

# SortalGI plug-in for Grasshopper

## User manual

SortalGI version 1.7

Manual update February 2022

Written by Rudi Stouffs

### Table of content

1. About the SortalGI plug-in	2
2. Installation and update	3
3. Common terms	5
4. Data types	8
5. Starting on a SortalGI-based parametric model	12
6. Creating a shape	14
7. Manipulating a shape	18
8. Creating a rule	21
9. Applying a rule	25
10. Creating and applying flows (composite rules)	29
11. Specifying shape descriptions	35
12. Specifying predicates	45
13. Specifying directives	49
Appendix A. A formal notation for shape descriptions	52
Appendix B. Description functions	57
Appendix C: A formal notation for flow descriptions	60
Appendix D: FAQ	64

## 1. About the SortalGI plug-in

A shape rule combines a specification of recognition and manipulation (search and replace). A shape rule is commonly specified in the form  $lhs \rightarrow rhs$ , where the left-hand-side ( $lhs$ ) of the rule specifies the pattern to be recognized and the manipulation of the current shape then involves replacing the recognized  $lhs$  by the right-hand-side ( $rhs$ ) of the shape rule in the shape under investigation. Recognition necessarily applies under some transformation, for example, a similarity transformation, and the resulting manipulation must occur under the same transformation for both  $lhs$  and  $rhs$ . That is, applying a rule  $a \rightarrow b$  to a given shape  $s$  involves determining a transformation  $f$  such that  $f(a)$  is a part of  $s$  ( $f(a) \leq s$ ), following which  $s$  is replaced by  $s - f(a) + f(b)$ .

A shape grammar generally defines a collection of rules together with an initial shape; then, the language defined by a shape grammar is the set of shapes generated by the rules from the initial shape. However, from a user's point of view, any collection of rules that serves a particular purpose can be considered a shape grammar, whether or not it requires a particular initial shape or, instead, can be applied to a wide variety of (initial) shapes.

*Sortal* grammars extend on shape grammars. Where shape grammars commonly rely on a combination of line segments and labelled points, *sortal* grammars take a modular representational approach, allowing for a wide variety of geometric and non-geometric elements to be included in the specification of shape rules and grammars. *Sortal* grammars utilize *sortal* structures as representational structures, where these structures are defined as formal compositions of other, primitive, *sortal* structures, termed *sorts*. As such, *sortal* grammars constitute a class of formalisms for design grammars and benefit from the fact that every component *sort* specifies a partial order relationship on its individuals and forms, defining both the matching operation and the arithmetic operations for rule application.

A shape grammar interpreter is the engine that supports the application of shape rules, including recognition and manipulation (search and replace). The SortalGI plug-in for Grasshopper encapsulates the SortalGI *sortal*/shape grammar interpreter and makes part of its functionality available within Rhino/Grasshopper. It allows the user to create and apply shape and description rules within the Grasshopper environment. The SortalGI interpreter supports emergence, that is, subshape recognition is unrestricted by how a shape has been drawn or structured.

Plug-in development by Bianchi Dy and Rudi Stouffs

System development by Bui Do Phuong Tung and Rudi Stouffs

Research and development led by Rudi Stouffs

## 2. Installation and update

Installation applies to both Windows and Mac (Rhino 6 and Rhino 7 only).

Installation takes two main steps. Firstly, install the SortalGI library in a place where Rhino can find it; this is required for every major update (e.g., from v1.6.0 to v1.7.0). Secondly, install the SortalGI plug-in (user objects) for Grasshopper; this is always required, also in the case of a minor update (e.g., from v1.7.0 to v1.7.1).

If you have not yet done so, download the latest SortalGI update from Food4Rhino (<http://www.food4rhino.com/app/sortalgi-shape-grammar-interpreter>) or sortal.org (<http://www.sortal.org/downloads/plugin.html>) and unzip the file.

### Step 1 [Windows]: Installing the SortalGI library

This step applies to initial installation as well as—to some extent—every major update (e.g., from v1.6.0 to v1.7.0).

There are generally two locations where Rhino expects the SortalGI library to be installed, either:

- C:\Users\me\AppData\Roaming\McNeel\Rhinoceros\6.0\scripts or equivalent on your computer
- C:\Program Files\Rhino 6\Plug-ins\IronPython\Lib or equivalent on your computer

You can identify both locations from Rhino's 'Module Search Paths' dialog:

- a) Open Rhino
- b) Type EditPythonScript in the Rhino Command box
- c) In the Rhino Python Editor window, select 'Options...' from the Tools menu
- d) Note the available 'Module Search Paths'

If you'd like, you can choose any other location and add it to the 'Module Search Paths'

The following steps install the SortalGI library and make it accessible to Rhino:

- e) Copy-paste the content of the folder 'lib' (the subfolders 'sortal' and 'site-packages') into the preferred location
- f) Add the location of the site-packages subfolder (e.g., C:\Program Files\Rhino 6\Plug-ins\IronPython\Lib\site-packages) into the 'Module Search Paths'
- g) Switch from the 'Files' tab to the 'Script Engine' tab (in the Python Options window).
- h) Check the 'Frames Enabled' option and click 'OK'

Only the installation of the 'sortal' subfolder—replacing it with the newer version—needs to be repeated for every major update.

Note:

- i. Installing the SortalGI library in IronPython\Lib will make the library available for all users but requires administrator access
- ii. A 'site-packages' folder may already exist in IronPython\Lib, it suffices to add the content of 'site-packages' to this folder. It remains important to add this subfolder into the 'Module Search Paths' if not already done so.
- iii. If different versions of the SortalGI library are installed in different locations, the ordering of the respective parent folders in the 'Module Search Paths' will define which version is being used

### Step 1 [Mac]: Installing the SortalGI library

This step applies to initial installation as well as—to some extent—every major update (e.g., from v1.6.0 to v1.7.0).

Copy-paste the content of the folder 'lib' (the subfolders 'sortal' and 'site-packages') into the location Macintosh HD/Users/me/Library/Application Support/McNeel/Rhinoceros/6.0/scripts or equivalent on your computer. Unpack 'site-packages' by moving its content to the 'scripts' folder.

Only the installation of the 'sortal' subfolder—replacing it with the newer version—needs to be repeated for every major update.

Note:

- i. The folder Library may not be visible. If so, select your home directory in the finder, choose "Show view options" from the View menu and check "Show Library Folder"

## Step 2 [Windows/Mac]: Installing the SortalGI plug-in

This step applies to initial installation and every (minor or major) update.

- a) Open Rhino and Grasshopper.
- b) In Grasshopper, choose File > Special Folders > User Object folder.
- c) Copy-paste the content of the folder 'user objects' into this 'User Object' folder (replacing any files with the same name, if already present).

The result should be automatically reflected in Grasshopper. There should be an 'SGI' tab in the Grasshopper Components Tab Panel and if you select the tab it should include all the User Objects (see below). If not, you may want to restart Grasshopper and Rhino for the changes to take effect.

Note:

- i. You can also use the SGI Update component to update the SortalGI components in the Grasshopper Components Tab Panel as well as in the current parametric model (see section 5. Starting on a SortalGI-based parametric model).

*Do note that compatibility between the components of v1.5.0 with respect to previous versions is rather poor, due to the fact that input and output parameters, both in terms of the number of parameters and types of the parameters, have changed quite a bit. Unfortunately, using SGI Update cannot resolve all these changes automatically.*



### 3. Common terms

The following object classes are defined to exchange information between SortalGI components in Grasshopper:

Shape (also denoted lhShape, rhShape, subShape, Shape1 or Shape2)

A Shape object contains the shape's representation as used by the SortalGI engine, together with the corresponding GH geometry (see section 6. Creating a shape).

Rule (also denoted pRule)

A Rule object contains the rule's representation as composed of a left-hand-side Shape object and a right-hand-side Shape object, an identifier Name, an optional description, and, possibly, one or more Predicates and/or Directives (see section 8. Creating a rule). Note that in a limited number of cases, the term Rules is used to allow for both Rule objects and Flow objects.

Flow

A Flow object contains the flow's representation as composed of an R (*flow structure*) object, an identifier Name, and an optional description (see section 10. Creating and applying flows (composite rules)). Note that in some cases, the term Flow is used to allow for either a Flow object or an R (*flow structure*) object.

R (also denoted sequenceR, disjunctionR or negationR)

R is used as a container term allowing for either a Rule object, a Flow object or an R (*flow structure*) object, where the latter underlies a Flow object, omitting the Name and description. However, the terms sequenceR, disjunctionR and negationR only refer to an R (*flow structure*) object.

In addition, the following terms are adopted to denote various information types:

Name (also denoted ruleName or flowName)

The name of a rule or flow must be a unique identifier, consisting only of letters, digits and/or underscores and always starting with a letter or underscore.

Type (also denoted descriptionType or sType)

The term descriptionType or Type is used to denote a description type name. Description types must be predefined before they can be used in the creation of a (description) shape. Description type names must be identifiers, consisting only of letters, digits and/or underscores and always starting with a letter or underscore. The term sType is a container term allowing for either a descriptionType or a spatialType, the latter being a fixed enumeration (see below).

Description (also denoted labelD, D, referenceD, auxRefD, conditionD, tupleD, functionD, expressionD, propertyD, directionD, lengthD, distanceD, valueD, angleD)

The term Description or labelD is used to denote a textual (shape) description, as distinguished from a spatial (shape) description (i.e., spatial elements) (see section 11. Specifying shape descriptions). While a shape description is inherently textual, it can also be specified as a numeric value or a vector. A shape description can also be composed as an

expression or tuple from other descriptions or description parts. The term D is generally used to denote any description or part thereof. There are many different kinds of descriptions, which is reflected in the many variant terms.

#### Tag (also denoted lineTag or refTag)

A tag is a label specified to a spatial element in order to identify it within a corresponding description during rule application. A tag is formed by an identifier—consisting only of letters, digits and/or underscores and always starting with a letter or underscore—preceded by a '#'. Any spatial element can be assigned a tag, but the term lineTag is only used to denote the tag of a line segment.

#### Predicate

A predicate is a formatted textual specification expressing a special condition on the application of a rule (see section 12. Specifying predicates).

#### Directive

A directive is a formatted textual specification expressing a value that is required for the unambiguous execution of the manipulation/replacement part of a parametric-associative rule application (see section 13. Specifying directives).

Finally, the following terms each reference a predefined enumeration of (textual) values:

#### colorMode

Any spatial element can have a color attribute assigned conforming to one of the predefined and preselected color modes: 'graytone', 'opaque', 'maxRGB' (maximum RGB values), 'sumRGB' (sum of RGB values), 'avgRGB' (average RGB values), 'alphaRGB' (alpha blending). Note that 'graytone', next to being limited to grayscales, is the only color mode where black, rather than white, specifies the highest value (see section 4. Data types).

#### spatialType

The spatial element types that the SortalGI plug-in currently supports are 'point', 'line segment', 'plane segment', 'circle', 'ellipse', 'circular arc' and 'quadratic Bezier' (see section 4. Data types).

#### atttype

The non-spatial attribute types that the SortalGI plug-in currently supports are 'labelID', 'color' and line 'thickness' (see section 4. Data types).

#### Op

Within a shape description, two kinds of operators can be used. Numerical expressions allow for the operators '+' (addition), '-' (subtraction), '\*' (multiplication), '/' (division), '%' (modulo operation) and '^' (exponentiation). Conditional expressions allow for '=' (equal), '<>' (not equal), '<' (less than), '>' (greater than), '<=' (less than or equal), '>=' (greater than or equal), '[' (within range) and '{}' (one of).

