Porgy Strategy Language: User Manual

Maribel Fernández, Hélène Kirchner, Bruno Pinaud, Jason Vallet, Janos Varga

▶ **To cite this version:**

HAL Id: hal-01566525

https://hal.archives-ouvertes.fr/hal-01566525v4

Submitted on 30 Jun 2020

Porgy Strategy Language: User Manual

Maribel Fernández[1], Hélène Kirchner[2], Bruno Pinaud[3], Jason Vallet[3], and
János Varga[1]

[1] King's College London, Department of Informatics, Strand, London WC2R 2LS,
UK `firstname.lastname@kcl.ac.uk`
[2] Inria, 200 avenue de la Vieille Tour, 33405 Talence, France
`helene.kirchner@inria.fr`
[3] University of Bordeaux, LaBRI CNRS UMR 5800, 33405 Talence Cedex, France
`firstname.lastname@u-bordeaux.fr`

Abstract. This document provides concrete syntax illustrated with examples for the PORGY's strategy language and the language for rule algorithm and conditions.
Last update: June 30, 2020

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. The complete formal syntax is described in Fernández et al. (2019).

For more information on PORGY we refer the reader to Pinaud et al. (2012) (interactive features), Fernández et al. (2019) for preliminary version of the language, Fernandez et al. (2018) (social network examples) and Varga (2018) on rule application conditions.

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;
$n, k \in \mathbb{N}$; $\pi_{i=1\ldots n} \in [0,1]$; $\sum_{i=1}^{n} \pi_i = 1$. 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.

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

Table 1. Concrete Syntax of the Strategy Language.

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 $(/*\ldots comments\ldots*/)$ for general multi-line comments and C++-style comments for one line comment $(//\ldots\backslash n)$.

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

- `ppick`$(T_1,\ldots,T_n,\Pi)$ picks one of the transformations for application, according to the probability distribution Π. If T and T' have respective probabilities π and π', $T \parallel T'$ has probability $\pi \times \pi'$.
- `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_P^Q, return respectively the whole graph G, P and Q.
  - `property`$(F, Elem[, Expr]\})$, applied to a located graph G_P^Q, is used to select elements of G_P^Q filtered by F that satisfy a certain property, specified by $Expr$. It can be seen as a filtering construct: if the expression F defines a subgraph G then `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`$(F, \mathtt{node}, Name == \text{``}Add''\text{)}$ returns all the nodes of the subgraph defined by the expression F that are named Add.

- * `property`$(F, \texttt{port}, Active == \text{``}true\text{''})$ 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`$(F, \texttt{node}, Colour == Valid)$ returns all the nodes of the subgraph defined by the expression F that have the same values for the attributes $Colour$ and $Valid$.
  - * `property`$(F, \texttt{edge}, StateE > 3)$ 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`$(F, \texttt{node}, Name =\sim \text{``}\hat{}\,\text{Num[0-9]\$''})$ 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, \texttt{port}, Name == \text{``}Principal''\&\&State == \text{``}Active'')$ returns all nodes having at least one port named "Principal" which also has an attribute $State$ set to "Active".

- `ngb`$(F, Elem[, Expr]\}$, applied to a located graph G_P^Q, returns a subset of the neighbours (i.e., adjacent nodes) of F according to $Expr$. When `edge` is used as the element (i.e., when we write `ngb`$(F, \texttt{edge}, Expr)$), it returns all the neighbours of F connected to F via edges which satisfy the expression $Expr$.

  - * `ngb`$(F, \texttt{node}, Name == \text{``}Add'')$, 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`$(F, \texttt{port}, Active == \text{``}true'')$ returns all the nodes not already in F that are adjacent to nodes that have a port with an attribute $Active$ set to the value $true$.
  - * `ngb`$(F, \texttt{edge}, State > 3)$ 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`$(F, \texttt{edge})$ returns all the nodes adjacent to nodes in F and not already in F.

- $[cup]$ $(\cup)$, $[cap]$ $(\cap)$ and $\backslash$ 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 $\cap$ 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:

$$\texttt{all}(\texttt{property}(Pos, \texttt{node}, Name == \text{``}Mult'')\cap$$
$$\texttt{property}(Pos, \texttt{port}, Name == \text{``}Aux''))$$

When nodes have more than one port, strategies

$$\texttt{all}(\texttt{property}(F, \texttt{port}, Name == \text{``}P\text{''}) \cap$$
$$\texttt{property}(F, \texttt{port}, State == \text{``}Active\text{''}))$$

and

$$\texttt{all}(\texttt{property}(F, \texttt{port}, Name == \text{``}P\text{''} \&\& State == \text{``}Active\text{''}))$$

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 *State* set to "Active". The latter strategy returns nodes having at least one port which satisfies both conditions at the same time.

