HAL open science

A Strategy Language for Graph Rewriting

Helene Kirchner, Olivier Namet, Maribel Fernandez

To cite this version:

Helene Kirchner, Olivier Namet, Maribel Fernandez. A Strategy Language for Graph Rewriting. LOPSTR 2011 - 21st International Symposium on Logic-Based Program Synthesis and Transformation, Jul 2011, Odense, Denmark. hal-00684244

HAL Id: hal-00684244

https://inria.hal.science/hal-00684244

Submitted on 31 Mar 2012

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

A Strategy Language for Graph Rewriting

Maribel Fernández[1], Hélène Kirchner[2], and Olivier Namet[1]

[1] King's College London, Department of Informatics, London WC2R 2LS, UK
{maribel.fernandez, olivier.namet}@kcl.ac.uk
[2] Inria, Domaine de Voluceau - Rocquencourt B.P. 105 - 78153 Le Chesnay France
helene.kirchner@inria.fr

Abstract. We give a formal semantics for a graph-based programming language, where a program consists of a collection of graph rewriting rules, a user-defined strategy to control the application of rules, and an initial graph to be rewritten. 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 language to deal with graph traversal and management of rewriting positions in the graph. This language is part of the graph transformation and visualisation environment PORGY.
Keywords: port graph, graph rewriting, strategies, visual environment

1 Introduction

To model complex systems, graphical formalisms are often preferred to textual ones, since they make it easier to visualise a system and convey intuitions about it. The dynamics of the system can then be specified using graph rewriting rules.

Graph rewriting has solid logic, algebraic and categorical foundations [15, 18], and graph transformations have many applications in specification, programming, and simulation tools [18]. In this paper, we focus on *port graph rewriting systems* [3], a general class of graph rewriting systems that have been successfully used to model biochemical systems and interaction net systems [27].

PORGY [2] is a visual environment that allows users to define port graphs and port graph rewriting rules, and to experiment with a rewriting system in an interactive way. To control the application of rewriting rules, PORGY uses a *strategy language*, which is the main subject of this paper.

Reduction strategies define which (sub)expression(s) should be selected for evaluation and which rule(s) should be applied (see [26, 12] 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.,[43, 41, 28]). Used for a long time in λ-calculus [8], strategies are present in programming languages such as Clean [33], Curry [24], and Haskell [25] and can be explicitely defined to rewrite terms in languages such as Elan [11], Stratego [42], Maude [29] or Tom [7]. They are also present in graph transformation tools such as PROGRES [39], AGG [19], Fujaba [32], GROOVE [37], GrGen [21] and GP [36]. 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 rewriting rules but also operators to define the location in the target graph where rules should, or should not, apply.

The PORGY environment is described in [2, 31], and we refer the reader to [20] for programming examples. In this paper, we focus on the design of the strategy language. Our main contribution is a formal semantics for the language, which allows users to analyse programs and reason about computations. We formalise the concept of graph program, and present semantic rules that associate to each graph program a result set (according to a given strategy), which is an abstraction of the derivation tree.

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. Subgraphs can be selected as focusing positions for rewriting interactively (in a visual way), or intensionally (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 focusing primitives. We prefer to separate the two notions of positions and rules to make programs more readable (the rewrite rules are not cluttered with encodings), and easier to maintain and adapt. In this sense, the language follows the separation of concerns principle [17]. For example, to change a traversal algorithm, it is sufficient to change the strategy and not the whole rewriting system.

The paper is organised as follows. In Section 2, we recall the concept of port graph. In Section 3, we present the syntax of the strategy language and formally define its semantics. Section 4 illustrates the language with examples, and Section 5 states some properties. Section 6 discusses related languages before concluding and giving directions for future work.

2 Background: Port Graph Rewriting

Several definitions of graph rewriting are available, using different kinds of graphs and rewriting rules (see, for instance, [14, 22, 9, 35, 10, 27]). In this paper we consider port graph rewriting systems [1, 3, 4], of which interaction nets [27] are a particular case.

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 and have attributes.

A port may be associated to a state (e.g., active/inactive or principal/auxiliary); this is formalised using a mapping from ports to port states. Similarly, nodes can also have associated properties such as colour, shape, root, leaf, etc. These attributes may be used for visualisation purposes and are also essential for the definition of strategy constructs, such as the `Property` and `NextSuc` operators introduced in Section 3; they are later illustrated in examples.

As shown in [1], a port graph can be considered as a labelled graph, and conversely, any labelled graph is a port graph in which each (port graph) node has a number of ports equal to the arity of its label. As a consequence, expressivity results, computational power, as well as correctness and completeness results on labelled graph rewriting can be translated to port graph rewriting. This is the approach taken here for defining rewriting on port graphs.

Let G and H be two port graphs. A *port graph morphism* $f : G \to H$ maps elements of G to elements of H preserving sources and targets of edges, constant node names and associated port name sets and states, up to variable renaming. We say that G and H are *isomorphic* if f is bijective.

A *port graph rewrite rule* $L \Rightarrow R$ is itself represented 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 arrow node describes the interface of the rule, avoiding dangling edges [22, 14] during rewriting as follows. For each port p in L, to which corresponds a non-empty set of ports $\{p_1, \ldots, p_n\}$ in R, the arrow node has a unique port r and the incident directed edges (p, r) and (r, p_i), for all $i = 1, \ldots, n$; all ports from L that are deleted in R are connected to the *black hole* port of the arrow node. We refer to [1] for full details. When the correspondence between ports in the left- and right-hand sides of the rule is obvious we omit the ports and edges involving the arrow node (as in Figure 2 in Section 4).