### Function

Within a shape description, functions allow for additional operations on numbers, texts/strings and tuples, or a combination thereof (see Functions in section 11. Specifying shape descriptions).

### Marking

Within a shape description, tuple markings are either parentheses, angle brackets or square brackets, to enclose the tuple, and commas or semicolons, as separators (either '(', '<', '[', ']', '(', ';>', '<;>', '[;]'). Alternatively, the enclosing marks can be omitted with spaces as separators (").

### Matching

Flows support four different matching approaches: 'greedy', 'possessive', 'lazy' and 'probabilistic' (see section 10. Creating and applying flows (composite rules)).

### Quantifier

A flow or flow structure can be iterated conform a specified quantifier. Aside from the predefined values '?' (zero or once), '\*' (zero, once or more times) and '+' (once or more times), the quantifier can also be composed from a minimum and, optionally, maximum value as in the forms '{min}', '{min,}' and '{min, max}'.

## 4. Data types

Shapes are generally composed of spatial elements. These spatial elements may have non-spatial attributes. Shapes may also include (textual) shape descriptions; in fact, a shape can be made up of only descriptions, only spatial elements, or a combination thereof. An empty shape is also allowed, although the left-hand-side shape of a rule (or *lhShape*) can never be empty.

### Spatial element types

The SortalGI engine currently supports the following spatial element types: points, line segments, plane segments, circles, ellipses, circular arcs and quadratic Bezier curves. These can be created as geometries in Rhino or Grasshopper and converted into shape elements using the SGI Shape or SGI dShape components (see section 6. Creating a shape). Note that circular arcs are not yet available within parametric-associative rules and, if specified, will be ignored.

Each spatial element type defines a *sortal* structure (or *sort*). It may be necessary to refer to a *sortal* structure by its name in order to identify a spatial element within a description. Note that *sortal* structures are different for non-parametric rules and parametric-associative rules (pRule). The former names end with '3D', the latter names with 'P3D'. *Sortal* structures corresponding to different spatial element types are combined under the operation of sum on *sortal* structures. The result is a composite *sortal* structure, e.g.,  $\text{curve3D} = \text{circle3D} + \text{ellipse3D} + \text{arc3D} + \text{bezier3D}$ .

spatial element type	sortal structure	
	non-parametric rules	parametric-associative rules
points	point3D	pointP3D
line segments	lineSeg3D	lineSegP3D
plane segments	planeSeg3D	planesegP3D
circles	circle3D	circleP3D
ellipses	ellipse3D	ellipseP3D
circular arcs	arc3D	-
quadratic Bezier curves	bezier3D	bezierP3D

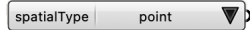
### SGI All Spatial Types



The SGI All Spatial Types component provides a list of all *spatialTypes* as may be present in shapes: 'point', 'line segment', 'plane segment', 'circle', 'ellipse', 'circular arc' and 'quadratic Bezier'.



## Sgi Spatial Types



The SGI Spatial Types selector allows to select from a list of spatialTypes as may be present in shapes: 'point', 'line segment', 'plane segment', 'circle', 'ellipse', 'circular arc' and 'quadratic Bezier'.

### Non-spatial attribute types

The SortalGI engine supports three non-spatial attribute types: descriptions (or labels), colors (or grayscales) and line thicknesses.

Descriptions are textual, in nature, and follow a strict format that allows them to be interpreted and matched by the SortalGI engine. This format allows for quoted strings (e.g., labels), numbers, vectors, tuples thereof, etc.). Descriptions that form part of the left-hand-side or right-hand-side of a rule may include parameters, expressions and references to other descriptions or to spatial elements (see section 11. Specifying shape descriptions).

Colors can be grayscale values or RGB values, depending on the selected color mode. The SortalGI engine distinguishes six color modes:

color mode	explanation
graytone	grayscale values between 0 (white) and 255 (black); the sum of two grayscale values is the maximum of both values. Note that this is the only color mode where the maximum value represents black, rather than white
opaque	RGB color values; the sum of two color values is always the second value
maxRGB	RGB color values; the sum of two color values has as RGB values the maximum values from both colors
sumRGB	RGB color values; the sum of two color values has as RGB values the sum of the respective RGB values from both colors
avgRGB	RGB color values; the sum of two color values has as RGB values the average of the respective RGB values from both colors
alphaRGB	RGB color values and alpha value; the sum of two color values is dependent on their respective alpha values

There are two ways to assign a color attribute to a spatial element. Firstly, the desired color can be specified directly to the original geometry in Rhino, using 'Custom' (instead of 'by Layer' or 'by Parent'). 'Custom' color specifications will be automatically retrieved when converting the geometry into a shape and assigned as a color attribute to the respective spatial element. Secondly, the SGI Attributed Shape component (see section 6. Creating a shape) can be used to assign a color to a spatial element when creating the shape. Note that there is no color attribute specified for points.

The *sortal* structure for descriptions and colors is different for each spatial element type, so as to be able to reference these unambiguously in a description. The *sortal* structure corresponding to a spatial element type is augmented with the *sortal* structures of the respective attribute types under the operation of attribution on *sortal* structures. The result

is a composite *sortal* structure, e.g.,  $\text{attrCircle3D} = \text{circle3D} \wedge (\text{cColor} + \text{cLabelD} + \text{cThickness})$ .

non-spatial attribute type	labels/descriptions	colors	line thicknesses
spatial element type	<i>sortal</i> structure		
points	pLabelD		
line segments	lLabelD	lColor	lThickness
plane segments	plLabelD	plColor	
circles	cLabelD	cColor	cThickness
ellipses	eLabelD	eColor	eThickness
circular arcs	aLabelD	aColor	aThickness
quadratic Bezier curves	bLabelD	bColor	

Note that descriptions may also occur as color attribute (but not yet as line thickness attribute) within a rule, in order to retrieve or assign color values within the rule (see section 6. Creating a shape). However, this is not (yet) applicable to the ‘graytone’ colorMode.

#### SGI Attribute Types



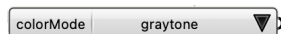
The SGI Attribute Types component provides a list of all attribute types that may be present as non-spatial attributes in shapes: ‘labelD’, ‘color’ and ‘thickness’.

#### SGI All Color Modes



The SGI All Color Modes component provides a list of all colorModes that may apply to color attributes: ‘graytone’, ‘opaque’, ‘maxRGB’, ‘sumRGB’, ‘avgRGB’ and ‘alphaRGB’.

#### SGI Color Modes



The SGI Color Modes selector allows to select from a list of colorModes that may apply to color attributes: ‘graytone’, ‘opaque’, ‘maxRGB’, ‘sumRGB’, ‘avgRGB’ and ‘alphaRGB’.

#### Description types

Descriptions can be assigned as attributes to spatial elements, but descriptions can also form part of the shape alongside spatial elements. In the case of descriptions as attribute, a single *sortal* structure is already made available to contain these descriptions. When descriptions form a direct part of the shape, there may be a need to distinguish descriptions by purpose. For this reason, the SGI Setup component (see section 5. Starting on a SortalGI-based parametric model) accepts a list of description type names, each an identifier, consisting only of letters, digits and/or underscores and always starting with a letter or underscore. For each description type name, a corresponding *sortal* structure is defined and made part of the global *sortal* structure or shape representational structure. For example, in the case of two description type names designBrief and temporaryDescriptions, the resulting *sortal* structure for shapes forming part of non-parametric rules would be:

bezier3D ^ bLabelD + circle3D ^ cLabelD + designBrief + ellipse3D ^ eLabelD + lineSeg3D ^  
lLabelD + planeSeg3D ^ plLabelD + point3D ^ pLabelD + temporaryDescriptions

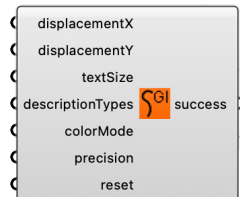
## 5. Starting on a SortalGI-based parametric model

### Creating a new parametric model using SortalGI components

Before adding any other SGI component, you should first add the SGI Setup component. This component initializes the SortalGI engine and makes all functionality available to the model.

If, instead, you add the SGI Setup component after other SGI components, you must arrange/put the component to the back (Ctrl+B or ⌘B) to ensure that the SGI Setup component is executed before all other components.

### SGI Setup



The SGI Setup component initializes the SortalGI engine and allows for some global settings.

Inputs:

- *displacementX*: optional displacement value along the X-axis for the purpose of translating any shape resulting from a rule application; if no displacement value is specified, then the rule application will automatically derive the translation distance from the bounding box of the shape (see section 9. Applying a rule)
- *displacementY*: optional displacement value along the Y-axis for the purpose of translating and spacing multiple shapes resulting from a rule application; if no displacement value is specified, then the rule application will automatically derive the translation distance from the bounding box of the shape (see section 9. Applying a rule)
- *textSize*: text size to visualize any labels or shape descriptions that are attributes to geometries resulting from a SortalGI component
- *descriptionTypes*: list of shape description Types, each identified by its name (see section 8. Specifying shape descriptions for a specification of descriptions)
- *colorMode*: colorMode, or a data tree of spatialType and colorMode pairs, governing how two color values, as attributes of a same spatial element, combine into a single value. Options are 'graytone', 'opaque', 'maxRGB', 'sumRGB', 'avgRGB' or 'alphaRGB' (default is 'opaque').
- *precision*: number of significant figures used for calculations and matching. Empirical evidence has shown that a precision of 6 to 8 significant figures tends to provide the best results (default is 8).
- *reset*: Boolean value specifying whether to reset the SortalGI engine (default is False)

Outputs:

- *success*: True or False indicating success of the setup

### Opening an existing parametric model using SortalGI components

If you find any errors with SortalGI components upon opening an existing parametric model, these might be caused by having older components embedded in the existing model when compared with the SortalGI version installed.

Firstly, check the version number of the specific component. If it is an older (or different) version number, you can use the SGI Update component to automatically update this and

any other components to the installed version. Note that any embedded component in the parametric model contains its own Python code and updating the SortalGI components in the 'UserObjects' folder does not automatically update the embedded components in the model. The SGI Update component will update both the SortalGI components in the 'UserObjects' folder (if instructed to do so) and the embedded components in the current parametric model.

Secondly, if the version number does correspond to the installed version, instead, the problem may relate to a difference in inputs and/or outputs between the specific embedded component in the model and the component present in the Grasshopper Components Tab Panel. In this case, you must replace the embedded component and all its connections using the available component.

### SGI Update



The SGI Update component updates the Python codes in the embedded components in the parametric model to the specified SortalGI version. If specified, it will also update the components in the Grasshopper Components Tab Panel.

#### Inputs:

- *sourceDir*: optional source directory where the SortalGI components should be copied from into the 'UserObjects' folder; if omitted, then only the Python codes of the embedded components in the parametric model will be updated to the current SortalGI version as available in the Grasshopper Components Tab Panel
- *updateThis*: Boolean value specifying whether to execute this SGI Update component (default is False)

#### Outputs:

- *success*: True or False value indicating success of the update

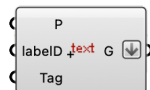
## 6. Creating a shape

Creating shapes using the SGI Shape or SGI dShape components may serve different purposes. Shapes can be operated upon directly (e.g., sum, difference or product). A shape can be used to define the left-hand-side or the right-hand-side of a rule. Rule application also requires an input shape and, optionally, an input subshape (see section 9. Applying a rule).