- $\texttt{ppick}(F_1, \dots, F_n, \Pi)$ 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:

  $$\texttt{update}(\text{``}plugin\_name\text{''}\{param1 : value, \dots, param_n : value\})$$

  If a parameter is not given, its default value will be used. If the plugin does not have parameters using $\texttt{update}(\text{``}plugin\_name\text{''})$ 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.

---

[4] See the Tulip documentation for more information on plugins.

- $F = F'$ returns id if both expressions denote isomorphic port graphs (same sets of nodes, ports and edges), otherwise returns fail. $F \mathrel{!=} F'$ returns id if the expressions do not denote isomorphic port graphs, otherwise returns fail. Similarly $F[subSet]F'$ ($\subset$) checks whether F denotes a subgraph of F'. We have also included an additional operation, which, although derivable from the rest of the language, facilitates the implementation: $\mathtt{isEmpty}(F)$ returns id if F denotes the empty graph and fail otherwise. It is defined as $F = emptySet$.
- $\mathtt{match}(T)$ returns id if there exists a subgraph isomorphism from the left-hand side of T to the current graph taking into account the current position and banned subgraphs. In other words, $\mathtt{match}(T)$ can be seen as an abbreviation of the strategy $\mathtt{if}(\mathtt{one}(T))\mathtt{then}(\mathsf{id})\mathtt{else}(\mathsf{fail})$ (see below), but it is directly implemented to improve its efficiency.

– Strategies S are defined with the additional following constructs:
  - id and fail are two atomic strategies that respectively denote success and failure.
  - The expression $S;S'$ represents sequential application of S followed by S'.
  - $\mathtt{if}(S)\mathtt{then}(S')[\mathtt{else}(S'')]$ checks if the application of S on a copy of G_P^Q returns id, in which case S' is applied to the original G_P^Q, otherwise S'' is applied to the original G_P^Q. If S'' is not specified then we consider $S'' = \mathsf{id}$.
  - $(S)\mathtt{orelse}(S')$ applies S if possible, otherwise applies S'. It fails if both S and S' fail.
  - $\mathtt{repeat}(S)[(k)]$ simply iterates the application of S until it fails, but, if k is specified between parenthesis, then the number of repetitions cannot exceed k.
  - $\mathtt{while}(S)\mathtt{do}(S')[(k)]$ keeps on sequentially applying S' while the expression S succeeds on a copy of the graph. If S fails, then id is returned. If k between parenthesis is specified, then the number of iterations cannot exceed k.
  - $\mathtt{try}(S)$ behaves like S if S succeeds, but if S fails, it still returns id. It is a derived operation which is defined as $(S)\mathtt{orelse}(\mathsf{id})$.
  - $\mathtt{not}(S)$ returns id (resp. fail) if S fails (resp. succeeds). This is also a derivable construct: it is defined as $\mathtt{if}(S)\mathtt{then}(\mathsf{fail})\mathtt{else}(\mathsf{id})$.
  - $\mathtt{ppick}(S_1, \ldots, S_n, \Pi)$ picks one of the stategies 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 $\mathtt{ppick}(T_1, \ldots, T_n, \Pi)$