Port Graph Rewriting. Let $L \Rightarrow R$ be a port graph rewrite rule and G a port graph such that there is an injective port graph morphism g from L to G; hence $g(L)$ is a subgraph of G. A *rewriting step* on G using $L \Rightarrow R$, written $G \to_{L \Rightarrow R} G'$, transforms G into a new graph G' obtained from G by replacing the subgraph $g(L)$ of G by $g(R)$, and connecting $g(R)$ to the rest of the graph as specified in the arrow node of the rule. We call $g(L)$ a redex, and say that G rewrites to G' using $L \Rightarrow R$ at the position defined by $g(L)$, or that G' is a *result* of applying $L \Rightarrow R$ on G at $g(L)$. 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. If there is no such injective morphism, we say that G is *irreducible* by $L \Rightarrow R$. Given a finite set $\mathcal{R}$ of rules, a port graph G *rewrites* to G', denoted by $G \to_{\mathcal{R}} G'$, if there is a rule r in $\mathcal{R}$ such that $G \to_r G'$. This induces a transitive relation on port graphs, denoted by $\to^*_{\mathcal{R}}$. Each *rule application* is a rewriting step and a *derivation*, or computation, is a sequence of rewriting steps. A port graph on which no rule is applicable is in *normal form*. Rewriting is intrinsically non-deterministic since it may be possible to rewrite several subgraphs of a port graph with different rules or use the same one at different places, possibly getting different results.

3 Strategy Language

In this section we introduce the concept of graph program and give the syntax and semantics of the strategy language. In addition to the well-known constructs

to select rewrite rules, the strategy language provides focusing primitives to select or ban specific positions in the graph for rewriting. The latter is useful to program graph traversals for instance, and is a distinctive feature of the language.

3.1 Graph programs

Definition 1. *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.

When applying a port graph rewrite rule, not only the underlying graph G but also the position and banned subgraphs may change. A *located port graph 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, the set operators union, intersection and complement (denoted respectively $\cup, \cap, \setminus$) apply to port graphs considered as sets of nodes, ports and edges.

Definition 2. *A* located port graph rewrite rule *is given by a port graph rewrite rule $L \Rightarrow R$ and two disjoint subgraphs M and N of R. It is denoted $(L \Rightarrow R)_M^N$. We write $G_P^Q \rightarrow^g_{(L\Rightarrow R)_M^N} G'^{Q'}_{P'}$ and say that the located graph G_P^Q* rewrites to $G'^{Q'}_{P'}$ using $(L \Rightarrow R)_M^N$ at position P avoiding Q*, if $G \rightarrow_{L\Rightarrow R} G'$ with a morphism g such that $g(L) \cap P \neq \emptyset$ and $g(L) \cap Q = \emptyset$; the new position subgraph P' and banned subgraph Q' are defined as $P' = (P \setminus g(L)) \cup g(M)$, $Q' = Q \cup g(N)$.*

In general, for a given rule $(L \Rightarrow R)_M^N$ and located graph G_P^Q, more than one morphism g, such that $g(L) \cap P$ is not empty and $g(L) \cap Q$ is empty, might exist (i.e., several rewriting steps at P avoiding Q might be possible). Thus, the application of the rule at P avoiding Q produces a *set of located graphs*.

Definition 3. *A* graph program *consists of a set of located port graph rewrite rules $\mathcal{R}$, a strategy expression S (built from $\mathcal{R}$ using the grammar below) 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 obvious.*

The formal semantics of a graph program is given below, using an Abstract Reduction System [40, 26, 12]. The idea is to build a set of rewrite derivations out of $\left[S, G_P^Q\right]$ according to the strategy S (i.e., a *derivation tree*). A graph program $[S, G_P^Q]$ may define a non-terminating computation if there is an infinite derivation (written as an undefined result $[\bot, G_P^Q]$). All finite derivations produce results that are *values* of the form $[\mathsf{Id}, G_P^Q]$ or $[\mathsf{Fail}, G_P^Q]$ (see Property 2 in Section 5).

Let L, R be port graphs, M, N positions, ρ a property, m, n integers, $p_{i=1\ldots n} \in [0,1]$

$$
\begin{aligned}
\textbf{(Focusing)}\ F &:= \texttt{CrtGraph} \mid \texttt{CrtPos} \mid \texttt{CrtBan} \mid \texttt{AllSuc}(F) \mid \texttt{OneSuc}(F) \\
&\mid \texttt{NextSuc}(F) \mid \texttt{Property}(\rho, F) \mid F \cup F \mid F \cap F \mid F \setminus F \mid \emptyset \\
\textbf{(Transformations)}\ T &:= (L \Rightarrow R)^N_M \mid (T \parallel T) \\
\textbf{(Applications)}\ A &:= \mathsf{Id} \mid \mathsf{Fail} \mid T \mid \texttt{one}(T) \\
\textbf{(Strategies)}\ S &:= A \mid S; S \mid \texttt{ppick}(S_1, p_1, \ldots, S_n, p_n) \mid \texttt{while}(S)\texttt{do}(S) \\
&\mid (S)\texttt{orelse}(S) \mid \texttt{if}(S)\texttt{then}(S)\texttt{else}(S) \mid \texttt{isEmpty}(F) \\
&\mid \texttt{setPos}(F) \mid \texttt{setBan}(F)
\end{aligned}
$$

Fig. 1. Syntax of the strategy language.

Definition 4. *Given a graph program* $[S, G^Q_P]$ *and its derivation tree, the* result set *is the multiset of* values *(*$[\mathsf{Id}, {G'}^{Q'}_{P'}]$ *or* $[\mathsf{Fail}, {G'}^{Q'}_{P'}]$*) in the tree, together with* $[\bot, G^Q_P]$ *if there is an infinite branch. We say that a result set is a* success *if there exists at least one graph program* $[\mathsf{Id}, {G'}^{Q'}_{P'}]$ *(for some* ${G'}^{Q'}_{P'}$*) or is a* failure *otherwise.*

Definition 5. *A graph program* $[S, G^Q_P]$ *is strongly terminating if there are no infinite branches in its derivation tree (i.e., its result set contains only values* $[\mathsf{Id}, G^Q_P]$ *or* $[\mathsf{Fail}, G^Q_P]$*). It is weakly terminating if there is at least one finite branch (i.e., its result set contains at least one of* $[\mathsf{Id}, G^Q_P]$ *or* $[\mathsf{Fail}, G^Q_P]$*).*

3.2 Syntax and Informal Description

The syntax of the strategy language is given in Figure 1. The *strategy expressions* used in graph programs are generated by the grammar rules from the non-terminal S. A strategy expression combines applications, generated by A, and focusing operations, generated by F. The application constructs and some of the strategy constructs are strongly inspired from term rewriting languages such as Elan [11], Stratego [42] and Tom [7]. The syntax presented here is a revised and simplified version of the one used in [2, 20]; the main difference is that we now have an explicit notion of banned subgraph, a more concise syntax for iterative commands and a non-deterministic construct for rule applications.