A shape may consist of points, line segments and plane segments, circles and ellipses, circular arcs and quadratic Bezier curves, as well as shape descriptions. Points may have shape descriptions (or labels) and colors assigned as attributes. The SGI Text Point, SGI Text Curve and SGI Text Surface components allow one to assign a label or shape description as a text to a point, curve or surface. Note that the resulting geometry is only recognized by any of the SGI components, specifically SGI Shape or SGI dShape. Other Grasshopper components will not recognize the text point/curve/surface.

The SGI Shape and SGI dShape components differ in the fact that the latter accepts shape descriptions using an extra input, while the former does not.

### SGI Text Point



The Text Point component creates a labelled point geometry, that is, a point with a label or shape description as attribute. A label must be double-quoted, otherwise it will be considered a shape description. The component can also be used to tag a point (see References in section 11. Specifying shape descriptions).

Inputs:

- *P*: point geometry
- *labelID*: optional text specifying the tag, label or shape Description of the text point (see section 11. Specifying shape descriptions for a specification of descriptions); multiple values can be combined into a single entry by separating them with vertical bars
- *Tag*: optional text specifying the tag, label or shape Description of the text point; multiple values can be combined into a single entry by separating them with vertical bars

Outputs:

- *G*: resulting text point

### SGI Text Curve



The Text Curve component creates a labelled curve geometry, that is, a curve with a label or shape description as attribute. A label must be double-quoted, otherwise it will be considered a shape description. The component can also be used to tag a curve (see References in section 11. Specifying shape descriptions).

Inputs:

- *C*: curve geometry
- *labelID*: optional text specifying the tag, label or shape Description of the text curve (see section 11. Specifying shape descriptions for a specification of descriptions);

multiple values can be combined into a single entry by separating them with vertical bars

- *Tag*: optional text specifying the tag, label or shape Description of the text curve; multiple values can be combined into a single entry by separating them with vertical bars

Outputs:

- *G*: text curve

#### SGI Text Surface



The Text Surface component creates a labelled surface geometry, that is, a surface with a label or shape description as attribute. A label must be double-quoted, otherwise it will be considered a shape description. The component can also be used to tag a surface (see References in section 11. Specifying shape descriptions).

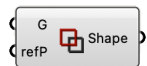
Inputs:

- *S*: surface geometry
- *labelID*: optional text specifying the tag, label or shape Description of the text surface (see section 11. Specifying shape descriptions for a specification of descriptions); multiple values can be combined into a single entry by separating them with vertical bars
- *Tag*: optional text specifying the tag, label or shape Description of the text surface; multiple values can be combined into a single entry by separating them with vertical bars

Outputs:

- *G*: text surface

#### SGI Shape



The SGI Shape component creates a shape from geometry and an optional reference point.

Inputs:

- *G*: geometry of points, lines, polylines, (flat) surfaces, meshes, boundary representations, circles, ellipses, (circular) arcs, quadratic Bezier curves and/or text elements; any part of the geometry not recognized will be ignored
- *refP*: optional reference point; if specified, the geometry will be moved from the reference point to the origin, allowing a shape that will serve as the left-hand-side or right-hand-side to a rule to be drawn or specified spatially separated from the other side of the rule

Outputs:

- *Shape*: Shape object

#### SGI dShape



The SGI dShape component creates a shape from geometry, shape descriptions (see section 11. Specifying shape descriptions) and an optional reference point. The descriptions may be omitted, so may be the geometry, though not both at the same time.

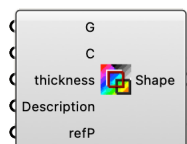
Inputs:

- *G*: geometry of points, lines, polylines, (flat) surfaces, meshes, boundary representations, circles, ellipses, (circular) arcs, quadratic Bezier curves and/or text elements; any part of the geometry not recognized will be ignored
- *Description*: one or more shape Descriptions, each item preceded by the shape description Type and a colon; multiple shape descriptions of the same type can be combined into a single item by separating them with a vertical bar
- *refP*: optional reference point; if specified, the geometry will be moved from the reference point to the origin, allowing a shape that will serve as the left-hand-side or right-hand-side to a rule to be drawn or specified spatially separated from the other side of the rule

Outputs:

- *Shape*: Shape object

SGI Attributed Shape



The SGI Attributed Shape component creates a shape from geometry and, optionally, attribute colors and/or thicknesses, shape descriptions (see section 11. Specifying shape descriptions) and an optional reference point. The descriptions may be omitted, so may be the geometry, though not both at the same time.

Inputs:

- *G*: geometry of points, lines, polylines, (flat) surfaces, meshes, boundary representations, circles, ellipses, (circular) arcs, quadratic Bezier curves and/or text elements; any part of the geometry not recognized will be ignored
- *C*: optional one or more colors, or descriptions referencing color values (the latter only within a rule)
- *thickness*: optional one or more line thickness values, expressed as print width (between 0.0 and 2.0)
- *Description*: optional one or more shape Descriptions, each item preceded by the shape description Type and a colon; multiple shape descriptions of the same type can be combined into a single item by separating them with a vertical bar
- *refP*: optional reference point; if specified, the geometry will be moved from the reference point to the origin, allowing a shape that will serve as the left-hand-side or right-hand-side to a rule to be drawn or specified spatially separated from the other side of the rule

Outputs:

- *Shape*: Shape object

SGI S2G



The SGI S2G component converts any shape into its geometry, colors and shape descriptions.

Inputs:

- *Shape*: Shape object

Outputs:

- *G*: geometry of the Shape object
- *M*: materials (color) for custom preview of the geometry



- *Description:* shape Descriptions of the Shape object (note that any shape description that is assigned as an attribute to part of the geometry of the Shape object is not included as it already forms part of the geometry)

## 7. Manipulating a shape

Following the creation of a shape, various geometrical operations are available as SortalGI components to act upon a shape; e.g., to translate/move a shape, rotate a shape, reflect/mirror a shape and scale a shape. Each of these components takes as input a shape and any additional data required to inform and apply the transformation, and returns the resulting shape. Their operation is quite identical to the corresponding Grasshopper components, except that they act upon a shape.

In addition, there are SortalGI components to union/sum two shapes, intersect/take the product of two shapes and take the difference of one shape with respect to another.

### SGI Move Shape



The SGI Move Shape component moves a shape along a translation vector. This component is very useful to ensure the visualization of shapes resulting from rule application do not overlap and are properly spaced (see section 9. Applying a rule).

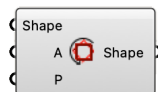
Inputs:

- *Shape*: Shape object
- *T*: translation vector

Outputs:

- *Shape*: resulting Shape object

### SGI Rotate Shape



The SGI Rotate Shape component rotates a shape about the normal vector of a base plane by a specified angle.

Inputs:

- *Shape*: Shape object
- *A*: rotation angle in radians
- *P*: rotation plane

Outputs:

- *Shape*: resulting Shape object

### SGI Mirror Shape



The SGI Mirror Shape component mirrors a shape about a base plane.

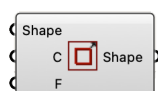
Inputs:

- *Shape*: Shape object
- *P*: mirror plane

Outputs:

- *Shape*: resulting Shape object

### SGI Scale Shape



The SGI Scale Shape component scales a shape about a center of scaling uniformly by a specified scaling factor.

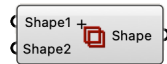
Inputs:

- *Shape*: Shape object
- *C*: center of scaling
- *F*: scaling factor

Outputs:

- *Shape*: resulting Shape object

SGI Sum



The SGI Sum component sums (combines) two shapes together.

Inputs:

- *Shape1*: Shape object
- *Shape2*: another Shape object

Outputs:

- *Shape*: resulting Shape object
- *T*: translation vector that can be used to move/displace the resulting shape wrt the original shapes

SGI Product



The SGI Product component determines the product (intersection) of two shapes.

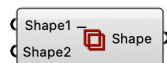
Inputs:

- *Shape1*: Shape object
- *Shape2*: another Shape object

Outputs:

- *Shape*: resulting Shape object
- *T*: translation vector that can be used to move/displace the resulting shape wrt the original shapes

SGI Difference



The SGI Difference component takes the difference (complement) of one shape with respect to another shape.

Inputs:

- *Shape1*: Shape object
- *Shape2*: another Shape object

Outputs:

- *Shape*: resulting Shape object
- *T*: translation vector that can be used to move/displace the resulting shape wrt the original shapes

SGI Sum All



The SGI Sum All component sums (combines) any number of shapes together.

Inputs:

- *Shapes*: list of Shape objects

Outputs:

- *Shape*: resulting Shape object
- *T*: translation vector that can be used to move/displace the resulting shape wrt the original shapes

## 8. Creating a rule

A rule is conceptually specified in the form  $lhs \rightarrow rhs$ , where the left-hand-side (*lhs*) of the rule specifies the pattern to be matched under some transformation and the right-hand-side (*rhs*) specifies the resulting pattern that replaces the matched pattern under the same transformation. That is, applying a rule  $a \rightarrow b$  to a given shape  $s$  involves determining a transformation  $f$  such that  $f(a)$  is a part of  $s$  ( $f(a) \leq s$ ), following which  $s$  is replaced by  $s - f(a) + f(b)$ .

A shape rule is commonly understood to imply that both *lhs* and *rhs* constitute a geometry, possibly including non-geometric attributes, e.g., labels or descriptions. A description rule, then, implies that both *lhs* and *rhs* constitute a shape description of the same shape description type. Combining a shape rule with one or more description rules specifies a compound rule, where the different component rules operate in parallel, although they may interact with each other.

A Rule object specifies such a compound rule although it can be used to specify a shape rule or, alternatively, one or more description rules. That is, which component rules are included depends on the shapes that are provided as *lhs* and *rhs* of the (compound) rule. If the *lhs* does not include any geometry, then the *rhs* may not include any geometry either, as no matching transformation can be determined from an empty shape. With respect to shape descriptions, if either the *lhs* or *rhs* includes a shape description type but the other side does not, then an empty shape description of that type is automatically included in the other side to ensure a full correspondence between shape description types.

Two types of rules are distinguished, parametric-associative rules and non-parametric rules. The latter are the easiest to understand. In the case of a non-parametric rule, the pattern specified by the *lhs* of the rule must match a part of the given shape under a similarity transformation (translation, rotation, reflection and/or uniform scaling). That is, when matching for a square of line segments, any square of line segments from the given shape will do, even if these line segments extend beyond the corner points of the square. The same applies when matching for a rectangle, however, only rectangles with the same ratio between length and width will be matched.

A parametric-associative rule matches a much larger variety of shapes. In principle, when matching a triangle of line segments, any triangle of line segments in the given shape will be matched, irrespective of its shape. The corresponding transformation is a topological transformation though there is no mathematical representation for such a transformation (unlike for a similarity transformation). However, some constraints do apply. Specifically, colinear, parallel and perpendicular lines (and points) are automatically identified in the *lhs* and considered as constraints for matching. Thus, specifying a right-angled triangle as the *lhs* will only match right-angled triangles in the given shape, however, specifying an equilateral or isosceles triangle as the *lhs* will have no effect, any triangle in the given shape will be matched.

