Maribel Fernández, Hélène Kirchner, Bruno Pinaud, Jason Vallet

To cite this version:
Maribel Fernández, Hélène Kirchner, Bruno Pinaud, Jason Vallet. Porgy Strategy Language: User Manual. [Research Report] Université de Bordeaux, LaBRI; Inria Bordeaux Sud-Ouest; King’s College London. 2018. hal-01566525v2

HAL Id: hal-01566525
https://hal.archives-ouvertes.fr/hal-01566525v2
Submitted on 25 Jul 2018

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.
Abstract. This document provides the specification of PORGY’s strategy language: concrete syntax illustrated with examples.

PORGY is a visual, interactive modelling tool based on port graph rewriting. In PORGY, system states are represented by port graphs, and the dynamic evolution of the system is defined via port graph rewrite rules. Strategy expressions are used to control the application of rules, more precisely, strategy expressions indicate both the rule to be applied at each step in a rewriting derivation, and the position in the graph where the rule is applied (the latter is done via focusing constructs).

Some of the strategy constructs are strongly inspired from term rewriting languages such as ELAN [Borovanský et al., 1998], Stratego [Visser, 2001] and Tom [Balland et al., 2007]. Focusing operators are not present in term rewriting languages (although they rely on implicit traversal strategies). The direct management of positions in strategy expressions, via the distinguished position and banned subgraphs in the target graph and in a located port graph rewrite rule are original features of the language and are managed using positioning constructs.

This document describes the concrete syntax of strategy expressions, explains how the different kinds of constructs are used, and provides examples.

For more information on PORGY we refer the reader to [Pinaud et al., 2012] (interactive features), [Andreï et al., 2011] for preliminary version of the language and, [Fernández et al., 2012,Vallet et al., 2015] (social network examples).

Concrete Syntax for Writing Strategies

The syntax of the strategy language is given in Table 1. Strategy expressions are generated by the grammar rules from the non-terminal $S$. A strategy expression combines applications of located rewrite rules, generated by the non-terminal $A$, and position updates, generated by the non-terminal $U$, using focusing expressions generated by $F$. 
The syntax presented here extends the one in [Fernández et al., 2012] by including a language to define subgraphs of a given graph by specifying simple properties, expressed with attributes of nodes, edges and ports.

Let \( L, R \) be port graphs; \( M, N \) subgraphs of \( R \); \( W \) a subgraph of \( L \); \( n, k \in \mathbb{N} \); \( \pi_i = \ldots = \pi_n \in [0, 1] ; \sum_{i=1}^{n} \pi_i = 1 \). Let \( attribute \) be an attribute; \( e \) a valid expression (quoted string, integer or double); \( function \) a computable function with arguments in \( parameters_list \); “\( script.py \)” a Python script which returns the probability distribution. \([x]\) means the item \( x \) is optional.

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

**Comments.** We use C-style comments (/ * ...comments... * /) for general multi-line comments and C++-style comments for one line comment (/// ...\n).

**Rule Constructs.** The simplest transformation is a located rule, which can only be applied to a located graph $G^Q_P$ if at least a part of the redex is in $P$, and does not involve $Q$. The syntax $T \parallel T'$ represents simultaneous application of the transformations $T$ and $T'$ on disjoint subgraphs of $G$; it succeeds if both are possible concurrently, and it fails otherwise.

- **pick**($T_1, ..., T_n, H$) picks one of the transformations for application, according to the probability distribution $H$. If $T$ and $T'$ have respective probabilities $\pi$ and $\pi'$, $T \parallel T'$ has probability $\pi \times \pi'$.
- **all**($T$) denotes all possible applications of the transformation $T$ on the located graph at the current position, creating a new located graph for each application. In the derivation tree, this creates as many children as there are possible applications.
- **one**($T$) computes only one of the possible applications of the transformation and ignores the others; more precisely, it makes a choice between all the possible applications, with equal probabilities.