Focusing. These constructs are used in strategy expressions to change the positions P and Q in the current located graph (e.g. to specify graph traversals).

- `CrtGraph` returns the whole current graph G. `CrtPos` and `CrtBan` return the current P and Q respectively in the located graph.
- `AllSuc`(F) returns the subgraph consisting of all immediate successors of the nodes in F, where an immediate successor of a node v is a node that has

a port connected to a port of v. $\texttt{OneSuc}(F)$ returns a subgraph consisting of one immediate successor of a node in F, chosen non-deterministically. $\texttt{NextSuc}(F)$ computes successors of nodes in F using for each node only the subset of its ports that have the attribute "next"; we call the ports in this distinguished subset the *next* ports (so $\texttt{NextSuc}(F)$ returns a subset of the nodes in $\texttt{AllSuc}(F)$).

- $\texttt{Property}(\rho, F)$ is a filtering construct, that returns a subgraph of G containing only the nodes from F that satisfy the decidable property ρ. It typically tests a property on nodes or ports, allowing us for instance to select the subgraph of red nodes or nodes with active ports (as mentioned in Section 2, ports and nodes in port graphs may have associated properties).
- $\cup$, $\cap$ and $\setminus$ are the standard set theory operators; $\emptyset$ denotes the empty set.

Transformations. The focusing 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. $(L \Rightarrow R)^N_M$ represents the application of the rule $L \Rightarrow R$ at the current position P and avoiding Q in G^Q_P, according to Definition 2. 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 *simultaneously.*

Applications. There are four kinds of applications according to the grammar: Id and Fail are two constant strategies that respectively denote success and failure. T denotes all possible applications of the transformation 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. `one(T)` non-deterministically computes only one of the possible applications of the transformation and ignore the others.

Strategies. The expression $S;S'$ represents sequential application of S followed by S'. When probabilities $p_1, \ldots, p_n \in [0, 1]$ are associated to strategies $S_1, \ldots, S_n$ such that $p_1 + \ldots + p_n = 1$, the strategy $\texttt{ppick}(S_1, p_1, \ldots, S_n, p_n)$ non-deterministically picks one of the strategies for application, according to the given probabilities. $\texttt{while}(S)\texttt{do}(S')$ keeps on sequentially applying S' while the expression S is successful; if S fails, then Id is returned. $(S)\texttt{orelse}(S')$ applies S if possible, otherwise applies S' and fails if both S and S' fail. $\texttt{if}(S)\texttt{then}(S')\texttt{else}(S'')$ checks if the application of S on (a copy of) G^Q_P returns Id, in which case S' is applied to (the original) G^Q_P, otherwise S'' is applied to the original G^Q_P. $\texttt{isEmpty}(F)$ behaves like Id if F returns an empty graph and Fail otherwise; this can be used for instance inside the condition of an `if` or `while`. $\texttt{setPos}(F)$ (resp. $\texttt{setBan}(F)$) sets the position subgraph P (resp. Q) to be the graph resulting from the focusing expression F.

3.3 Semantics

Focusing. The focusing operators defined by the grammar for F in Fig. 1 have a functional semantics. They apply to the current located graph, and compute

a subgraph (i.e., they return a subgraph of G). We define below the result of focusing operations on a given located graph.

$\texttt{CrtGraph}(G_P^Q)$	$= G$	$\texttt{CrtPos}(G_P^Q) = P$ $\quad$ $\texttt{CrtBan}(G_P^Q) = Q$
$\texttt{AllSuc}(F)(G_P^Q)$	$= G'$	where G' consists of all immediate successors of nodes in $F(G_P^Q)$
$\texttt{OneSuc}(F)(G_P^Q)$	$= G'$	where G' consists of one immediate successor of a node in $F(G_P^Q)$, chosen non-deterministically
$\texttt{NextSuc}(F)(G_P^Q)$	$= G'$	where G' consists of the immediate successors, via ports labelled "next", of nodes in $F(G_P^Q)$
$\texttt{Property}(\rho, F)(G_P^Q)$	$= G'$	where G' consists of all nodes in $F(G_P^Q)$ satisfying ρ
$(F_1\ op\ F_2)(G_P^Q)$	$= F_1(G_P^Q)\ op\ F_2(G_P^Q)$	where op is $\cup, \cap, \setminus$

Transformations, Applications and Strategy Operators. The constructs in the grammars for T, A and S in Fig. 1 are defined by semantic rules given below. Thus, the semantics of the strategy language is also defined by rewriting, as done for example in [30]. Our semantic rules are applied to configurations containing graph programs, defining a *small step* operational semantics in the style of [34].

In order to deal with the non-determinism introduced by rewriting (see Definition 2), we introduce the notion of configuration.

Definition 6. *A* configuration *is a multiset* $\{O_1, \ldots, O_n\}$ *where each* O_i *is either a graph program or an intermediate object (built with* auxiliary operators $\texttt{if}_2, \texttt{orelse}_2$ *and* $;_2$ *on graph programs, strategy expressions and located graphs), denoted by angular brackets (e.g.* $\langle [S_1, G_P^Q];_2 S_2, G_P^Q \rangle$*).*

In the semantic rules, we abuse notation and identify a singleton multiset with its element and work modulo the flattening of multisets, i.e., modulo associativity, commutativity and the axiom $\{\{X\}, Y\} = \{X, Y\}$; for instance $\{\{O_a, \ldots, O_b\}, O_y, \ldots, O_z\} = \{O_a, \ldots, O_b, O_y, \ldots, O_z\}$. We type variables in 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; A_1 is a variable of type application; S_3 represents a strategy expression). The auxiliary function $isSuccess([S, G_P^Q])$ returns $True$ or $False$ depending on whether the result set associated to $[S, G_P^Q]$ is a success or a failure. This function terminates if the graph program is strongly terminating (in implementations, backtracking or breadth-first search will be used to ensure that even if a strategy is weakly terminating, a partial result set is computed).