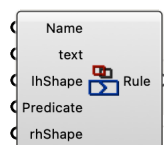
While in some cases it may be difficult to predict the exact matching results of the *lhs* of a parametric-associative rule, the matching mechanism broadly follows the following steps:

1. Identify all (infinite) lines that carry any line segment in the *lhs*.
2. Identify all (infinite) lines that carry any line segments in the given shape.

3. Enumerate all combinations of lines from the given shape that match the number of lines for the *lhs*.
4. Eliminate all combinations that do not preserve parallelism and perpendicularity between lines as specified by the *lhs*.
5. Identify all intersection points of (infinite) lines in the *lhs* and note whether the intersection point falls inside, outside or is an endpoint of any line segment on each infinite line.
6. Do the same for the remaining combinations of (infinite) lines for the given shape:
  - a. Eliminate any combinations where an inside intersection point for the *lhs* is not matched with an inside intersection point for the given shape.
  - b. Eliminate any combinations where an intersection point that is an endpoint for the *lhs* is not matched with an intersection point that is either an endpoint or an inside point for the given shape.
7. For the *lhs*, identify all endpoints of line segments on these (infinite) lines and note their ordering also with respect to the inside intersection points.
8. Do the same for the given shape and eliminate any remaining combinations where two intersection points in the *lhs* are contained within a single line segment and the corresponding intersection points in the given shape are not.

A similar mechanism applies to plane segments.

#### SGI Rule



The SGI Rule component creates a **non-parametric** rule from a left-hand-side (*lhs*) and a right-hand-side (*rhs*) shape, a name, a (optional) brief description, and any number of predicates (see section 12. Specifying predicates). If a shape description type is present as part of one shape (*lhs* or *rhs*) but absent from the other shape, an empty shape description of that type is automatically added to the other shape within the rule.

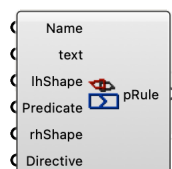
Inputs:

- *Name*: rule Name (may contain only letters, digits and underscores, not starting with a digit); this rule name should be unique
- *text*: optional, brief description of the rule
- *lhShape*: Shape object representing the left-hand-side of the rule
- *Predicate*: optional Predicate or list thereof
- *rhShape*: Shape object representing the right-hand-side of the rule

Outputs:

- *Rule*: non-parametric Rule object

#### SGI pRule



The SGI pRule component creates a **parametric-associative** rule from a left-hand-side (*lhs*) and a right-hand-side (*rhs*) shape, a name, a (optional) brief description, and any number of predicates and/or directives (see sections 12. Specifying predicates and 13. Specifying directives). If a shape description type is present as part of one shape (*lhs* or *rhs*) but absent

from the other shape, an empty shape description of that type is automatically added to the other shape within the rule.

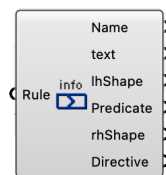
Inputs:

- *Name*: rule Name (may contain only letters, digits and underscores, not starting with a digit); this rule name should be unique
- *text*: optional, brief description of the rule
- *lhShape*: Shape object representing the left-hand-side of the rule
- *Predicate*: optional Predicate or list thereof
- *rhShape*: Shape object representing the right-hand-side of the rule
- *Directive*: optional Directive or list thereof

Outputs:

- *pRule*: parametric-associative Rule object

### SGI Rule Info



The SGI Rule Info component deconstructs any (parametric-associative or non-parametric) rule into its left-hand-side and right-hand-side shapes, its rule name and description, and its predicates and directives, if any.

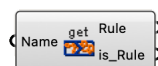
Inputs:

- *Rule*: Rule object

Outputs:

- *Name*: rule Name
- *text*: rule description
- *lhShape*: left-hand-side Shape object
- *Predicate*: zero, one or more Predicates
- *rhShape*: right-hand-side Shape object
- *Directive*: zero, one or more Directives

### SGI Get Rule



The SGI Get Rule component retrieves a (parametric-associative or non-parametric) rule or flow (see section 10. Creating and applying flows (composite rules)) by its name.

Inputs:

- *Name*: rule or flow Name

Outputs:

- *Rule*: (non-parametric or parametric-associative) Rule or Flow object (or null)
- *is\_Rule*: Boolean value indicating whether the object is a Rule (True) or Flow (False) object

### SGI All Rules



The SGI All Rules component retrieves a list of all existing (parametric-associative or non-parametric) rules and flows and their names.

No inputs:

#### Outputs:

- *ruleNames*: list of all rule Names
- *Rules*: list of all (parametric-associative or non-parametric) Rule objects
- *flowNames*: list of all flow Names
- *Flows*: list of all Flow objects

#### Importing SortalGI rules from a different parametric model

Rules and flows (see section 10. Creating and applying flows (composite rules)) created using SortalGI components in one parametric model can be exported to a text file and subsequently imported into another parametric model to be applied there. The SGI Export component exports any number of rules and flows into a text file in the SDL (Sortal Description language) format. The SGI Import component imports any text file in the SDL format and makes the rules and flows available for rule application.

#### SGI Export



The SGI Export component writes any number of rules and/or flows (see section 10. Creating and applying flows (composite rules)) into a text file in the SDL (Sortal Description language) format. Note that if the file already exists, its content will be overwritten.

#### Inputs:

- *filePath*: file path the rules and flows will be written to
- *Rules*: list of Rule and/or Flow objects

No outputs.

#### SGI Import



The SGI Import component reads a text file in the SDL (Sortal Description language) format and makes the rules and/or flows available for application.

#### Inputs:

- *filePath*: the path of the SDL file that the rules and flows will be read from

#### Outputs:

- *rules*: list of Rule objects
- *flows*: list of Flow objects
- *messages*: list of messages indicating success or failure reading the file and creating the Rule or Flow objects



## 9. Applying a rule

Applying a rule to a given shape involves determining a transformation under which the left-hand-side (*lhs*) of the rule is a part of the given shape. That is, rule application involves two steps: recognition and manipulation (search and replace); recognition implies matching the *lhs* of the rule under some transformation to a part of the given shape and manipulation implies replacing the recognized *lhs* by the right-hand-side (*rhs*) of the shape rule under the same transformation.

Obviously, the *lhs* of a shape rule may match multiple parts of the same given shape. These matches may correspond to different but similar parts, e.g., if the *lhs* of a non-parametric rule specifies a square, the rule will match any square in the given shape independent of its location, rotation, reflection or scale (under a similarity transformation). However, these matches may also apply to the same part in different ways. Again, if the *lhs* of a non-parametric rule specifies a square, which has 90° rotational symmetry, and the *rhs* specifies the same square moved diagonally, then any square in the given shape will amount to four matches as the square may be moved into any of its four diagonal directions.

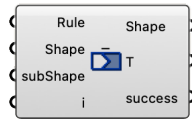
The SortalGI plug-in distinguishes four rule application components: the first one, SGI Apply, applies only a single match (either randomly selected or specified by its index), while the second one, SGI Apply All, applies all matches in parallel, returning as many results as there are matches, and the third one, SGI Apply All Together, applies all (or a selection of) matches together (in parallel), returning a single, combined result. The fourth one, SGI Derive, takes a series of rules as input and applies each rule in sequence, returning all intermediate results as well as the final result. All four components accept both a shape and an optional subshape. If specified, the latter must be a subshape, that is, part of, the former. If a subshape is specified then recognition/matching is restricted to the subshape. This allows one to reduce the number of matches where appropriate. Manipulation will always apply to the entire shape.

Finally, a fifth component, SGI Matches, does not actually apply the given rule but, instead yields all the matching shapes to the left-hand-side of the rule. As such, the Matches component can be used to search for a given shape. As the results will be returned in a list, serving this list as input to a rule application component will ensure rule application (both recognition and manipulation) applies separately to each result, if possible, allowing for a divide-and-conquer approach that may be more efficient for subsequent rule applications.

For any of these components, every resulting shape is accompanied by a translation vector. In the case of SGI Apply (and SGI Apply All Together), the translation vector allows the resulting shape to be visualized aside from the original shape, along the X-axis. In the case of SGI Apply All (and SGI Matches), the translation vectors allow the resulting shapes to be visualized one above the other, along the Y-axis, and aside from the original shape, along the X-axis. In the case of SGI Derive, the translation vectors allow the resulting shapes to be visualized one aside from the other, and from the original shape, along the X-axis. The extent of the translation vector is specified by the *displacementX* and *displacementY* values provided to the SGI Setup component or, if no value is provided, by the bounding box of the original shape (see section 3. Starting on a SortalGI-based parametric model).

All rule application components accept parametric-associative and non-parametric rules.

## SGI Apply



The SGI Apply component determines all possible matches of a rule with respect to a shape (or subshape), but applies only a single one, either randomly selected or as specified by an index value.

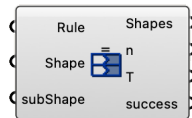
Inputs:

- *Rule*: Rule object
- *Shape*: Shape object to apply the rule to
- *subShape*: optional Shape object to restrict matches to; if specified, this shape must be a subshape, that is, part of, the shape *Shape*
- *i*: optional index to select which match to consider for rule application; a value of –1 (default) selects a random match, any number outside the index range yields the last one among the list of matches

Outputs:

- *Shape*: Shape object resulting from rule application; if no match is found then the original shape is returned
- *T*: translation vector to allow the shape to be drawn next to the original shape, along the X-axis
- *success*: True or False indicating whether a match was found or not

## SGI Apply All



The SGI Apply All component determines and applies all possible matches of a rule with respect to a shape (or subshape).

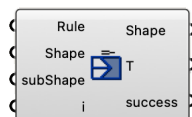
Inputs:

- *Rule*: Rule object
- *Shape*: Shape object to apply the rule to
- *subShape*: optional Shape object to restrict matches to; if specified, this shape must be a subshape, that is, part of, the shape *Shape*

Outputs:

- *Shapes*: list of Shape objects corresponding to the number of rule applications found; if no match is found then the original shape is returned
- *n*: number of matches found, corresponds to the length of the lists *Shapes* and *T*
- *T*: list of translation vectors to allow the shapes to be drawn one above the other, along the Y-axis, and next to the original shape, along the X-axis
- *success*: True or False indicating whether at least one match was found or not

## SGI Apply All Together



The SGI Apply All Together component determines and applies (in parallel) all or a specified selection of possible matches of a rule with respect to a shape (or subshape), and combines them into a single shape. This behavior corresponds to the shape schema  $x \rightarrow \sum F(\text{prt}(x))$  when *F* refers to a single rule.

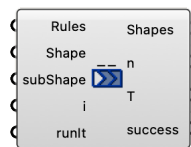
Inputs:

- *Rule*: Rule object
- *Shape*: Shape object to apply the rule to
- *subShape*: optional Shape object to restrict matches to; if specified, this shape must be a subshape, that is, part of, the shape *Shape*
- *i*: optional list of indices to select which matches to include in the result; in case of an empty list all matches are included

Outputs:

- *Shape*: Shape object resulting from all or the specified selection of rule applications; if no match is found then the original shape is returned
- *T*: translation vector to allow the shape to be drawn next to the original shape, along the X-axis
- *success*: True or False indicating whether a match was found or not

### SGI Derive



The SGI Derive component acts as a sequence of SGI Apply components. Given a list of rules, it applies each in sequence.