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

- **Focusing constructs**
  - **crtGraph**, **crtPos** and **crtBan**, applied to a located graph $G^Q_P$, return respectively the whole graph $G$, $P$ and $Q$.
  - **property**($F, Elem[,Expr]$), applied to a located graph $G^Q_P$, is used to select elements of $G^Q_P$ filtered by $F$ that satisfy a certain property, specified by $Expr$. It can be seen as a filtering construct: if the expression $F$ defines a subgraph $G$ then **property**($F, Elem, Expr$) returns a subgraph $G'$ of $G$ that satisfies the decidable property $Expr$. Depending on the value of $Elem$, the property is evaluated on nodes, ports, or edges, allowing us for instance to select the red nodes and red edges, or nodes with active ports, as shown in examples below. If $Expr$ is not specified, it is considered as the Boolean expression true to allow to select all elements indicated by $Elem$.
    - **property**($F, node, Name == "Add")$ returns all the nodes of the subgraph defined by the expression $F$ that are named Add.
    - **property**($F, port, Active == "true")$ returns all the nodes of the subgraph defined by the expression $F$ that have at least one port with an attribute $Active$ with the value true.
* `property(F, node, Colour == Valid)` returns all the nodes of the subgraph defined by the expression `F` that have the same values for the attributes `Colour` and `Valid`.

* `property(F, edge, StateE > 3)` returns all the edges (including the nodes at their extremities) of the subgraph defined by the expression `F` that have an attribute `StateE` with a value greater than 3.

* `property(F, node, Name =~ ^Num[0-9]$/) returns all the nodes of the subgraph defined by the expression `F` with a name valid over the regular expression `^Num[0-9]$` (the name must start by the string “Num” and terminate by a number). This syntax is inspired by languages such as Perl, Java or the more recent C++11.

* `property(F, port, Name == "Principal" && State == "Active")` returns all nodes having at least one port named “Principal” which also has an attribute `State` set to “Active”.

* `ngb(F, node, Name == "Add")`, returns all the nodes that are adjacent to nodes named `Add` in `F` but are not in `F` themselves (i.e., it returns the neighbours of the nodes in `F` named `Add`).

* `ngb(F, port, Active == "true")` returns all the nodes not already in `F` that have a port with an attribute `Active` set to the value `true`.

* `ngb(F, edge, State > 3)` returns the nodes (not already in `F`) at the other extremity of edges connected to nodes in `F`, where the edge has an attribute `State` with a value greater than 3.

* `ngb(F, edge)` returns all the nodes adjacent to nodes in `F` and not already in `F`.

• `cup (∪), [cap] (∩) and \ are union, intersection and complement of port graphs; \emptySet (\emptySet) denotes the empty graph. We assume the usual priorities (e.g., intersection has priority over union) and operations of the same priority are evaluated left to right.

We can combine multiple Property operators using intersection `∩` to filter multiple times. For example, to select the nodes in the subgraph denoted by `Pos` that are names `Mult` and that have at least one port named `Aux` we write:

```latex
all(property(Pos, node, Name == "Mult") ∩
       property(Pos, port, Name == "Aux"))
```

When nodes have more than one port, strategies

```latex
all(property(F, port, Name == "P") ∩
       property(F, port, State == "Active"))
```
and

\[ \text{all(property(F, port, Name == "P" \\& State == "Active"))} \]

are not equivalent. The first strategy returns nodes having at least one port named “P” and another port (same port or not) with the attribute \textit{State} set to “Active”. The latter strategy returns nodes having at least one port which satisfies both conditions at the same time.

- \texttt{ppick(F_{1}, \ldots, F_{n}, II)} picks one of the positions for application, according to the given probability distribution.

- **Determine Constructs.**
  - \texttt{one(F)} returns one node in \( F \) chosen at random and \texttt{all(F)} returns the full \( F \).
  - **Update Constructs.**
    - \texttt{setPos(D)} (resp. \texttt{setBan(D)}) sets the position subgraph \( P \) (resp. \( Q \)) to be the graph resulting from the expression \( D \). It always succeeds (i.e., returns id).
    - \texttt{update(function_name\{parameters_list\})} updates attributes and their values in the graph using an external TULIP plugin, with given parameters. The plugin must be loaded by TULIP and in the “PORGY” group\(^4\). The syntax is the following:
      
      \[ \text{update("plugin_name"\{param1 : value, \ldots, param}_n \ : \ value\}) } \]

      If a parameter is not given, its default value will be used. If the plugin does not have parameters using \texttt{update("plugin_name")} is enough. If a plugin name or a parameter name is not valid, an error will be raised and the strategy will not be executed. This is useful to update global properties of the graph, in order to focus on specific nodes. For example, in social networks, selecting a “central” node. This is also a way of interfacing with another language (e.g. a Python program or a plugin written outside PORGY).

