^Q_P) = \emptyset}{[\texttt{one}(L_W \Rightarrow R^N_M),G^Q_P] \rightarrow_1 \{[\mathsf{Fail},G^Q_P]\}}$$

We omit the rules for orelse and repeat, and for the commands $\texttt{setPos}(F)$, $\texttt{setBan}(F)$ and $\texttt{isEmpty}(F)$, which are non-deterministic if the expression F is non-deterministic. Note that in focusing constructs, non-determinism is introduced by the operator $\texttt{OneNgb}(F)$.

5 Examples

Using focusing (specifically the `Property` construct), we can create concise strategies that perform traversals[1]. In this way, we can for instance switch between *outermost and innermost term rewriting* (on trees). This is standard in term-based languages such as ELAN [7] or Stratego [32, 9]; here we can also define traversals in graphs that are not trees. More examples can be found in [1, 24, 14].

The following strategy allows us to check if a graph is connected using a standard connectivity test. Assuming that all nodes of the initial graph have the Boolean attribute *state* set to false, we just need one rewriting rule, which simply sets *state* to *true* on a node. We start with the strategy *pick-one-node* to randomly select a node v as a starting point. Then, the rule is applied to all neighbours of v. When the rule cannot be applied any longer, the position subgraph is set to all neighbours of the previously used

[1] Working examples can be downloaded from `http://tulip.labri.fr/TulipDrupal/?q=porgy`.

nodes which still have *state* set to $false$ (*visit-neighbours-at-any-distance*). The strategy continues until the position subgraph is empty. If the rule can still be applied somewhere in the graph, there is a failure (*check-all-nodes-visited*). Note the use of attributes and focusing constructs to traverse the graph. Below the strategy R is an abbreviation for $\texttt{one}(R)$.

$$
\begin{aligned}
\textit{pick-one-node:}\ & \texttt{setPos}(\texttt{CrtGraph});\\
& R;\\
& \texttt{setPos}(\texttt{Property}((Node, state == true), \texttt{CrtGraph}));\\
\textit{visit-neighbours-at-any-distance:}\ & \texttt{setPos}(\texttt{AllNgb}(\texttt{CrtPos}));\\
& \texttt{while}(\texttt{not}(\texttt{isEmpty}(\texttt{CrtPos})))\texttt{do}(\\
& \quad \texttt{if}(R)\texttt{then}(R)\texttt{else}(\\
& \quad \texttt{setPos}(\texttt{AllNgb}(\texttt{CrtPos})\setminus\\
& \texttt{Property}((Node, state == true), CrtGraph))));\\
\textit{check-all-nodes-visited:}\ & \texttt{setPos}(\texttt{CrtGraph});\\
& \texttt{not}(R)
\end{aligned}
$$

The next example uses node and edge attributes encoded inside two rules to build a spanning tree from a graph (see Fig. 2). The rules are: *start*, which is used to select the root of the tree, and $LC0$, which builds a branch of the tree. $LC0$ works as follows: given an existing node v of the tree, if v is linked to another node not part of the tree with an edge also not part of the tree, add both of them to the tree. The strategy used to build one spanning tree is very simple:

$$
\begin{aligned}
& \texttt{one}(start);\\
& \texttt{repeat}(\texttt{one}(LC0))
\end{aligned}
$$

If one wants all possible spanning trees, `one()` has simply to be replaced by `all()`. Figure 3 shows the results for three applications of the strategy.

6 Properties

In this section we discuss termination and completeness of the strategy language.

Termination. A strategic graph program $\left[S, G_P^Q\right]$ is *terminating* if there is no infinite transition sequence from the initial configuration $\{\left[S, G_P^Q\right]\}$. It is *weakly terminating* if a configuration having at least one result can be reached.

Result set. The *result set* associated to a sequence of transitions out of the configuration $\{\left[S, G_P^Q\right]\}$ is the set of all the results in the configurations in the sequence. Given a strategic graph program $\left[S, G_P^Q\right]$, if the sequence of transitions out of the initial configuration $\{\left[S, G_P^Q\right]\}$ ends in a terminal configuration then the result set of the sequence is a *complete result set* for the program $\left[S, G_P^Q\right]$. If a strategic graph program does not reach a terminal configuration (in case of non-termination) then the complete result set is undefined ($\perp$).