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

```python
from tulip import tlp
#toy example to compute a equiprobabilistic choice between all rules
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 strategy, 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.

Ω_L (resp. Ω_R) generates attribute values in records D

associated with L (resp. R);

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

(Expression)	$S ::= \Omega_R = \Psi; [S]$
(Attribute access)	$\Omega_R ::= \mathtt{node}(n_R).attribute \mid \mathtt{edge}(e_R).attribute$
	$\mid \mathtt{port}(p_R).attribute$
	$\Omega_L ::= \mathtt{node}(n_L).attribute \mid \mathtt{edge}(e_L).attribute$
	$\mid \mathtt{port}(p_L).attribute$
(Instruction)	$\Psi ::= r \mid \Omega_L \mid (\Psi) \mid -\Psi \mid \Psi \ op \ \Psi$
	$\mid \mathtt{random}(r) \mid \mathtt{max}(\Psi, \Psi) \mid \mathtt{min}(\Psi, \Psi)$
	$op ::= + \mid - \mid \times \mid / \mid \%$

Table 2. Grammar for rule formulas

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

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

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 `random(r)` returns a double number $0 < x < r$; `max(x, y)` (resp. `min(x,y)`) returns the maximum (resp. minimum) of both arguments.

Syntax for Rule Application Conditions

A rule application condition is a textual constraint on the matching subgraph that has to evaluate to **true** for the rewriting to proceed on that particular redex. The grammar presented in this section forms a language for the Boolean expression B that is part of the *Where* attribute of the arrow node in every port graph rewrite rule. The EBNF grammar for B is defined in Table 3. The structure of the grammar was inspired by C++ and follows the operator precedence of C++, too.

Let $L(V_L, E_L, P_L, D_L)$ be a port graph, the left-hand side of a rule, and $G(V_G, E_G, P_G, D_G)$ the port graph being rewritten, with V the set of nodes, P the set of ports, E the set of edges, D the associated record and $n_L \in V_L, e_L \in E_L, p_L \in P_L$. Then $g(L)$ is a *match* of the left-hand side found in G by total port graph morphism g from L to G.

We point out that when referring to n_L, e_L or p_L the user has to use their internally assigned id. Also, due to the implementation of Porgy, there is no **port**terminal in the grammar – they have to be referred to as **node**. This is because the underlying graph engine (Tulip) processes ports as nodes.

Terminal **string** is an arbitrary-length string, in double quotes, made up of letters, digits and symbols and **number** can be any real number. Similarly, **attribute_name** is a valid name of an attribute from D, in double quotes.

We highlight the **NotNode()** operator: it iterates all nodes of G and checks if there exists a node with an attribute **attribute_name** and if the comparison on them evaluates to true. Intuitively, if at least one such node is found in G, **NotNode()** returns false. It is very important to note here that this check is performed on the entire graph G, not just in $g(L)$. This is fundamentally different from the rest of the rule application condition grammar, which only applies to $g(L)$. This is a consequence of the definition of the port graph rewrite rule which states that all variables in the Boolean expression of the *Where* attribute have to occur in L, so that the matching algorithm can work with them. When a match $g(L)$ is found, all variables of B are mapped so that their actual values can be found. However, when checking the absence of a node, we are not constrained by this, because we are not specifying a node in **NotNode()** – we are only specifying an attribute comparison that *must* evaluate to false on all nodes of G.

Examples of rule application conditions are provided below.

Rule Application Condition Examples

Suppose we want to constrain a rule to only apply if the integer value of a certain node attribute is lower than the sum of the value of two other attributes. A rule application condition ensuring this is shown in Listing 1.1.

```
n(1)."Quantity" < n(1)."Capacity" + n(1)."Buffer";
```

Listing 1.1. Rule application condition example, sum of attributes

$\langle node \rangle$	::=	'node' | 'n' '(' n_L ')'
$\langle edge \rangle$	::=	'edge' | 'e' '(' e_L ')'
$\langle port \rangle$	::=	'node' | 'n' '(' p_L ')'
$\langle element\ attr \rangle$	::=	($\langle node \rangle$ | $\langle edge \rangle$ | $\langle port \rangle$) '.' `attribute_name`
$\langle factor \rangle$	::=	`number` | `string`
	|	$\langle element\ attr \rangle$ | '!' $\langle factor \rangle$
	|	'(' $\langle expression \rangle$ ')' | 'random(' $\langle factor \rangle$ ')'
	|	'max(' $\langle expression \rangle$ ',' $\langle expression \rangle$ ')'
	|	'min(' $\langle expression \rangle$ ',' $\langle expression \rangle$ ')'
$\langle term \rangle$	::=	$\langle factor \rangle$ {('*' | '/' | '%') $\langle factor \rangle$}
$\langle expression \rangle$	::=	$\langle term \rangle$ {('+' | '−') $\langle term \rangle$}
$\langle comp\ operator \rangle$	::=	'==' | '!=' | '>' | '<' | '>=' | '<='
$\langle comparison \rangle$	::=	$\langle expression \rangle$ $\langle comp\ operator \rangle$ $\langle expression \rangle$
	|	'NotNode(' `attribute_name` $\langle comp\ operator \rangle$ $\langle expression \rangle$ ')' ';'
$\langle rule\ application\ condition \rangle$	::=	$\langle comparison \rangle$ {$\langle comparison \rangle$}

Table 3. The Rule Application Condition grammar.

Rule application conditions can be evaluated cross-node as well, the only constraint is that all nodes, edges and ports referenced have to exist in the matching isomorphic subgraph. Suppose we want to extend the previous condition by also applying a constraint on some attribute value of nodes 11 and 21. Remember, these ids are to reference objects in the rule LHS only; they are mapped to the isomorphic subgraph during the rewriting step. Then the condition looks as written in Listing 1.2.