– Graph rewrite rules are themselves strategy operators:

$$[(L \Rightarrow R)_M^N, G_P^Q] \rightarrow \{[\mathsf{Id}, {G_1}_{P_1}^{Q_1}], \ldots, [\mathsf{Id}, {G_k}_{P_k}^{Q_k}]\}$$
$$\text{if } G_P^Q \rightarrow^{g_i}_{(L \Rightarrow R)_M^N} {G_i}_{P_i}^{Q_i} (\forall i, 1 \leq i \leq k) \text{ with } g_1 \ldots g_k \text{ pairwise different.}$$
$$[(L \Rightarrow R)_M^N, G_P^Q] \rightarrow [\mathsf{Fail}, G_P^Q] \qquad \text{if the rule is } \textit{not} \text{ applicable}$$

In the first rule, all possible applications of the rule are considered.

– Parallelism is allowed through the operator $||$ which works on rules only (not on general strategies). To define the semantics of $(L_1 \Rightarrow R_1)^{N_1}_{M_1} || \ldots ||(L_k \Rightarrow R_k)^{N_k}_{M_k}$, we define a new rule $((L_1 \cup \ldots \cup L_k) \Rightarrow_{1\ldots k} (R_1 \cup \ldots \cup R_k))^{N_1 \cup \ldots \cup N_k}_{M_1 \cup \ldots \cup M_k}$, where $\Rightarrow_{1\ldots k}$ contains all the ports and edges of $\Rightarrow_i$ (for $1 \leq i \leq k$). It implements simultaneous application of rules at disjoint redexes (note that two nodes may have the same label, but if they are different nodes, then the union will contain both nodes).

$$
\begin{array}{ll}
[(L_1 \Rightarrow R_1)^{N_1}_{M_1} || \ldots ||(L_k \Rightarrow R_k)^{N_k}_{M_k}, G^Q_P] \rightarrow & \\
\qquad [((L_1 \cup \ldots \cup L_k) \Rightarrow_{1\ldots k} (R_1 \cup \ldots \cup R_k))^{N_1 \cup \ldots \cup N_k}_{M_1 \cup \ldots \cup M_k}, G^Q_P] & \\
\qquad\qquad \text{if } \forall i,\ g_i(L_i) \cap P \neq \emptyset & \\
[(L_1 \Rightarrow R_1)^{N_1}_{M_1} || \ldots ||(L_k \Rightarrow R_k)^{N_k}_{M_k}, G^Q_P] \rightarrow [\mathsf{Fail}, G^Q_P] & \text{otherwise}
\end{array}
$$

– The non-deterministic `one()` 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 of the reducts, non-deterministically.

$$
\begin{array}{ll}
[\mathtt{one}((L_1 \Rightarrow R_1)^{N_1}_{M_1} || \ldots ||(L_k \Rightarrow R_k)^{N_k}_{M_k}), G^Q_P] \rightarrow & \\
\qquad [\mathtt{one}(((L_1 \cup \ldots \cup L_k) \Rightarrow_{1\ldots k} (R_1 \cup \ldots \cup R_k))^{N_1 \cup \ldots \cup N_k}_{M_1 \cup \ldots \cup M_k}), G^Q_P] & \\
\qquad\qquad \text{if } \forall i,\ g_i(L_i) \cap P \neq \emptyset & \\
[\mathtt{one}((L_1 \Rightarrow R_1)^{N_1}_{M_1} || \ldots ||(L_k \Rightarrow R_k)^{N_k}_{M_k}), G^Q_P] \rightarrow [\mathsf{Fail}, G^Q_P] & \text{otherwise} \\
[\mathtt{one}((L \Rightarrow R)^N_M), G^Q_P] \rightarrow [\mathsf{Id}, G'^{Q'}_{P'}] & \text{if } G^Q_P \rightarrow^g_{(L \Rightarrow R)^N_M} G'^{Q'}_{P'} \\
& \text{for a chosen } g \\
[\mathtt{one}((L \Rightarrow R)^N_M), G^Q_P] \rightarrow [\mathsf{Fail}, G^Q_P] & \text{if the rule is } \textit{not} \text{ applicable}
\end{array}
$$

– Position definition:

$$
\begin{array}{lll}
[\mathtt{setPos}(F), G^Q_P] & \rightarrow [\mathsf{Id}, G^Q_{P'}] & \text{where } P' \text{ is } F(G^Q_P) \\
[\mathtt{setBan}(F), G^Q_P] & \rightarrow [\mathsf{Id}, G^{Q'}_P] & \text{where } Q' \text{ is } F(G^Q_P) \\
 & & \\
[\mathtt{isEmpty}(F), G^Q_P] & \rightarrow [\mathsf{Id}, G^Q_P] & \text{if } F(G^Q_P) \text{ is empty} \\
[\mathtt{isEmpty}(F), G^Q_P] & \rightarrow [\mathsf{Fail}, G^Q_P] & \text{if } F(G^Q_P) \text{ is not empty}
\end{array}
$$

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

– Sequential application: below E denotes Id or Fail.

$$
\begin{array}{ll}
[S_1; S_2, G^Q_P] & \rightarrow \langle [S_1, G^Q_P];_2 S_2, G^Q_P \rangle \\
 & \quad \text{if } S_1 \neq \mathsf{Id}, S_1 \neq \mathsf{Fail} \\
\langle \{[E, {G_0}^{Q_0}_{P_0}], [S^1_1, {G_1}^{Q_1}_{P_1}], \ldots, [S^k_1, {G_k}^{Q_k}_{P_k}]\};_2 S_2, G^Q_P \rangle & \rightarrow \{[E; S_2, {G_0}^{Q_0}_{P_0}], \langle \{[S^1_1, {G_1}^{Q_1}_{P_1}], \\
 & \qquad \ldots, [S^k_1, {G_k}^{Q_k}_{P_k}]\};_2 S_2, G^Q_P \rangle\} \\
[\mathsf{Id}; S, G^Q_P] & \rightarrow [S, G^Q_P] \\
[\mathsf{Fail}; S, G^Q_P] & \rightarrow [\mathsf{Fail}, G^Q_P]
\end{array}
$$

The first rule for sequences ensures that S_1 is applied first to G_P^Q: it builds the configuration $[S_1, G_P^Q]$ — we say that S_1 is *promoted* so that it can be applied. The second rule can then be applied when a value is obtained from S_1 (note that k could be zero, in which case there is only $[E; S_2, {G_0}_{P_0}^{Q_0}]$ in the right hand side). Sequential application is strict: if the first strategy does not return a result, the final result is undefined.