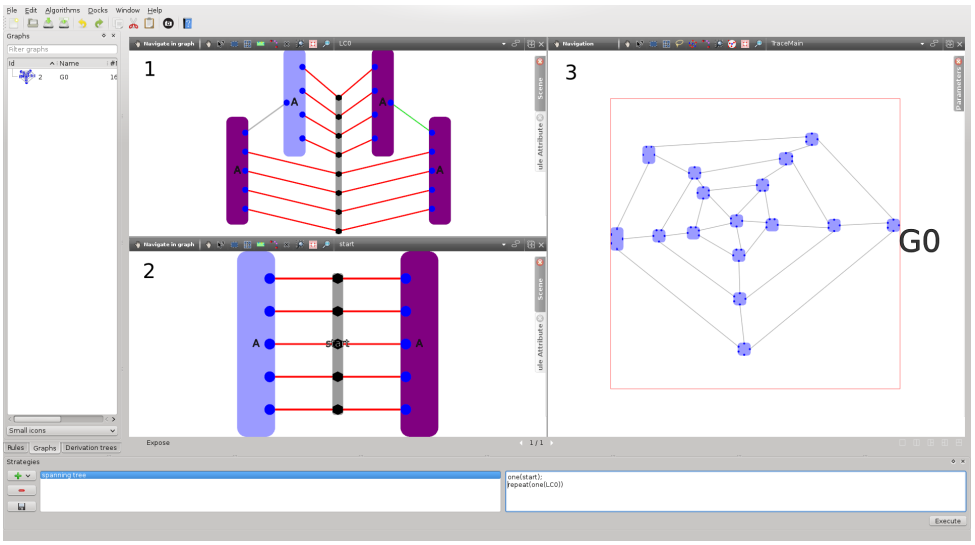

Figure 2: Computation of a spanning tree. Panel 1 shows the rule $LC0$. It is used to add nodes and edges to the spanning tree. Panel 2 shows the rule which sets the root of the tree. Panel 3 is the root of the derivation tree with the graph used for computation.

Note that there may exist more than one sequence of transitions out of the initial configuration $\{\left[S, G_P^Q\right]\}$ ending in a terminal configuration. However, for the core part of the language (that is, excluding the non-deterministic constructs `ppick()`, `()orelse()`, `repeat()`, `one()`, and `OneNgb()`), strategic graph programs have at most one terminal configuration (none if the program is non-terminating). As a consequence, each strategic graph program in the core language has at most one complete result set (Prop. 6.4).

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). We can also characterise the terminal configurations. The next lemma is useful for the termination proof:

Lemma 6.1 *If* $[S_1, G_P^Q]$ *is terminating and* S_2 *is such that* $[S_2, G'^Q_P]$ *is terminating for any* G'^Q_P*, then* $[S_1; S_2, G_P^Q]$ *is terminating.*

Property 6.2 (Termination) *The sublanguage that excludes the* `while()` *and* `repeat()` *constructs is terminating.*

Property 6.3 (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* $\{[S, G_P^Q]\} \to C$.

Proof By induction on S. According to the definition of transition in Sect. 4, 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 the cases in the grammar for S).

The language contains non-deterministic operators in each of its syntactic categories: `OneNgb()` for Position Update, `one()` for Applications and `ppick()`, `()orelse()` and `repeat()` for Strategies. For the sublanguage that excludes them, we have the property:

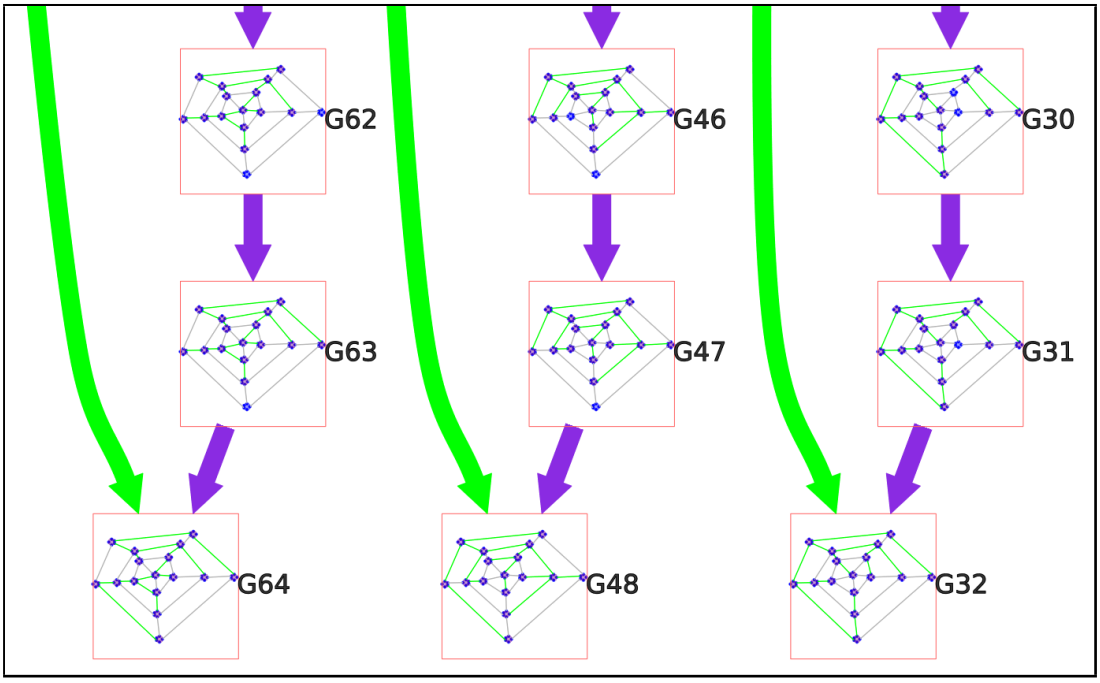

Figure 3: Three spanning trees given by applying the strategy three times from $G0$ (see Fig. 2).

Property 6.4 (Unique Complete Result Set) *Each strategic graph program in the sublanguage that excludes* `OneNgb()`, `one()`, `ppick()`, `()orelse()` *and* `repeat()` *has at most one* complete result set.

Proof If we exclude those constructs, the transition system is deterministic, so there is at most one sequence of transitions out of any initial configuration. Hence there is at most one terminal configuration and therefore at most one complete result set.

With respect to the computation power of the language, it is easy to state the Turing completeness property. The proof is similar to that in [17].

Property 6.5 (Turing Completeness) *The set of all strategic graph programs* $\left[S_{\mathscr{R}}, G_P^Q\right]$ *is Turing complete,* i.e. *can simulate any Turing machine.*

7 Implementation

PORGY is implemented on top of the visualisation framework Tulip [3] as a set of Tulip plugins. The strategy language is one of these plugins. A version of Tulip bundled with PORGY can be downloaded from `http://tulip.labri.fr/TulipDrupal/?q=porgy`.

Our first challenge was to implement port graphs, because Tulip only supports nodes and edges from a graph theory point of view. We had to develop an abstract layer on top of the Tulip graph library to be able to easily work with port graphs.

When applying a rule $L \Rightarrow R$ on a graph G, the first operation is to compute the morphism between the left-hand side L and G. This problem, known as the graph-subgraph isomorphism, still receives great attention from the community. We have implemented Ullman's original algorithm [31] because its implementation is straightforward and it is used as a reference in many papers.

The derivation tree is implemented with the help of metanodes (a node which represents a graph) and quotient graph functionalities of Tulip (a graph of metanodes). Each node of the derivation tree represents a graph G, except red nodes which represent failures (Fail). Inside each node, the user sees an interactive drawing of the graph (see panel 4 of Fig. 1). See [24] for more details about the interactive features of PORGY and how we implemented them.

The strategy plugin is developed with the Spirit C++ library from Boost[2]. This plugin works as a compiler: its inputs are a strategy defined as a text string and the Tulip graph datastructure, the output are low-level Tulip graph operations. Boost (precisely its Random library) is also used to generate the random numbers needed for the probabilistic operators. For instance, we use a non-uniform generator for `ppick()` to be able to choose a strategy following the given probabilities.

8 Conclusion

The strategy language defined in this paper is part of PORGY, an environment for visual modelling and analysis of complex systems through port graphs and port graph rewrite rules. It also offers a visual representation of rewriting traces as a derivation tree. The strategy language is used in particular to guide the construction of this derivation tree. The implementation uses the small-step operational semantics of the language. Some of these steps require a copy of the strategic graph program; this is done efficiently in PORGY thanks to the cloning functionalities of the underlying TULIP system [3]. Verification and debugging tools for avoiding conflicting rules or non-termination are planned for future work.

References

[1] Oana Andrei, Maribel Fernández, Hélène Kirchner, Guy Melançon, Olivier Namet & Bruno Pinaud (2011): *PORGY: Strategy-Driven Interactive Transformation of Graphs*. In Rachid Echahed, editor: 6^{th} *Int. Work. on Computing with Terms and Graphs*, 48, pp. 54–68, doi:10.4204/EPTCS.48.7.