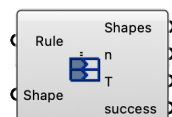
Inputs:

- *Rules*: list of Rule objects
- *Shape*: Shape object to apply the first rule to
- *subShape*: optional Shape object to restrict the first match to, or a list of shape objects to restrict consecutive matches to; if specified, the shape must be a subshape, that is, part of, the input shape of the respective rule
- *i*: optional index to select which matches to consider for rule application; a value of – 1 (default) selects a random match, any number outside the index range yields the last one among the list of matches; may be specified as a list of indices
- *runIt*: Boolean value specifying whether to execute the component or not

Outputs:

- *Shapes*: list of Shape objects, one for each successful rule application; if no match is found for any rule then the input shape for the first rule is returned
- *n*: number of successful rule applications, corresponds to the length of the lists *Shapes* and *T* if greater than 0
- *T*: list of translation vectors to allow the shapes to be drawn one next to the other and to the original shape, along the X-axis
- *success*: list of True or False values indicating for each rule object whether at least one match was found or not

### SGI Matches



The SGI Matches component determines all possible matches of a rule with respect to a shape. Note that depending on the right-hand-side of the rule, identical matches may result corresponding to otherwise distinct rule applications.

Inputs:

- *Rule*: Rule object

- *Shape*: Shape object to match the rule to

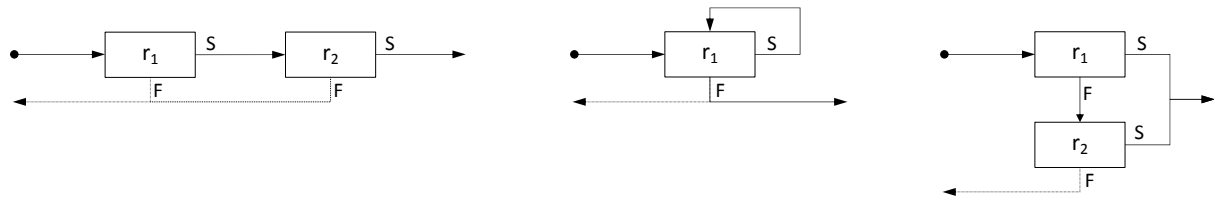
Outputs:

- *Shapes*: list of Shape objects corresponding to the rule matches
- *n*: number of matches found, corresponds to the length of the lists *Shapes* and *T*
- *T*: list of translation vectors to allow the shapes to be drawn one above the other, along the Y-axis, and next to the original shape, along the X-axis
- *success*: True or False indicating whether at least one match was found or not

## 10. Creating and applying flows (composite rules)

Flows are composite rules embedding algorithmic patterns such as sequence, iteration and selection. Two rules apply in sequence if upon a successful application of the first rule, the second rule applies to the result from the application of the first rule. Similarly, a single rule can be applied iteratively if upon every successful application of the rule, a new application is attempted on the result of the last successful application. Selection, on the other hand, specifies two (or more) alternative rules, where each may be attempted to be applied but, as soon as one rule applies successfully, the remaining rules are ignored.

A conceptual, diagrammatic representation of the three algorithmic patterns sequence (left), iteration (middle) and selection (right):



In the diagrams above, rule application flows from left to right, starting from the dot and continuing with rule  $r_1$ . Upon successful (S) application of rule  $r_1$ , the application flow continues as indicated. Upon failure (F), depending on the pattern and its parameters, the application flow may continue (solid arrow) or backtracking may occur (dashed arrow). Note that backtracking is a more complex process of revisiting previous rules in search of alternative solutions that cannot be fully captured in the diagrams above. In general, a rule may have multiple potential applications and, within a flow, only one application will be selected to proceed with. Backtracking, then, may lead to the subsequent selection of an alternative application to proceed with. In a sequence, every rule must apply successfully, or none at all will apply. An iteration customarily ends at some point. Whether this ending is considered success (and the flow proceeds) or failure (and backtracking occurs) is dependent on the minimum number of iterations specified. A selection only fails if all of its rules fail to apply. Note that any rule within a flow may itself be composed as a flow, such that flows can be hierarchically composed of sub-flows, each with their own pattern of sequence, iteration or selection.

### Flow matching approaches

Backtracking can be suppressed by adopting a possessive matching approach rather than a greedy matching approach, as is the default. Specifically, assigning a possessive matching approach to a sub-flow ensures that when this sub-flow has successfully applied, no backtracking to this sub-flow will occur from any later point in the super-flow. However, backjumping may occur to any non-possessive sub-flow that precedes the possessive sub-flow, upon which the flow application may eventually return to the possessive sub-flow. Obviously, if every sub-flow is assigned a possessive matching, backjumping will be suppressed as well.

An alternative to greedy and possessive matching is lazy matching. Where greedy (and possessive) matching will iterate until failure or the maximum number of iterations has been achieved, lazy matching will end an iteration as soon as the minimum number of iterations has been achieved. Similar to greedy matching, backtracking may occur, but in the case of iterative backtracking under lazy matching, an additional iteration will be tried rather than backtracking to the previous iteration.

Finally, a probabilistic approach to iteration is provided as well. In this case, a random number of times to iterate, within the minimum-maximum range, will be selected and tried.

## SGI Flow Matching

flow Matching greedy ▼

The SGI Flow Matching selector allows to select from a list of flow Matching approaches: greedy, possessive, lazy and probabilistic.

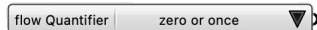
### Iteration flow quantifiers

An iteration is necessarily defined by the number of iterative applications that are expected or allowed. We adopt a quantifier (from regular expressions) to indicate both the minimum and maximum number of iterations allowed:

quantifier	number of iterations		
	min	max	explanation
?	0	1	Zero or one time — no backtracking occurs
*	0	-	Zero, one or more times — the iteration proceeds until rule application fails, no backtracking occurs
+	1	-	One or more times — the iteration proceeds until rule application fails, backtracking only occurs if the rule fails at the very first time
{ <i>n</i> }	<i>n</i>	<i>n</i>	Exactly <i>n</i> times — the iteration proceeds until rule application fails, backtracking occurs if fewer than <i>n</i> applications succeed
{ <i>n</i> ,}	<i>n</i>	-	<i>n</i> or more times — the iteration proceeds until rule application fails, backtracking occurs if fewer than <i>n</i> applications succeed
{ <i>n</i> , <i>m</i> }	<i>n</i>	<i>m</i>	Any number of times between <i>n</i> and <i>m</i> — the iteration proceeds until rule application fails, backtracking occurs if fewer than <i>n</i> applications succeed

## SGI Flow Quantifier

flow Quantifier zero or once ▼

The SGI Flow Quantifier selector allows to select from a sublist of iteration flow Quantifiers, specifically, '?', '\*' and '+'.  


### SGI Flow Quantifier {*n*,*m*}

{ min {*n*,*m*} Quantifier max }

The SGI Flow Quantifier {*n*,*m*} component composes an iteration flow quantifier from a minimum and (optional) maximum value, specifying, respectively, the minimum and maximum number of times a flow (structure) would be iterated upon. If a maximum value is omitted, the flow (structure) would be iterated upon any number of times, but at least the minimum number of times.

Inputs:

- *min*: minimum number of times a flow structure would be iterated upon
- *max*: optional, maximum number of times a flow would be iterated upon

Outputs:

- *Quantifier*: iteration flow Quantifier expressed as text, in the form {*min*}, {*min*,} or (*min*, *max*)

## Flow creation

The SortalGI plug-in distinguishes four flow creation components. The first creates a flow structure as a composite rule embedding a sequence pattern and, optionally, an iteration pattern nesting the sequence pattern. The second does the same for a selection instead of a sequence patterns. The third creates a flow structure as a composite rule embedding the negation of a rule, flow or flow structure, reversing success and failure. Finally, the last component creates a proper flow, from a flow structure, a flow name and an optional description. In addition, a list component serves to ensure the proper ordering of rules and/or flow structures.

### SGI List



The SGI List component is an auxiliary component that constructs a list from any number of inputs. Note that the inputs are guaranteed to be appended to the list in order.

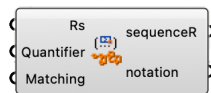
Inputs:

- *x, y*: by default, the component offers two input parameters, but additional input parameters can be inserted; all inputs specified will be added to the resulting list

Outputs:

- *L*: list resulting from the specified number of inputs

### SGI Rule Sequence



The SGI Rule Sequence component creates a rule sequence from a list of rules, flows and/or flow structures and, optionally, a quantifier and matching approach. A rule sequence is a flow structure embedding the sequence pattern. It applies successfully if each of the component rules (or flows) applies, in order. Backtracking may occur within the sequence. The optional quantifier serves to add an iteration pattern nesting the sequence pattern, where the quantifier specifies the minimum and maximum number of times the sequence can be applied iteratively.

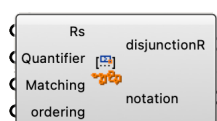
Inputs:

- *Rs*: ordered list of Rule objects, Flow objects and/or R (*flow structure*) objects (sequence, disjunction or negation)
- *Quantifier*: optional Quantifier specifying how many times the sequence may or should be iterated (one of '?', '\*', '+', '{n}', '{n,}', or '{n,m}'); default is no iteration, which is equivalent to '{1}'.
- *Matching*: optional Matching approach (greedy: 'G', possessive: 'PO', lazy: 'L', or probabilistic: 'PR'); default is greedy.

Outputs:

- *sequenceR*: R (*flow structure*) object (sequence)
- *notation*: formal, textual specification of the rule sequence (see Appendix C: A formal notation for flow descriptions for an explication of the format)

### SGI Rule Disjunction



The SGI Rule Disjunction component creates a rule disjunction from a list of rules, flows and/or flow structures and, optionally, a quantifier, matching approach and ordering. A rule

disjunction is a flow structure embedding the selection pattern. It applies successfully if any one of the component rules (or flows) applies. The component rules (or flows) are tried either in the order specified, or in a random order. The optional quantifier serves to add an iteration pattern nesting the selection pattern, where the quantifier specifies the minimum and maximum number of times the selection can be applied iteratively.

Inputs:

- *Rs*: ordered list of Rule objects, Flow objects and/or R (*flow structure*) objects (sequence, disjunction or negation)
- *Quantifier*: optional Quantifier specifying how many times the selection may or should be iterated (one of '?', '\*', '+', '{n}', '{n,}', or '{n,m}'); default is no iteration, which is equivalent to '{1}'.
- *Matching*: optional Matching approach (greedy: 'G', possessive: 'PO', lazy: 'L', or probabilistic: 'PR'); default is greedy.
- *ordering*: optional Boolean value specifying the order in which the component rules (or flows) are tried; default (True) is in the order specified, otherwise (False) a random order is applied

Outputs:

- *disjunctionR*: R (*flow structure*) object (disjunction)
- *notation*: formal, textual specification of the rule disjunction (see Appendix C: A formal notation for flow descriptions for an explication of the format)

## SGI Rule Negation



The SGI Rule Negation component creates a rule negation from a single rule, flow or flow structure. A rule negation is a flow structure that applies successfully only if application of the underlying rule, flow or flow structure fails and vice versa . Note that a rule negation only checks whether a rule applies and rule application itself is necessarily suppressed, as rule application would imply that rule negation failed and, as such, backtracking would occur.

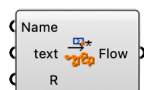
Inputs:

- *R*: Rule object, Flow object or R (*flow structure*) object (sequence or disjunction)

Outputs:

- *negationR*: R (*flow structure*) object (negation)
- *notation*: formal, textual specification of the rule negation (see Appendix C: A formal notation for flow descriptions for an explication of the format)

## SGI Flow



The SGI Flow component creates a flow from a rule, flow or flow structure (sequence, disjunction or negation), a flow name and an optional, brief description. A flow can be considered a composite rule; as such, the flow name must not only be unique among all flows but also among all rules.