– Conditional: the first rule promotes S_1, so that it is applied to G_P^Q and tested with the auxiliary function $isSuccess()$.

$$
\begin{array}{l}
[\texttt{if}(S_1)\texttt{then}(S_2)\texttt{else}(S_3), G_P^Q] \quad \rightarrow \\
\qquad\qquad\qquad\qquad \langle \texttt{if}_2(isSuccess([S_1, G_P]))\texttt{then}(S_2)\texttt{else}(S_3), G_P^Q\rangle \\
\langle \texttt{if}_2(True)\texttt{then}(S_2)\texttt{else}(S_3), G_P^Q\rangle \rightarrow [S_2, G_P^Q] \\
\langle \texttt{if}_2(False)\texttt{then}(S_2)\texttt{else}(S_3), G_P^Q\rangle \rightarrow [S_3, G_P^Q]
\end{array}
$$

– Iteration:

$$
\begin{array}{l}
[\texttt{while}(S_1)\texttt{do}(S_2), G_P^Q] \rightarrow \\
\qquad \langle \texttt{if}_2(isSuccess([S_1, G_P^Q]))\texttt{then}(S_2; \texttt{while}(S_1)\texttt{do}(S_2))\texttt{else}(\mathsf{Id}), G_P^Q\rangle
\end{array}
$$

– Priority choice:

$$
\begin{array}{l}
[(S_1)\texttt{orelse}(S_2), G_P^Q] \qquad\qquad\qquad\qquad \rightarrow \\
\qquad\qquad\qquad\qquad \langle([S_1, G_P^Q])\texttt{orelse}_2(S_2), G_P^Q\rangle \\
\langle(\{[\mathsf{Id}, {G_0}_{P_0}^{Q_0}], [S_1^1, {G_1}_{P_1}^{Q_1}], \ldots, [S_1^k, {G_k}_{P_k}^{Q_k}]\})\texttt{orelse}_2(S_2), G_P^Q\rangle \rightarrow \\
\qquad \{[\mathsf{Id}, {G_0}_{P_0}^{Q_0}], \langle(\{[S_1^1, {G_1}_{P_1}^{Q_1}] \ldots, [S_1^k, {G_k}_{P_k}^{Q_k}]\})\texttt{orelse}_2(S_2), G_P^Q\rangle\} \\
\langle(\{[\mathsf{Fail}, {G_1}_{P_1}^{Q_1}], \ldots, [\mathsf{Fail}, {G_k}_{P_k}^{Q_k}]\})\texttt{orelse}_2(S_2), G_P^Q\rangle \qquad \rightarrow [S_2, G_P^Q]
\end{array}
$$

Here, S_1 is promoted so that it can be applied to G_P^Q. If it fails (i.e., all the derivations end with Fail) then S_2 is applied to the initial graph. Note that again k could be zero in the second rule, in which case the right hand side is just $[\mathsf{Id}, {G_0}_{P_0}^{Q_0}]$. 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 non-deterministic operators: evaluating S_1 in the condition and in the "then" branch could yield different values.

– Probabilistic choice: we assume $prob(p_1, \ldots, p_n)$ returns the element $j \in \{1 \ldots n\}$ with probability p_j.

$$
[\texttt{ppick}(S_1, p_1, \ldots, S_n, p_n), G_P^Q] \rightarrow [S_j, G_P^Q] \quad \text{where } prob(p_1, \ldots, p_n) = j
$$

4 Examples

In this section we give examples to illustrate the expressivity of the language. The `not` and `try` operators, well-known in strategy languages for term rewriting, are not primitive in our language but can be derived, as well as $\texttt{repeat}(S)$, and bounded iteration; $|||$ is a weaker version of $\|$.

- $\texttt{not}(S) \triangleq \texttt{if}(S)\texttt{then}(\mathsf{Fail})\texttt{else}(\mathsf{Id})$ fails if S succeeds and succeeds if S fails.
- $\texttt{try}(S) \triangleq (S)\texttt{orelse}(\mathsf{Id})$ is a strategy that behaves like S if S succeeds, but if S fails then it behaves like Id.
- $\texttt{repeat}(S) \triangleq \texttt{while}(S)\texttt{do}(S)$ applies S as long as possible.
- $\texttt{while}(S_1)\texttt{do}(S_2)\texttt{max}(n) \triangleq$ $\texttt{if}(S_1)\texttt{then}(S_2;\texttt{if}(S_1)\texttt{then}(S_2;\dots)\texttt{else}(\mathsf{Id}))\texttt{else}(\mathsf{Id})$ representing a series of `if()`s of the same form, with exactly n occurrences of S_2.
- $\texttt{for}(n)\texttt{do}(S) \triangleq S;\dots;S$ where S is repeated n times.
- $A \,|||\, A'$ is similar to $A||A'$ except that it returns Id if at least one application of A or A' is possible (it can be generalised to n applications in parallel): $A_1|||A_2 \triangleq \texttt{if}(A_1)\texttt{then}(\texttt{if}(A_1||A_2)\texttt{then}(A_1||A_2)\texttt{else}(A_1))\texttt{else}(A_2)$

Using focusing (specifically the `Property` construct), we can create concise strategies that perform traversals. In this way, we can define outermost or innermost term rewriting (on trees) without needing to change the rewrite rules. This is standard in term-based languages such as ELAN [11] or Stratego [42][13]; here we can also define traversals in graphs that are not trees.