```
n(1)."Quantity" < n(1)."Capacity" + n(1)."Buffer";
n(1)."Quantity" > n(11)."Quantity" + n(21)."Quantity";
```
Listing 1.2. Rule application condition example, cross-node

Conditions on edges look similar. For example, if the "Capacity" value of an edge must be greater than the sum of some connecting nodes, we can implement that as presented in Listing 1.3.

```
e(2)."Capacity" > n(1)."Quantity" + n(11)."Quantity"
                + n(21)."Quantity";
```
Listing 1.3. Rule application condition example, edge attribute

Now let us extend the previous example with a condition on ports and let the engine enforce both. We may want to constrain that the value of attribute `Capacity` of port number 5 must be greater than (or equal to) `Quantity * Size` of two attribute nodes. This is enforced by Listing 1.4. Remember, ports are implemented as nodes, so we have to use the node syntax when referring to ports.

```
n(5)."Capacity" >= n(1)."Quantity" * n(1)."Size";
n(5)."Capacity" >= n(2)."Quantity" * n(2)."Size";
```

Listing 1.4. Rule application condition example, port attribute

Lastly, suppose that we only want a rule to apply if a certain node **does not** exist in the graph. The `NotNode()` operator was proposed to enforce such constraints. For example, there can be a condition in the system model to not allow the appearance of a node with `Size` that equals to the sum of `Size` of two other nodes. We can implement that condition as shown in Listing 1.5.

```
NotNode("Size" == n(1)."Size" + n(2)."Size");
```

Listing 1.5. Rule application condition example, NotNode

Bibliography

Fernández, M., H. Kirchner, and B. Pinaud (2019). Strategic Port Graph Rewriting: an Interactive Modelling Framework. *Mathematical Structures in Computer Science 29*(5), 615–662, doi:10.1017/S0960129518000270.

Fernandez, M., H. Kirchner, B. Pinaud, and J. Vallet (2018). Labelled Graph Strategic Rewriting for Social Networks. *Journal of Logical and Algebraic Methods in Programming 96*(C), 12–40, doi:10.1016/j.jlamp.2017.12.005.

Varga, J. (2018). Finding the Transitive Closure of Functional Dependencies using Strategic Port Graph Rewriting. In M. Fernández and I. Mackie (Eds.), *Proceedings Tenth International Workshop on Computing with Terms and Graphs, TERMGRAPH@FSCD 2018, Oxford, UK, 7th July 2018*, Volume 288 of *EPTCS*, pp. 50–62. doi:10.4204/EPTCS.288.5.

Fernández, M., H. Kirchner, and O. Namet (2012). A strategy language for graph rewriting. In G. Vidal (Ed.), *Logic-Based Program Synthesis and Transformation*, Volume 7225 of *Lecture Notes in Computer Science*, pp. 173–188. Springer Berlin Heidelberg, doi:10.1007/978-3-642-32211-2_12.

Pinaud, B., G. Melançon, and J. Dubois (2012). PORGY: A Visual Graph Rewriting Environment for Complex Systems. *Computer Graphics Forum 31*(3), 1265–1274, doi:10.1111/j.1467-8659.2012.03119.x.

Balland, E., P. Brauner, R. Kopetz, P.-E. Moreau, and A. Reilles (2007). Tom: Piggybacking Rewriting on Java. In F. Baader (Ed.), *Rewriting Techniques and Applications (RTA)*, Volume 4533 of *Lecture Notes in Computer Science*, pp. 36–47. Springer, doi:10.1007/978-3-540-73449-9_5.

Visser, E. (2001). Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In *Proceedings of the International Conference on Rewriting Techniques and Applications (RTA'01)*, Volume 2051 of *Lecture Notes in Computer Science*, pp. 357–361. Springer-Verlag, doi:10.1007/3-540-45127-7_27.

Borovanský, P., C. Kirchner, H. Kirchner, P.-E. Moreau, and C. Ringeissen (1998). An overview of ELAN. *ENTCS 15*, doi:10.1016/S1571-0661(05)82552-6.