Inputs:

- *Name*: flow Name (may contain only letters, digits and underscores, not starting with a digit); this flow name should be unique
- *text*: optional, brief description of the flow
- *R*: Rule object, Flow object or R (*flow structure*) object (sequence, disjunction or negation)

Outputs:



- *Flow*: resulting Flow object

### SGI Flow Info



The SGI Flow Info component provides the flow name, flow description, flow specification and the list of rules that form part of this specification of the flow.

Inputs:

- *Flow*: Flow object

Outputs:

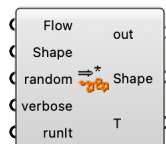
- *Name*: flow Name
- *text*: flow description
- *notation*: formal, textual flow specification

### Flow application

The SortalGI plug-in distinguishes two flow application components: the first one, SGI Apply Flow, accepts any flow or flow structure, and returns only a single outcome, while the second one, SGI Apply All Flow, accepts any flow but not any flow structure, and returns all outcomes (as can be determined through backtracking).

For either components, every resulting shape is accompanied by a translation vector. In the case of SGI Apply Flow, the translation vector allows the resulting shape to be visualized aside from the original shape, along the X-axis. In the case of SGI Apply All Flow, the translation vectors allow the resulting shapes to be visualized one above the other, along the Y-axis, and aside from the original shape, along the X-axis. The extent of the translation vector is specified by the *displacementX* and *displacementY* values provided to the SGI Setup component or, if no value is provided, by the bounding box of the original shape (see section 5. Starting on a SortalGI-based parametric model).

### SGI Apply Flow



The SGI Apply Flow component determines a single outcome from the application of a flow or flow structure (sequence, disjunction or negation) onto a shape.

For each rule in the flow (or flow structure), if there are multiple potential rule applications, by default a random selection is made, although this behavior can be overridden by setting the random parameter input to False, in which case the first rule application is always selected. The latter ensures the same result each time the flow is applied to the same shape.

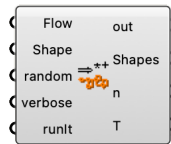
Inputs:

- *Flow*: Flow object or R (*flow structure*) object (sequence, disjunction or negation)
- *Shape*: Shape object to apply the flow (or flow structure) to
- *random*: optional Boolean value indicating whether a random (True, default) or fixed (False) selection is made from among the applications for one rule; a fixed selection yields the same result each time the component is executed
- *verbose*: optional Boolean value specifying whether the process should be verbally recorded (default is True)
- *runIt*: Boolean value specifying whether to run this component (default is False)

Outputs:

- *out*: verbal description of the flow application process, if the verbose input parameter is set to True
- *Shape*: Shape object resulting from the flow (or flow structure) application
- *T*: translation vector to allow the shape to be drawn next to the original shape, along the X-axis

#### SGI Apply All Flow



The SGI Apply All Flow component determines all outcomes (as can be determined through backtracking) from the application of a flow onto a shape.

For each rule in the flow, if there are multiple potential rule applications, by default a random selection is made, although this behavior can be overridden by setting the random parameter input to False, in which case the first rule application is always selected. The latter ensures the same result each time the flow is applied to the same shape.

#### Inputs:

- *Flow*: Flow object
- *Shape*: Shape object to apply the flow to
- *random*: optional Boolean value indicating whether a random (True, default) or fixed (False) selection is made from among the applications for one rule; a fixed selection yields the same result each time the component is executed
- *verbose*: optional Boolean value indicating whether the process should be verbally recorded (default is True)
- *runIt*: Boolean value specifying whether to run this component (default is False)

#### Outputs:

- *out*: verbal description of the flow application process, if the verbose input parameter is set to True
- *Shapes*: list of Shape objects corresponding to the number of flow applications found
- *n*: number of matches found, corresponds to the length of the lists *Shapes* and *T*
- *T*: list of translation vectors to allow the shapes to be drawn one above the other, along the Y-axis, and next to the original shape, along the X-axis

## 11. Specifying shape descriptions

We use the term shape description to distinguish it from a rule description. The latter is a textual description that is used to explain the purpose of a rule to the user; it is not interpreted by the SortalGI engine. Shape descriptions, on the other hand, follow a strict format that allows them to be interpreted and matched by the SortalGI engine (see Appendix A. A formal notation for shape descriptions for an explication of the format).

### Parametric shape descriptions

Shape descriptions are parametric in nature, that is, when adopted as the left-hand-side (*lhs*) of a (shape) description rule, a shape description may contain one or more parameters that can be matched onto parts of the description under investigation. When adopted as the right-hand-side (*rhs*) of a (shape) description rule, a shape description may also contain parameter references although the parameters should have already been specified in the corresponding *lhs*, such that the value of the parameter reference in the *rhs* can be taken from the matching of the *lhs*. Obviously, shape descriptions that do not form part of a shape description rule should not contain any parameters or parameter references, otherwise matching will necessarily fail.

Example ('description' is the shape description type and 'a' is a parameter):

description: 4.0

description: a

### Shape description types

A single shape or rule may specify more than one description. For example, one shape description may be used to constrain rule application while another may serve to count the number of rule applications performed on the shape. In order to be able to correctly match shape descriptions belonging to the *lhs* and the *rhs* of the rule, shape descriptions must be typed, that is, each shape Description that is not used as an attribute to a point must be preceded by its Type (type and description are separated by a colon). Shape description types must be prescribed in the SGI Setup component (see section 5. Starting on a SortalGI-based parametric model).

Multiple shape descriptions may share the same description type. These can be collected in a single line, using a vertical bar to separate the various descriptions.

Examples:

min\_width: 10

colors: "black" | "white"

### SGI Description



The SGI Description component composes one or more shape descriptions from a number of description types and description values. If a single type is input with a list of values, then the values are all assigned to the same type. Otherwise, each type is assigned a single value and if there is an insufficient number of values, then the last one is repeated.

Inputs:

- *Type*: one or more shape description Types
- *D*: one or more Descriptions (or parameters), without Type specification

Outputs:

- *Description*: Description text or list thereof, including the Type specification

### Description literals

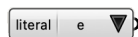
Literal values in descriptions may be numbers, double quoted strings or predefined keywords. The latter include `e`, `nil`, `pi`, `true` and `false`. `e` and `nil` are equivalent and represent an 'empty' entity. Depending on the context, the 'empty' entity may be interpreted to denote zero, an empty string or an empty tuple. The literals `pi`, `true` and `false` denote the numbers ' $\pi$ ', 1 and 0, respectively.

Examples:

status: `true`

list: `e`

### SGI Description Literals



The SGI Description Literals selector allows to select from a list of literals for shape descriptions.

### Description tuples

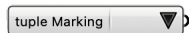
While shape descriptions are specified in textual form, they can be structured as nested lists/tuples. Tuples should be enclosed using either parentheses, angle brackets or square brackets. A top-level tuple may have the enclosing brackets omitted. The entities within a tuple should be separated using either commas or semicolons. Again, a top-level tuple may have the separating marks omitted.

Examples:

segment: `<(0, 0), (1, 0)>`

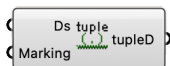
cubes: `("l:", 10, "c:", (0, 0), "r:", 0) ("l:", 10, "c:", (5, 5), "r:", 45)`

### SGI Tuple Markings



The SGI Description Markings selector allows to select from a list of Markings for description tuples.

### SGI Description Tuple



The SGI Description Tuple component composes a description tuple from a list of values and the specified markings (or none).

Inputs:

- *Ds*: list of Descriptions as tuple elements
- *Marking*: tuple Marking, either parentheses, angle brackets or square brackets to enclose the tuple, and commas or semicolons as separators (either '`(,)`', '`<,>`', '`[,]`', '`(;)`', '`<;>`', '`[;]`'); alternatively, the enclosing marks can be omitted with spaces as separators ('', default)

Outputs:

- *tupleD*: Description tuple as text

### Description parameters

A description parameter is a variable term that is specified by an identifier (any sequence of letters, digits and/or underscores starting either with a letter or underscore) and embedded

in the *lhs* of a description rule. Under rule application, the parameter will be matched to a literal or a tuple. If the parameter forms part of a string expression (see String expressions below), this literal can be any part of a literal string. If the parameter forms part of a tuple, it matches a specific element of the tuple, unless it is signified by a kleene star ('\*') or a kleene plus ('+'), in which case it can match any subsequence of elements of the tuple, respectively, including or excluding an empty subsequence. The use of a kleene star or kleene plus signifier allows for the matching of variable length tuples.

Examples:

fixed\_length: <"Fixed", var1> <var2, var3> var4

variable\_length: (0, 0) (x1, y1) remainder\*

### Parameter conditionals

Any description parameter may be specified a conditional that constrains the possible values of this parameter. The conditional must follow the parameter and both must be separated only by a question mark ('?'). The conditional may be either enumerative or equational, or specify a range. An enumerative conditional explicates a finite set of possible values. This set must contain either all numbers or all (double quoted) strings, and the set must be enclosed using curly brackets. An equational conditional specifies a numeric equality or inequality on the parameter, in the form of a conditional operator ('=', '<>', '<', '<=', '>', or '>=') and operand. The operand must be either a number or a numerical expression (see Numerical expressions below) operating on numbers, parameters—previously defined—, functions (see Functions below) and/or references (see References below). Neither strictly enumerative, nor strictly equational, it is possible to specify a range of numeric values using a minimum and maximum value enclosed in square brackets.

Examples:

yard: value?{nil, "default"}

rooms: <nrooms?>2, rooms>

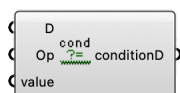
range: a?[0, 10]

### SGI Conditional Operators



The SGI Conditional Operators selector allows to select from a list of conditional Ops for shape descriptions.

### SGI Conditional Expression



The SGI Conditional Expression component composes a conditional expression as a concatenation of a parameter or function, a question mark ('?'), a conditional operator ('=', '<>', '<', '<=', '>', or '>=') and an argument. If the operator is specified as '[]', the operator is omitted and the argument, which must be a pair of numeric values, enclosed by square brackets. If the operator is specified as '{}', the operator is omitted and the argument, which must be a list of all numbers or all (double quoted) strings, enclosed by braces.

Inputs:

- *D*: Description part, either a parameter or function
- *Op*: conditional Op ('=', '<>', '<', '<=', '>', '>=', '[]', '{}')
- *value*: single number or numerical expression; a pair of numeric values; or a list of all numbers or all (double quoted) strings

Outputs:

- *conditionD*: conditional Description expression as text

### Numerical expressions

A numerical expression can be embedded in a parameter conditional (in the *lhs* of a description rule) or in the *rhs* of a description rule. A numerical expression can operate on literal keywords, numbers, numerical functions (see Functions below), parameters and references (see References below). Numerical expressions may include the operators plus ('+'), minus ('-'), times ('\*'), divided-by ('/'), modulo ('%') and to-the-power-of ('^'), with the usual operator precedence rules applying and the use of parentheses to override these rules where necessary. Other operations are available in the form of numerical functions.