Outermost rewriting on trees: We define the abbreviation $\mathit{start} \triangleq \texttt{Property}(\mathit{root}, \texttt{CrtGraph})$, which selects the subgraph containing just the root of the tree. The *next* ports (see definition of the $\texttt{NextSuc}(F)$ operator in Section 3.3) for each node in the tree are defined to be the ones that connect with their children. The strategy for outermost rewriting with a rule R is:

$$\begin{array}{l}\texttt{setPos}(\mathit{start});\\ \texttt{while}(\texttt{not}(\texttt{isEmpty}(\texttt{CrtPos})))\texttt{do}\\ \quad(\texttt{if}(R)\texttt{then}(R;\texttt{setPos}(\mathit{start}))\texttt{else}(\texttt{setPos}(\texttt{NextSuc}(\texttt{CrtPos}))))\end{array}$$

Thus, if R can be applied then we apply it and set the position back to the root of the tree. Otherwise, $\texttt{setPos}(\texttt{NextSuc}(\texttt{CrtPos}))$ makes all children of all elements in the current position the new current position, thus descending one step into the tree.

Innermost rewriting on trees: We define the abbreviations $\mathit{start} \triangleq \texttt{Property}(\mathit{leaf}, \texttt{CrtGraph})$, which selects the leaves of the tree, and $\mathit{rest} \triangleq \texttt{CrtGraph} \setminus \mathit{start}$. For each node, the *next* port connects with the parent node.

$$\begin{array}{l}\texttt{setPos}(\mathit{start});\texttt{setBan}(\mathit{rest});\\ \texttt{while}(\texttt{not}(\texttt{isEmpty}(\texttt{CrtPos})))\texttt{do}(\\ \quad\texttt{if}(R)\texttt{then}(R;\texttt{setPos}(\mathit{start});\texttt{setBan}(\mathit{rest}))\\ \quad\texttt{else}(\texttt{setPos}(\texttt{NextSuc}(\texttt{CrtPos}));\texttt{setBan}(\texttt{CrtBan}\setminus\texttt{CrtPos}))\ )\end{array}$$

Thus, if R can be applied then we apply it and set the position back to the leaves of the tree and put all the other elements of the tree into the banned subgraph. Otherwise, we move up the tree one level with $\texttt{setPos}(\texttt{NextSuc}(\texttt{CrtPos}))$ and the banned subgraph is updated again to all the remaining elements of the tree (with $\texttt{setBan}(\texttt{CrtBan}\setminus\texttt{CrtPos})$).

Sorting: The following example shows how a non-ordered list of three colours (Blue, Red and White) can be sorted to represent the French flag (Blue first, then White and finally Red). We have three port nodes representing each colour (shown in Figure 2) that have two ports each: a *previous* port (to the left of the node) and a *next* port (to the right of the node). We also have a *Mast* port node

at the beginning of the list. Using the three rules in Figure 2, we can swap two colours if they are in the wrong order. Using the ||| operator we apply as many of these rules as we can in parallel. Our overall strategy would then be:

$$\texttt{repeat}(\texttt{setPos}(\texttt{CrtGraph}); ((\mathit{white1} ||| \mathit{red1}) ||| \mathit{red2}))$$

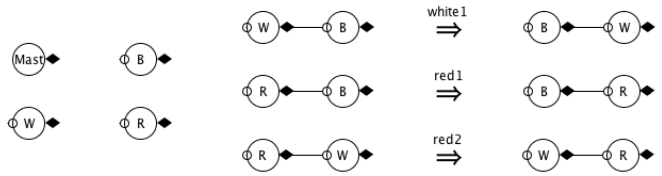

Fig. 2. The four port node types and the three flag sorting rules.

We can also program a sorting algorithm that starts from the mast node. If no rule can be applied, we move P one position across the list and try again. After a rule is applied we reset P to be just *Mast*. When we reach the end of the list, the program terminates and the list is correctly ordered. By defining: $\mathit{swap} \triangleq ((\mathit{white}1)\texttt{orelse}(\mathit{red}1))\texttt{orelse}(\mathit{red}2)$ and $\mathit{backToMast} \triangleq \texttt{Property}(\mathit{mast}, \texttt{CrtGraph})$, the strategy is:

$\texttt{setPos}(\mathit{backToMast});$
$\mathsf{while}(\mathsf{not}(\mathsf{isEmpty}(\mathsf{CrtPos})))\mathsf{do}$
$(\texttt{if}(\mathit{swap})\texttt{then}(\mathit{swap}; \texttt{setPos}(\mathit{backToMast}))\texttt{else}(\texttt{setPos}(\mathsf{NextSuc}(\texttt{CrtPos}))))$

This example illustrates the separation of concerns mentioned in the introduction: to program a flag-sorting algorithm that proceeds from the mast onwards, we do not need to change the rewrite rules (in contrast with other graph rewriting languages where focusing constructs are not available and conditional rewriting is used). More examples can be found in [31].

5 Properties

In this section we discuss termination and completeness of the semantic rules of the strategy language. We refer to [31] for the proofs omitted or just sketched here due to space constraints.

Graph programs are not terminating in general, however we can identify a terminating sublanguage (i.e. a sublanguage for which the semantic rules are terminating) and we can characterise the graph programs in normal form.

Property 1 (Termination). The sublanguage that excludes the while construct is strongly terminating.