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

- **Comparison constructs:**
  - \( C \) includes comparison operators for graphs and a matching construct that checks whether a rule matches the current graph.
    - \( F = F' \) returns \texttt{id} if both expressions denote isomorphic port graphs (same sets of nodes, ports and edges), otherwise returns \texttt{fail}. \( F != F' \) returns \texttt{id} if the expressions do not denote isomorphic port graphs, otherwise returns \texttt{fail}. Similarly \( F[\text{subSet}]F' \subset \) checks whether \( F \) denotes

\(^4\) See the TULIP documentation for more information on plugins.
a subgraph of $F'$. We have also included an additional operation, which, although derivable from the rest of the language, facilitates the implementation: $\text{isEmpty}(F)$ returns $\text{id}$ if $F$ denotes the empty graph and fail otherwise. It is defined as $F = \text{emptySet}$.

- **match($T$)** returns $\text{id}$ if there exists a subgraph isomorphism from the left-hand side of $T$ to the current graph taking into account the current position and banned subgraphs. In other words, $\text{match}(T)$ can be seen as an abbreviation of the strategy $\text{if(one}(T)\text{)then(id)else(fail)}$ (see below), but it is directly implemented to improve its efficiency.

- Strategies $S$ are defined with the additional following constructs:
  - $\text{id}$ and $\text{fail}$ are two atomic strategies that respectively denote success and failure.
  - The expression $S;S'$ represents sequential application of $S$ followed by $S'$.
  - $\text{if}(S)\text{then}(S')\text{else}(S'')$ checks if the application of $S$ on a copy of $G_P^Q$ returns $\text{id}$, in which case $S'$ is applied to the original $G_P^Q$, otherwise $S''$ is applied to the original $G_P^Q$. If $S''$ is not specified then we consider $S'' = \text{id}$.
  - $(S)\text{orelse}(S')$ applies $S$ if possible, otherwise applies $S'$. It fails if both $S$ and $S'$ fail.
  - $\text{repeat}(S)\text{[}(k)\text{]}$ simply iterates the application of $S$ until it fails, but, if $k$ is specified between parenthesis, then the number of repetitions cannot exceed $k$.
  - $\text{while}(S)\text{do}(S')\text{[}(k)\text{]}$ keeps on sequentially applying $S'$ while the expression $S$ succeeds on a copy of the graph. If $S$ fails, then $\text{id}$ is returned. If $k$ between parenthesis is specified, then the number of iterations cannot exceed $k$.
  - $\text{try}(S)$ behaves like $S$ if $S$ succeeds, but if $S$ fails, it still returns $\text{id}$. It is a derived operation which is defined as $(S)\text{orelse}(\text{id})$.
  - $\text{not}(S)$ returns $\text{id}$ (resp. fail) if $S$ fails (resp. succeeds). This is also a derivable construct: it is defined as $\text{if}(S)\text{then(fail)else(id)}$.
  - $\text{ppick}(S_1,\ldots,S_n,\Pi)$ picks one of the strategies for application, according to the given probability distribution. This construct generalises the probabilistic constructs on rules and positions seen above.

**Using a Python Script with $\text{ppick}(T_1,\ldots,T_n,\Pi)$**

The construct $\text{ppick}(T_1,\ldots,T_n,\Pi)$ picks one of the $T$ operation according to the probability distribution $\Pi$. $\Pi$ can be given as a list of probabilities, e.g., $\text{ppick}(T_1,\ldots,T_n,\{p_1,\ldots,p_n\})$ where $\sum_{i=1}^n p_i = 1$ or a Python script, e.g., $\text{ppick}(T_1,\ldots,T_n,\text{"script.py"})$ which computes and returns the probabilities.