Example ('vol', 'radius' and 'length' specify parameter references) :

volume: vol – pi^2 \* radius \* (length / 2)^2 + 4 / 3 \* pi \* (length / 2)^3

### String expressions

A string expression in the *lhs* of a description rule enables the identification of substrings in the matching process. Here, a string expression is a concatenation of literals and parameters (with or without conditional). A parameter can match any substring, conditioned by the literal components (and the conditional, if present). A concatenation of two parameters, without a literal separating the two parameters, would not be possible, unless the first parameter has an enumerative conditional.

A string expression in the *rhs* of a description rule can include literals, parameter references (see References below), numerical expressions (enclosed in parentheses) and functions returning either numbers or strings (see Functions below). The result is the concatenation of all components upon their evaluation into literal numbers or strings.

Examples (the two lines below may form the *lhs* and *rhs* of the same description rule):

be: be1 be20.“, ”.be21.“-rafter beam in front, ”.be22.“-rafter beam in back” “with ”.c?=(be21 + be22).“ columns”

be: be1 be20.“, ”.be21.“-rafter beam abutting ”.be22 “with ”.(c + 1).“ columns”

### Tuple expressions

Tuple expressions allow one to append or prepend an entity to a tuple, join two tuples or add two tuples. The operations to append, prepend and join all take the same format: two operands separated by a space. The appropriate interpretation is arrived at by looking at the structure of the two operands. If the entity shares a similar “structure” with the first element of the tuple, e.g., both are numbers or both are a tuple of similar structure, then the entity will be appended or prepended to the tuple depending on its position with respect to the tuple. If both operands are (nested) tuples, and the elements of both tuples have the same structure, then a join operation will be assumed, combining the elements from both tuples in a new, single tuple. If no structural similarity exists, then the expression will instead be interpreted as a tuple omitting enclosing brackets and separator.

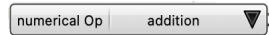
Adding two tuples adds the respective entities: if both entities are numbers they are summed; if both entities are strings they must be identical; if both entities are tuples and have the same structure, then addition is applied recursively.

Examples (the latter also includes a function):

position: a + (1, 0)

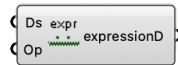
positions: a last(a) + (0, 1)

### SGI Numerical Operators



The SGI Numerical Operators selector allows to select from a list of numerical Ops for shape descriptions.

### SGI Expression



The SGI Expression component composes a string or numeric expression from a list of inputs, using the string concatenation operator (':') or a numerical operator, respectively.

Inputs:

- *Ds*: list of values to be concatenated in the expression (textual or numeric)
- *Op*: (numerical) Op; default is ':' for a string expression; '+', '-', '\*', '/', '%', '^' are accepted for a numeric expression

Outputs:

- *expressionD*: Description expression as text

### Functions

Functions allow for additional operations on numbers, texts/strings and tuples, or a combination thereof. A function returns a single value from any one of these three entity types. Strictly numerical functions include sqrt, sin, cos and tan, asin, acos and atan, taking a single number as argument and returning a number. Functions operating on texts/strings include determining the length of a string and determining a left and right substring, with the length of the substring specified as an additional argument to the function.

Functions operating on tuples include determining the length of a tuple, retrieving the first or last element of a tuple, or any element (item) by its index, the minimum (min) and maximum (max) value inside a tuple, retrieving a tuple of only unique elements, a tuple of pairs extracting consecutive elements pairwise from the operand tuple, a tuple of pairs (segments) such that the *i*th pair is made up of the *i*th and (*i*+1)th elements of the operand tuple, a tuple of tuples identifying loops in the operand tuple and a tuple of tuples representing an adjacencies matrix. The latter function takes two arguments, a tuple of 'enclosures' and a tuple of 'connecting' elements.

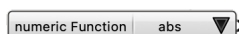
Tuples of numbers can be considered as vectors, currently only vectors of length two or three are considered. Functions on vectors require the different vectors to have the same length. These functions include determining the magnitude (mag) of a vector or the distance (also mag) or angle between two vectors, adding (vectoradd) or subtracting (vectorsubtract) two vectors, taking the dotproduct or crossproduct of two vectors or scaling a vector by a number (vectorscale).

Finally, a function to generate a random number takes as input (a tuple of) two or three numbers, with the first two specifying the range and the optional third one the step. More information on functions is provided in Appendix B. Description functions.

Examples:

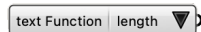
positions: a (random(0,10,1), 0)

### SGI Numeric Functions



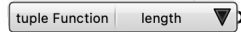
The SGI Numeric Functions selector allows to select from a list of numeric Functions for shape descriptions.

#### SGI Text Functions



The SGI Text Functions selector allows to select from a list of text Functions for shape descriptions.

#### SGI Tuple Functions



The SGI Tuple Functions selector allows to select from a list of tuple Functions (omitting vector functions) for shape descriptions.

#### SGI Vector Functions



The SGI Vector Functions selector allows to select from a list of vector Functions for shape descriptions.

#### SGI Function Concat



The SGI Function Concat component returns a functional description expression that is a concatenation of a function and, within parentheses, its arguments (see Appendix B. Description functions).

Inputs:

- *Function*: description Function
- *Ds*: one or more Description arguments to the function, each argument either a number, a text, a description or a tuple of these

Outputs:

- *functionD*: functional Description expression as a concatenation of the function and, within parentheses, its arguments

#### References

We distinguish three kinds of references. Firstly, parameter references are variable terms in the *rhs* of a description rule, which reference variable terms (parameters) in the *lhs* of the same (or another) description rule. The value of the parameter reference in the *rhs* is the value of the same parameter in the *lhs* upon the matching of the *lhs*.

Secondly, a description reference is similar to a parameter reference but references a variable term in another shape description (that is part of the same rule). In such case, the parameter must be preceded by the description type in order to identify the appropriate description and parameter. Alternatively, rather than referencing a specific parameter, the entire value of the description can be referenced using the term *value*. The same applies to descriptions used within a color attribute specification within a shape rule.

Finally, a shape reference similarly references data from the shape rule component of the rule. A shape reference may take one of two forms. Firstly, shape elements can be referenced by the element type (see Shape element types below); however, referencing a unique element will only work if there is only one element of the specific type, otherwise the reference will be ambiguous. Otherwise, the element can be disambiguated by additionally specifying its attribute label (or description), provided the element has an attribute and the attribute label is unique (see example below). Secondly, spatial elements can be tagged in the shape rule. Spatial element tags can be understood as attributes to the elements, similar



to labels (tags are recognized by the '#' symbol preceding the tag identifier). However, different from attributes, tags are particular to the rule in question and only subsist within the rule matching and application process of this rule. As such, tags are not considered attributes; within a shape description, the tag solely serves to identify the spatial element the description is referencing.

Example querying the positions of two points with given labels:

constraint: `a?>=mag(point3D.value:plabelD.value="1", point3D.value:plabelD.value="2")`

constraint: `a?>=mag(#pt1.value, #pt2.value)`

### SGI Type Properties



The SGI Type Properties component retrieves the list of property names for a spatial type (see Shape element types and their available properties below), and returns each name concatenated to the appropriate element type or, if specified, an element tag. This combination can be used in a description rule to retrieve the property value. The element type is dependent on the spatial type and whether it applies to a non-parametric (default) or parametric-associative rule. It can be additionally specified to apply to an element from the left-hand-side (default) or right-hand-side of the rule.

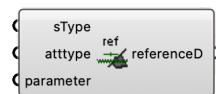
Inputs:

- *spatialType*: spatialType ('point', 'line segment', 'plane segment', 'circle', 'ellipse', 'circular arc' or 'quadratic Bezier')
- *for\_pRule*: Boolean value specifying whether the element type refers to a parametric-associative (True) or non-parametric rule (False, default)
- *of\_rhShape*: Boolean value specifying whether the element will be part of the right-hand-side (True) or left-hand-side (False, default) of the rule
- *Tag*: optional element Tag

Outputs:

- *propertyDs*: list of shape properties as Descriptions, in the form of concatenations of element Tag/Type and property names for a spatialType

### SGI Description Reference



The SGI Description Reference component returns a description reference expression that is a concatenation of a shape (attribute) description type (or attribute type, e.g., 'color') and the specified parameter or, otherwise, the term 'value'. The resulting expression references the parameter (if specified) or value of a shape description or shape (or color) attribute description.

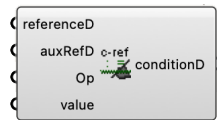
Inputs:

- *sType*: either a description Type or spatialType ('point', 'line segment', 'plane segment', 'circle', 'ellipse', 'circular arc' or 'quadratic Bezier')
- *atttype*: optional atttype ('labelD' or 'color')
- *parameter*: optional parameter

Outputs:

- *referenceD*: Description expression referencing the parameter (if specified) or value of a shape description or shape (or color) attribute description

## SGI Conditional Reference



The SGI Conditional Reference component composes a conditional reference as a concatenation of a (main) description reference or shape property, a colon (':'), a (auxiliary) description reference or shape property, a conditional operator ('=', '<>', '<', '<=', '>', or '>=') and an argument. The argument should either be a number, a vector or a string. The resulting expression constrains the main reference or property by its relation to the auxiliary shape (attribute) description or spatial type and the condition on the (parameter) value of this auxiliary reference. For example, in the case of multiple line segments, the property value of a specific line segment can be identified based on the value of its shape attribute description by specifying a condition on the shape attribute description.

Inputs:

- *referenceD*: main Description reference or shape property
- *auxRefD*: auxiliary Description reference or shape property
- *Op*: conditional Op ('=', '<>', '<', '<=', '>', or '>=', '[ ]', '{ }')
- *value*: numeric, vector or quoted string value

Outputs:

- *conditionD*: conditional Description reference

## Shape element types and their available properties

Every spatial type, except for circular arcs, is identified by two names. The first one should be used within non-parametric rules and the second within parametric-associative rules (pRule). Note that circular arcs are not yet available within parametric-associative rules and, if specified, will be ignored.

type	name	property	output	value
points	point3D	value	vector tuple*	position
	pointP3D			
line segments	lineSeg3D	root	vector tuple*	root point (nearest point to the origin)
	lineSegP3D	direction unitDir start end midpoint length squareLength  oDirection oStart oEnd	vector tuple* vector tuple* vector tuple* vector tuple* vector tuple* number number  vector tuple* vector tuple* vector tuple*	direction vector unit direction vector start point endpoint midpoint line length square value of line length  original direction vector† original start point† original endpoint†
plane segments	planeSeg3D	normal	vector tuple*	normal vector

	planesegP3D	area outer	number tuple of vector tuples*	plane area list of outer boundary vertices
circles	circle3D	normal center	vector tuple*	plane normal vector
	circleP3D	radius diameter circumference area	vector tuple* number number number number	center point radius diameter circumference area of the circle
ellipses	ellipse3D	normal center	vector tuple*	plane normal vector
	ellipseP3D	foci  radii  area	vector tuple* tuple of vector tuples* tuple of numbers  number	center point list of focal points  list of longer and shorter radii area of the ellipse
circular arcs	arc3D	normal center radius diameter circumference start end length angle  area	vector tuple* vector tuple* number number number vector tuple* vector tuple* number number number	plane normal vector circle center point circle radius circle diameter circle circumference endpoint (ccw) endpoint (cw) arc length angle covered by the arc (in radians) area covered by the arc
quadratic Bezier curves	bezier3D	normal start	vector tuple*	plane normal vector
	bezierP3D	controlPoint end vertex	vector tuple* vector tuple* vector tuple* vector tuple*	1st control point 2nd control point 3rd control point maximum or minimum of the curve
labels/ descriptions as spatial element attribute  for points, line segments, plane segments, circles, ellipses, circular arcs and quadratic Bezier curves, respectively	pLabelD	value	string	label or description string
	lLabelD			
	pLabelD			
	cLabelD			
	eLabelD			
	aLabelD			
	bLabelD			

colors as spatial element attribute	lColor	depending on colorMode: if 'graytone': value	depending on colorMode: if 'graytone': number	depending on colorMode: if 'graytone': between 0 (white) and 255 (black) else: R, G, B and alpha values hexadecimal string in format '0xRRGGBB'
	plColor			
	cColor			
	eColor	else: value  RGB	else: tuple of integers  string	
	aColor			
	bColor			

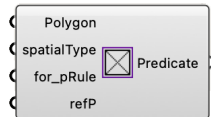
\*A vector tuple is a tuple of two or three numbers.