Proof. To prove that a graph program in this sublanguage does not generate an infinite derivation with the semantic rules, we interpret configurations as natural numbers and show that this interpretation strictly decreases with each application of a rewrite rule within the semantics. The interpretation of a configuration

is defined as follows: $int(\{O_1, \ldots, O_n\}) = \sum_{i=1}^{n} \mathcal{I}(O_i)$ where $\mathcal{I}([S, G]) = size(S)$ and $\mathcal{I}(\langle S, G\rangle) = size(S)$. The size function is defined below (B denotes a Boolean and C a configuration):

$$
\begin{array}{lll}
size(\mathsf{Id}) = 0 & size(\mathsf{Fail}) = 0 & size((L \Rightarrow R)^N_M) = 1 \\
 & & size(\texttt{one}(T)) = 1 \\
size(\texttt{setPos}(F)) = 1 & size(\texttt{setBan}(F)) = 1 & size(\texttt{isEmpty}(F)) = 1 \\
size(S_1; S_2) = 2 + size(S_1) + size(S_2) & & \\
size(C;_2 S) = 1 + int(C) + size(S) & & \\
size(\texttt{if}(S_1)\texttt{then}(S_2)\texttt{else}(S_3)) = 2 + size(S_1) + size(S_2) + size(S_3) & & \\
size(\texttt{if}_2(B)\texttt{then}(S_2)\texttt{else}(S_3)) = 1 + size(S_2) + size(S_3) & & \\
size((S_1)\texttt{orelse}(S_2)) = 2 + size(S_1) + size(S_2) & & \\
size((C)\texttt{orelse}_2(S_2)) = 1 + int(C) + size(S_2) & & \\
size(\texttt{ppick}(S_1, p_1, \ldots, S_i, p_i)) = 1 + size(S_1) + \ldots + size(S_i) & & \\
size(T_1 || T_2) = 1 + size(T_1) + size(T_2) & &
\end{array}
$$

Property 2 (Characterisation of Normal Forms). Every graph program which is not a value $\left[\mathsf{Id}, G^Q_P\right]$ or $\left[\mathsf{Fail}, G^Q_P\right]$ is reducible using the semantic rules.

Proof. By inspection of the semantic rules, every graph program $[S, G^Q_P]$ different from $[\mathsf{Id}, G^Q_P]$ or $[\mathsf{Fail}, G^Q_P]$ can be matched by a left-hand side of a semantic rule. This is true because S has a top operator which is one of the syntactic constructions and there is a semantic rule which applies to it. Moreover every expression of the form $\langle X, G^Q_P\rangle$ is reducible, because X either contains a graph program $[S, G^Q_P]$ different from $[\mathsf{Id}, G^Q_P]$ or $[\mathsf{Fail}, G^Q_P]$ that so can be matched by a left-hand side of a rule as above, or one of the auxiliary rules can apply.

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

Property 3 (Result Set). Each graph program in the sublanguage that excludes `OneSuc()`, `one()` and `ppick()`, has at most one *result set*.

Intuitively, in a strategy, an application of a transformation with `one()` creates a configuration that is a subset of the configuration computed by the same transformation without `one()`.

Property 4 (Result Set with `one()`*).* A graph program in the sublanguage that excludes `OneSuc()` and `ppick()` (but contains `one()`) produces a result set that is a (non-strict) subset of the same graph program where all occurrences of `one()` have been removed from its strategy.

With respect to the expressive power of the language, it is easy to state, as in [23], the Turing completeness property.

Property 5 (Completeness). The set of all graph programs $\left[S_{\mathcal{R}}, G_P^Q\right]$ is Turing complete, i.e. can simulate any Turing machine.

It is also interesting to consider which sublanguages of our language are Turing complete.

Property 6 (Complete sublanguage). The sublanguage consisting of graph programs where S is built from Id, Fail, rules $(L \Rightarrow R)_M^N$, sequential composition (;), iteration (while), and `orelse` is Turing complete.

The same result could be obtained by replacing `orelse` with the conditional construct if then else. Perhaps more surprising is the fact that Turing machine computations can be simulated by a term rewriting system consisting of just one rule, using a strategy that forces the reduction steps to take place at specific positions, as shown by Dauchet [16]. Given a sequence of transitions in the Turing machine, Dauchet [16] shows how to build a rewrite rule to simulate the transitions, using a strategy to build S-deep-only derivations, selecting the position for rewriting according to the instruction used by the machine. It follows that the sublanguage consisting of focusing operators, sequential composition (;), iteration (`while`) and rule application $(L \Rightarrow R)_M^N$, together with Id, Fail, `setPos()` and `setBan()` is Turing complete. The `setPos()` construct can be used to simulate Dauchet's strategy, by moving the focus of rewriting to the corresponding subterm after each rewrite step. Building a strategy expression for Dauchet's system is a task left for future work.

6 Related Work and Conclusion

The strategy language defined in this paper is part of the PORGY system [2], 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. The implementation uses the small-step operational semantics given above. Some of these semantic rules require a copy of the graph program; this is done efficiently in PORGY thanks to the cloning functionalities of the underlying TULIP system [5].

Graph rewriting is implemented in a variety of tools. In AGG [19], application of rules can be controlled by defining *layers* and then iterating through and across layers. PROGRES [39] allows users to define the way rules are applied and includes non-deterministic constructs, sequence, conditional and loops. The Fujaba [32] Tool Suite offers a basic strategy language, including conditionals, sequence and method calls, but no parallelism. GROOVE [37] permits to control the application of rules, via a control language with sequence, loop, random choice, try()else() and simple function calls. In GReAT [6] 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 [21] uses the concept of search plans to represent different matching strategies. GP [36] 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.

None of the languages above has focusing 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. 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.

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 [11] and Stratego [42]. It can be applied to terms as a particular case (since terms are just trees). When applied to trees, the constructs dealing with applications and strategies are similar to those found in Elan or Stratego. The focusing sublanguage on the other hand can be seen as a lower level version of these languages, because term traversals are not directly available in our language but can be programmed using focusing constructs.

The PORGY environment is still evolving. The strategy language could be enhanced by allowing, for instance, more parallelism (using techniques from the K semantic framework [38]) and focusing constructs to deal with properties of edges. Verification and debugging tools for avoiding conflicting rules or non-termination for instance are planned for future work.

Acknowledgements. We thank the members of the PORGY team for many valuable discussions. This work was supported by Inria's "Associate team" programme: PORGY project.