The basename of the given Python script file (i.e. filename without path information and extension) is used as the name of the function to call inside the Python script. The function must have 5 parameters which are in this order: the graph used to apply the rules on (TULIP graph python object), a list of rules
to test (array of strings), the position subgraph and the banned subgraph (both
graphs described by a Boolean property). The script must return a Python array
of strings (TULIP library does not support conversion from Python dictionary to
C++ object) which has as elements the name of a rule followed by its application
probability and so on for each rule. Note that, all modifications made by the
Python script on the graph structure are not kept. A sample Python script is
given in Listing 1. See the TULIP Python documentation for more details on
using the TULIP Python API.

```
from tulip import tlp

def script(graph, rules, position, ban):
    # graph: the graph where the rule are applied
    # rules: string array of all rules used in the ppick operator
    # position: Tulip Boolean property for the Position set
    # ban: Tulip Boolean properties for the Ban set
    # for position and ban, a 'True' indicates the element belongs to the set.
    # proba: a rule name followed by its probability (converted to a string)
    proba = []
    for r in rules:
        proba.append(r)
        proba.append(str(1/len(rules)))
    return proba
```

Listing 1: Sample Python3 script to compute probabilities for a ppick operator
given a set of rules.

**Macros**

The user interface of PORGY allows to define multiple strategies. From one strat-
egy, it is possible to call other strategies by surrounding the name of the strategy
to use by ‘#’. When the strategy is executed, each string ‘#x#’ is replaced by
the content of x. This operation is accessible through a mouse context menu.

**Concrete Syntax for Computing Attributes Values**

The value of any attribute of any element (nodes or ports) in the right-hand
side can be computed with a formula. There is a dedicated user interface in the
“Algorithm” configuration tab associated to each rule. All formulas associated to
a rule are stored into each rule and are separated by a semi-colon. Table 2 gives
the associated grammar. Up to now, we only support expressions for attributes
of type double or integer without mixing them.
Let $r \in \mathbb{R}$; Let $L = (V_L, P_L, E_L, D_L)$ and $R = (V_R, P_R, E_R, D_R)$ be port graphs (resp. left and right-hand side); with $V$ the set of nodes, $P$ the set of ports, $E$ the set of edges, $D$ the associated record;

$n_L \in V_L$, $e_L \in E_L$, $p_L \in P_L$ (resp. $n_R$, $e_R$ and $p_R$ for $R$); 
attribute be an existing attribute.

$\Omega_L$ (resp. $\Omega_R$) generates attribute values in records $D$ associated with $L$ (resp. $R$);

$[S]$ means the item $S$ is optional.

<table>
<thead>
<tr>
<th>(Expression)</th>
<th>$S ::= \Omega_R = \Psi \cdot [S]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Attribute access)</td>
<td>$\Omega_R ::= \text{node}(n_R).\text{attribute} \mid \text{edge}(e_R).\text{attribute}$</td>
</tr>
<tr>
<td></td>
<td>$\mid \text{port}(p_R).\text{attribute}$</td>
</tr>
<tr>
<td></td>
<td>$\Omega_L ::= \text{node}(n_L).\text{attribute} \mid \text{edge}(e_L).\text{attribute}$</td>
</tr>
<tr>
<td></td>
<td>$\mid \text{port}(p_L).\text{attribute}$</td>
</tr>
<tr>
<td>(Instruction)</td>
<td>$\Psi ::= r \mid \Omega_L \mid (\Psi) \mid \lnot \Psi \mid \Psi , \text{op} , \Psi$</td>
</tr>
<tr>
<td></td>
<td>$\mid \text{random}(r) \mid \text{max}(\Psi, \Psi) \mid \text{min}(\Psi, \Psi)$</td>
</tr>
<tr>
<td></td>
<td>$\text{op} ::= + \mid - \mid \times \mid / \mid %$</td>
</tr>
</tbody>
</table>

**Table 2. Grammar for rule formulas**

For example, to compute a new value for an attribute called $\text{Sigma}$ of a node $n$ where the value is a maximum between the old value and a ratio between the value of another attribute over a random number is written:

$$\text{node}(n).\text{Sigma} = \text{max} (\text{edge}(e).\text{Influence}/\text{random}(1), \text{node}(n).\text{Sigma});$$

where $n$ and $e$ (and if necessary $p$) are the internal TULIP element identifier. This information is available through the “Get Information” interactor.

An Instruction always returns a double number and follows the standard mathematical priority. The construct $\text{random}(r)$ returns a double number $0 < x < r$; $\text{max}(x, y)$ (resp. $\text{min}(x, y)$) returns the maximum (resp. minimum) of both arguments.

**References**