[2] Oana Andrei & Hélène Kirchner (2009): *A Higher-Order Graph Calculus for Autonomic Computing*. In: *Graph Theory, Computational Intelligence and Thought. Golumbic Festschrift*, *Lecture Notes in Computer Science* 5420, Springer, pp. 15–26, doi:10.1007/978-3-642-02029-2_2.

[3] David Auber, Daniel Archambault, Romain Bourqui, Antoine Lambert, Morgan Mathiaut, Patrick Mary, Maylis Delest, Jonathan Dubois & Guy Mélançon (2012): *The Tulip 3 Framework: A Scalable Software Library for Information Visualization Applications Based on Relational Data*. Technical Report RR-7860, Inria.

[4] Emilie Balland, Paul Brauner, Radu Kopetz, Pierre-Etienne Moreau & Antoine Reilles (2007): *Tom: Piggybacking Rewriting on Java*. In Franz Baader, editor: *RTA*, *LNCS* 4533, Springer, pp. 36–47, doi:10.1007/978-3-540-73449-9_5.

[5] H.P. Barendregt, M.C.J.D. van Eekelen, J.R.W. Glauert, J. R. Kennaway, M.J. Plasmeijer & M.R. Sleep (1987): *Term graph rewriting*. In: *Proc. of PARLE, Parallel Architectures and Languages Europe*, *LNCS* 259-II, Springer-Verlag, pp. 141–158, doi:10.1007/3-540-17945-3_8.

[6] Klaus Barthelmann (1996): *How To Construct A Hyperedge Replacement System For A Context-Free Set Of Hypergraphs*. Technical Report, Universität Mainz, Institut für Informatik.

[7] Peter Borovanský, Claude Kirchner, Hélène Kirchner, Pierre-Etienne Moreau & Christophe Ringeissen (1998): *An overview of ELAN*. *EPTCS* 15, pp. 55–70, doi:10.1016/S1571-0661(05)82552-6.

[2] see `http://www.boost.org/libs/spirit` for more details

[8] Tony Bourdier, Horatiu Cirstea, Daniel J. Dougherty & Hélène Kirchner (2009): *Extensional and Intensional Strategies*. In: *Proc. 9th Int. Work. on Reduction Strategies in Rewriting and Programming*, pp. 1–19, doi:10.4204/EPTCS.15.1.

[9] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas & Eelco Visser (2008): *Stratego/XT 0.17. A Language and Toolset for Program Transformation*. *Science of Computer Programming, Special issue on Experimental Systems and Tools*, doi:10.1016/j.scico.2007.11.003.

[10] Andrea Corradini, Ugo Montanari, Francesca Rossi, Hartmut Ehrig, Reiko Heckel & Michael Löwe (1997): *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*, World Scientific, pp. 163–246, doi:10.1142/9789812384720_0003.

[11] Edsger W. Dijkstra (1982): *Selected writings on computing — a personal perspective*. Texts and monographs in computer science, Springer, doi:10.1007/978-1-4612-5695-3.

[12] Claudia Ermel, Michael Rudolf & Gabriele Taentzer (1997): *The AGG Approach: Language and Environment*. In Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski & Grzegorz Rozenberg, editors: *Handbook of Graph Grammars and Computing by Graph Transformations, Vol. 2: Applications, Languages, and Tools*, World Scientific, pp. 551–603.

[13] Maribel Fernandez, Hélène Kirchner, Ian Mackie & Bruno Pinaud (2014): *Visual Modelling of Complex Systems: Towards an Abstract Machine for PORGY*. In: *Lecture Notes in Computer Science*, 8493, Springer, p. To appear. See also http://cie2014.inf.elte.hu/.

[14] Maribel Fernández, Hélène Kirchner & Olivier Namet (2012): *A Strategy Language for Graph Rewriting*. In Germán Vidal, editor: *Logic-Based Program Synthesis and Transformation*, *LNCS* 7225, Springer, pp. 173–188, doi:10.1007/978-3-642-32211-2_12.

[15] Rubino Geiß, Gernot Veit Batz, Daniel Grund, Sebastian Hack & Adam Szalkowski (2006): *GrGen: A Fast SPO-Based Graph Rewriting Tool*. In: *Proc. of ICGT*, *LNCS* 4178, Springer, pp. 383–397, doi:10.1007/11841883_27.