References

1. O. Andrei. *A Rewriting Calculus for Graphs: Applications to Biology and Autonomous Systems.* PhD thesis, Institut National Polytechnique de Lorraine, 2008.
2. O. Andrei, M. Fernández, H. Kirchner, G. Melançon, O. Namet, and B. Pinaud. PORGY: Strategy driven interactive transformation of graphs. In *Proceedings of TERMGRAPH 2011, Saarbrucken, April 2011.* EPTCS, 2011.
3. O. Andrei and H. 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.
4. O. Andrei and H. 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.
5. D. Auber. Tulip – A huge graph visualization framework. In P. Mutzel and M. Jünger, editors, *Graph Drawing Software*, Mathematics and Visualization Series, pages 105–126. Springer Verlag, 2003.

6. D. Balasubramanian, A. Narayanan, C. P. van Buskirk, and G. Karsai. The Graph Rewriting and Transformation Language: GReAT. *ECEASST*, 1, 2006.
7. E. Balland, P. Brauner, R. Kopetz, P.-E. Moreau, and A. Reilles. Tom: Piggybacking Rewriting on Java. In F. Baader, editor, *RTA*, volume 4533 of *LNCS*, pages 36–47. Springer, 2007.
8. H. Barendregt. *The Lambda Calculus, Its Syntax and Semantics*. North Holland, 1981.
9. H. Barendregt, M. van Eekelen, J. Glauert, J. R. Kennaway, M. Plasmeijer, and M. 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.
10. K. Barthelmann. How to construct a hyperedge replacement system for a context-free set of hypergraphs. Technical report, Universität Mainz, Institut für Informatik, 1996.
11. P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and C. Ringeissen. An overview of ELAN. *ENTCS*, 15, 1998.
12. T. Bourdier, H. Cirstea, D. J. Dougherty, and H. 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.
13. M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. Stratego/XT 0.17. A language and toolset for program transformation. *Science of Computer Programming*, 2008. Special issue on Experimental Systems and Tools.
14. A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. 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.
15. B. 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.
16. M. Dauchet. Simulation of Turing machines by a left-linear rewrite rule. In *Proc. of RTA'89*, volume 355 of *Lecture Notes in Computer Science*, pages 109–120. Springer, 1989.
17. E. W. Dijkstra. *Selected writings on computing - a personal perspective*. Texts and monographs in computer science. Springer, 1982.
18. H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1-3*. World Scientific, 1997.
19. C. Ermel, M. Rudolf, and G. Taentzer. The AGG approach: Language and environment. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 2: Applications, Languages, and Tools*, pages 551–603. World Scientific, 1997.
20. M. Fernández and O. Namet. Strategic programming on graph rewriting systems. In *Proceedings International Workshop on Strategies in Rewriting, Proving, and Programming, IWS 2010*, volume 44 of *EPTCS*, pages 1–20, 2010.
21. R. Geiß, G. V. Batz, D. Grund, S. Hack, and A. Szalkowski. GrGen: A Fast SPO-Based Graph Rewriting Tool. In *Proc. of ICGT*, volume 4178 of *LNCS*, pages 383–397. Springer, 2006.
22. A. Habel, J. Müller, and D. Plump. Double-pushout graph transformation revisited. *Mathematical Structures in Computer Science*, 11(5):637–688, 2001.

23. A. Habel and D. 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.
24. M. Hanus. Curry: A multi-paradigm declarative language (system description). In *Twelfth Workshop Logic Programming, WLP'97, Munich*, 1997.
25. S. L. P. Jones. *Haskell 98 language and libraries: the revised report.* Cambridge University Press, 2003.
26. C. Kirchner, F. Kirchner, and H. Kirchner. Strategic computations and deductions. In *Reasoning in Simple Type Theory. Studies in Logic and the Foundations of Mathematics, vol.17*, pages 339–364. College Publications, 2008.
27. Y. Lafont. Interaction nets. In *Proceedings of the 17th ACM Symposium on Principles of Programming Languages (POPL'90)*, pages 95–108. ACM Press, 1990.
28. S. Lucas. Strategies in programming languages today. *Electr. Notes Theor. Comput. Sci.*, 124(2):113–118, 2005.
29. N. Martí-Oliet, J. Meseguer, and A. Verdejo. Towards a strategy language for Maude. *Electr. Notes Theor. Comput. Sci.*, 117:417–441, 2005.
30. N. Martí-Oliet, J. Meseguer, and A. Verdejo. A rewriting semantics for Maude strategies. *Electr. Notes Theor. Comput. Sci.*, 238(3):227–247, 2008.
31. O. Namet. *Strategic Modelling with Graph Rewriting Tools.* PhD thesis, King's College London, 2011.
32. U. Nickel, J. Niere, and A. Zündorf. The FUJABA environment. In *ICSE*, pages 742–745, 2000.
33. M. J. Plasmeijer and M. C. J. D. van Eekelen. *Functional Programming and Parallel Graph Rewriting.* Addison-Wesley, 1993.
34. G. D. Plotkin. A structural approach to operational semantics. *J. Log. Algebr. Program.*, 60-61:17–139, 2004.
35. D. Plump. Term graph rewriting. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 2: Applications, Languages, and Tools*, pages 3–61. World Scientific, 1998.
36. D. Plump. The Graph Programming Language GP. In S. Bozapalidis and G. Rahonis, editors, *CAI*, volume 5725 of *LNCS*, pages 99–122. Springer, 2009.
37. A. Rensink. The GROOVE Simulator: A Tool for State Space Generation. In *AGTIVE*, volume 3062 of *LNCS*, pages 479–485. Springer, 2003.
38. G. Rosu and T.-F. Serbanuta. An overview of the K semantic framework. *J. Log. Algebr. Program.*, 79(6):397–434, 2010.
39. A. Schürr, A. J. Winter, and A. Zündorf. The PROGRES Approach: Language and Environment. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 2: Applications, Languages, and Tools*, pages 479–546. World Scientific, 1997.
40. Terese. *Term Rewriting Systems.* Cambridge University Press, 2003. M. Bezem, J. W. Klop and R. de Vrijer, eds.
41. R. Thiemann, C. Sternagel, J. Giesl, and P. 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.
42. E. 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.
43. E. Visser. A survey of strategies in rule-based program transformation systems. *J. Symb. Comput.*, 40(1):831–873, 2005.