[16] Annegret Habel, Jürgen Müller & Detlef Plump (2001): *Double-pushout graph transformation revisited*. *Mathematical Structures in Computer Science* 11(5), pp. 637–688, doi:10.1017/S0960129501003425.

[17] Annegret Habel & Detlef Plump (2001): *Computational Completeness of Programming Languages Based on Graph Transformation*. In: *Foundations of Software Science and Computation Structures, 4th Int. Conference, FOSSACS 2001, Proc.*, *LNCS* 2030, Springer, pp. 230–245. Available at `http://link.springer.de/link/service/series/0558/bibs/2030/20300230.htm`.

[18] M. Hanus, H. Kuchen & J.J. Moreno-Navarro (1995): *Curry: A Truly Functional Logic Language*. In: *Proc. ILPS'95 Work. on Visions for the Future of Logic Programming*, pp. 95–107.

[19] Simon L. Peyton Jones (2003): *Haskell 98 language and libraries: the revised report*. Cambridge Univ. Press.

[20] Claude Kirchner, Florent Kirchner & Hélène Kirchner (2008): *Strategic Computation and Deduction*. In Christoph Benzmüller, Chad E. Brown, Jörg Siekmann & Richard Statman, editors: *Reasoning in Simple Type Theory. Festchrift in Honour of Peter B. Andrews on His 70th Birthday*, *Studies in Logic and the Foundations of Mathematics* 17, College Publications, pp. 339–364. Available at `http://hal.inria.fr/inria-00433745`.

[21] Yves Lafont (1990): *Interaction Nets*. In: *Proc. of the 17th ACM Symposium on Principles of Programming Languages (POPL'90)*, ACM Press, pp. 95–108, doi:10.1145/96709.96718.

[22] Narciso Martí-Oliet, José Meseguer & Alberto Verdejo (2005): *Towards a Strategy Language for Maude*. *EPTCS* 117, pp. 417–441, doi:10.1016/j.entcs.2004.06.020.

[23] Ulrich Nickel, Jörg Niere & Albert Zündorf (2000): *The FUJABA environment*. In: *ICSE*, pp. 742–745, doi:10.1145/337180.337620.

[24] Bruno Pinaud, Guy Melançon & Jonathan Dubois (2012): *PORGY: A Visual Graph Rewriting Environment for Complex Systems*. *Computer Graphics Forum* 31(3), pp. 1265–1274, doi:10.1111/j.1467-8659.2012.03119.x.

[25] M. J. Plasmeijer & M. C. J. D. van Eekelen (1993): *Functional Programming and Parallel Graph Rewriting*. Addison-Wesley.

[26] Gordon D. Plotkin (2004): *A structural approach to operational semantics*. *J. Log. Algebr. Program.* 60-61, pp. 17–139, doi:10.1016/j.jlap.2004.03.009.

[27] Detlef Plump (1998): *Term Graph Rewriting*. In Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski & Grzegorz Rozenberg, editors: *Handbook of Graph Grammars and Computing by Graph Transformations, Vol. 2: Applications, Languages, and Tools*, World Scientific, pp. 3–61.

[28] Detlef Plump (2009): *The Graph Programming Language GP*. In Symeon Bozapalidis & George Rahonis, editors: *CAI*, *LNCS* 5725, Springer, pp. 99–122, doi:10.1007/978-3-642-03564-7_6.

[29] Arend Rensink (2003): *The GROOVE Simulator: A Tool for State Space Generation*. In: *AGTIVE*, *LNCS* 3062, Springer, pp. 479–485, doi:10.1007/978-3-540-25959-6_40.

[30] Andy Schürr, Andreas J. Winter & Albert Zündorf (1997): *The PROGRES Approach: Language and Environment*. In Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski & Grzegorz Rozenberg, editors: *Handbook of Graph Grammars and Computing by Graph Transformations, Vol. 2: Applications, Languages, and Tools*, World Scientific, pp. 479–546, doi:10.1142/9789812384720_0007.

[31] J.R. Ullman (1976): *An Algorithm for Subgraph Isomorphism*. *Journal of the ACM* 23(1), pp. 31–42, doi:10.1145/321921.321925.

[32] Eelco Visser (2001): *Stratego: A Language for Program Transformation Based on Rewriting Strategies System Description of Stratego 0.5*. In Aart Middeldorp, editor: *Rewriting Techniques and Applications*, *Lecture Notes in Computer Science* 2051, Springer Berlin Heidelberg, pp. 357–361, doi:10.1007/3-540-45127-7_27.