†The direction vector of a line segment is a normalized vector: it has a positive X-coordinate (or a positive Y-coordinate if the X-coordinate is 0; or a positive Z-coordinate if both the X and Y-coordinates are 0). The start and end points of a line segment are the respective endpoints of the line segment with the normalized direction vector. The original direction vector, and original start and end points, reflect the situation before normalization, that is, how the line segment was created. Note that the original direction vector loses its meaning when line segments are operated upon (e.g., combined with another line segment).

## 12. Specifying predicates

A predicate serves to express a special condition on the application of a rule. Such condition cannot simply be explicated within the left-hand-side shape. As an example, a predicate may specify that a polygonal area must be devoid of any spatial elements. Most predicates are only applicable to parametric-associative rules, however, a few predicates are also applicable to non-parametric rules. These are, specifically, the void, inside and outside predicates.

### SGI Void Predicate



The SGI Void Predicate component creates a void predicate from one or more polygonal geometries and, optional, spatial types, and an optional reference point. It is applicable to both non-parametric and parametric-associative rules. The void predicate stipulates that a given polygonal area is to contain no spatial elements (points, line segments, plane segments) at all or of the specified type; spatial elements may coincide with the boundary. It must be noted that while the predicate explicates the vertices by their coordinates, in the case of a parametric-associative rule, they must necessarily coincide with any of the line segments in the *lhs* shape in order for the vertices to be recognized via the parametric-associative matching mechanism.

If the numbers of inputs are the same, it is assumed they correspond; otherwise, all spatial types specified are considered for each geometry, unless they come in the form of a list of lists. In the latter case, surplus spatial type inputs are ignored.

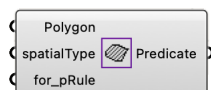
Inputs:

- *Polygon*: polygonal geometry (one or more); may be expressed as points, line segments, closed polyline, flat surface or boundary representation
- *spatialType*: optional list of spatialTypes ('point', 'line segment', 'plane segment', 'circle', 'ellipse', 'circular arc' or 'quadratic Bezier')
- *for\_pRule*: optional Boolean value specifying whether the element type(s) refers to a parametric-associative (True) or non-parametric rule (False, default); only considered if a *spatialType* is specified
- *refP*: optional reference point; if specified, the geometry will be considered moved from the reference point to the origin (assuming the same reference point is used to similarly move the left-hand-side shape of the rule)

Outputs:

- *Predicate*: Predicate text

### SGI Inside Predicate



The SGI Inside Predicate component creates an inside predicate from one or more polygonal geometries and, optional, spatial types. It is applicable to both non-parametric and parametric-associative rules. The inside predicate stipulates that all spatial elements (points, line segments, plane segments) of the specified type (if specified) matching (part of) the *lhs* shape are to be entirely contained within the given polygonal area; although, spatial elements may touch or coincide with the boundary. It must be noted that, unlike the void predicate, the coordinates of the vertices are taken at absolute value and not affected by any transformation as resulting from the matching.

If the numbers of inputs are the same, it is assumed they correspond; otherwise, all spatial types specified are considered for each geometry, unless they come in the form of a list of lists. In the latter case, surplus spatial type inputs are ignored

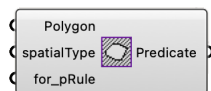
Inputs:

- *Polygon*: polygonal geometry; may be expressed as points, line segments, closed polyline, flat surface or boundary representation
- *spatialType*: optional list of spatialTypes ('point', 'line segment', 'plane segment', 'circle', 'ellipse', 'circular arc' or 'quadratic Bezier')
- *for\_pRule*: optional Boolean value specifying whether the element type refers to a parametric-associative (True) or non-parametric rule (False, default); only considered if a *spatialType* is specified

Outputs:

- *Predicate*: Predicate text

#### SGI Outside Predicate



The SGI Outside Predicate component creates an outside predicate from one or more polygonal geometries and, optional, spatial types. It is applicable to both non-parametric and parametric-associative rules. The outside predicate stipulates that all spatial elements (points, line segments, plane segments) of the specified type (if specified) matching (part of) the *lhs* shape are to be entirely outside of the given polygonal area; although, spatial elements may touch or coincide with the boundary. It must be noted that, unlike the void predicate, the coordinates of the vertices are taken at absolute value and not affected by any transformation as resulting from the matching.

If the numbers of inputs are the same, it is assumed they correspond; otherwise, all spatial types specified are considered for each geometry, unless they come in the form of a list of lists. In the latter case, surplus spatial type inputs are ignored

Inputs:

- *Polygon*: polygonal geometry; may be expressed as points, line segments, closed polyline, flat surface or boundary representation
- *spatialType*: optional list of spatialTypes ('point', 'line segment', 'plane segment', 'circle', 'ellipse', 'circular arc' or 'quadratic Bezier')
- *for\_pRule*: optional Boolean value specifying whether the element type refers to a parametric-associative (True) or non-parametric rule (False, default); only considered if a *spatialType* is specified

Outputs:

- *Predicate*: Predicate text

#### SGI Maxline Predicate



The SGI Maxline Predicate component creates a maxline predicate from one or more line element tags. It is only applicable to parametric-associative rules. The maxline predicate stipulates that any line segment matching the tagged line segment must use its full extent to match the line segment.

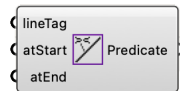
Inputs:

- *lineTag*: one or more element Tags of line segments

Outputs:

- *Predicate*: Predicate text

### SGI Bound Predicate



The SGI Bound Predicate component creates a bound predicate from one or more line element tags and Boolean values specifying whether the respective endpoint of the line element must be a boundary point or not. It is only applicable to parametric-associative rules. The bound predicate stipulates a matching line to be bound at an indicated endpoint. It is similar to maxline but is able to limit the line from a specific endpoint.

Any surplus Boolean values are ignored, any missing values are considered false; unless only a single value is specified, in which case it is copied. Note that the endpoints of the tagged line segment will initially be ordered as identified when constructing the line segment, but this may change upon manipulating the segment (e.g., through rule application), after which the endpoints would be ordered corresponding their coordinates (first X, then Y and finally Z).

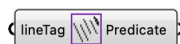
Inputs:

- *lineTag*: one or more element Tags of line segments
- *atStart*: one or more Boolean values specifying whether the startpoint of the (respective) line segments must be a boundary point or not
- *atEnd*: one or more Boolean values specifying whether the endpoint of the (respective) line segments must be a boundary point or not

Outputs:

- *Predicate*: Predicate text

### SGI Shortest-Line Predicate



The SGI Shortest-Line Predicate component creates a shortest line predicate from one or more line element tags. It is only applicable to parametric-associative rules. The shortest line predicate stipulates that the line segment matching the tagged line must be the shortest line in the matching shape. In the case of multiple inputs, the matched lines identified as the shortest lines must all have the same length.

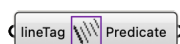
Inputs:

- *lineTag*: one or more element Tags of line segments

Outputs:

- *Predicate*: Predicate text

### SGI Longest-Line Predicate



The SGI Longest-Line Predicate component creates a longest line predicate from one or more line element tags. It is only applicable to parametric-associative rules. The longest line predicate stipulates that the line segment matching the tagged line must be the longest line in the matching shape. In the case of multiple inputs, the matched lines identified as the longest lines must all have the same length.

Inputs:

- *lineTag*: one or more element Tags of line segments

Outputs:

- *Predicate*: Predicate text

## SGI Description Predicate



The SGI Description Predicate component creates a description predicate from one or more conditional description expressions (see section 11. Specifying shape descriptions). As the conditional expression may reference one or more spatial element properties, the description predicate may stipulate a constraint over such properties. The description predicate is only applicable to parametric-associative rules.

Inputs:

- *conditionD*: one or more conditional expression Descriptions

Outputs:

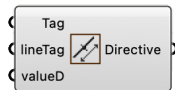
- *Predicate*: Predicate text



### 13. Specifying directives

Directives are value specifications for applying a parametric-associative rule that cannot be derived from or expressed within the right-hand-side shape of the rule. As an example, a directive may specify the distance from a new line added in the *rhs* to an existing point. Directives are only applicable to parametric-associative rules.

#### SGI Point-on-Line Directive



The SGI Point-on-Line Directive component creates a point on line directive from one or more target and line element tags and parameter values. Any discrepancy between the numbers of inputs is resolved by copying the respective last value.

The point on line directive specifies the parameter value for the position of a new point on an existing line segment, with respect to the endpoints of the line with respective parameter values 0 and 1. The new point may serve as the endpoint of a new (target) line segment. The parameter value can be explicated as a numeric value between 0 and 1 or as a description enclosed within backward quotes. For example, the description ``random((0.3, 0.7))`` prescribes a random value between 0.3 and 0.7.

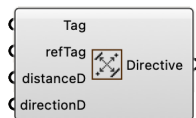
Inputs:

- *Tag*: one or more element Tags of line segments or points (*rhs* shape)
- *lineTag*: one or more element Tags of line segments (*lhs* (or *rhs*) shape)
- *valueD*: one or more parameter values, each either a numeric value or a Description (enclosed within backward quotes)

Outputs:

- *Directive*: Directive text

#### SGI Distance Directive



The SGI Distance Directive component creates a distance directive from one or more target and reference element tags and distance values, and an optional direction vector. Any discrepancy between the numbers of inputs is resolved by copying the respective last value. The distance directive specifies the distance from a new spatial element (line or point) to an existing spatial element (line or point). There are 4 possible cases:

- *Line-line distance*: the new line must be parallel to the existing line; a direction vector can be additionally specified to indicate the direction in which the line is added. The direction vector can be explicated as a coordinate tuple or as a description enclosed within backward quotes. For example, the description ``#plane.normal`` prescribes the normal vector of a tagged plane as the direction vector.
- *Line-point distance*: the new line must run through an existing point, line endpoint or line intersection point; the distance is measured perpendicular from the line to the point
- *Point-line distance*: the new point must be on another existing line not parallel to the reference line; the distance is measured perpendicular from the line to the point
- *Point-point distance*: the new point must be on an existing line; the distance is measured between both points

Inputs:

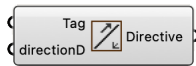
- *Tag*: one or more element Tags of target points or line segments (*rhs* shape)

- *refTag*: one or more element Tags of reference points or line segments (*lhs* or *rhs* shape)
- *distanceD*: one or more distance values, each either a numeric value or a Description (enclosed within backward quotes)
- *directionD*: optional, one or more direction vectors, each either a vector, a coordinate tuple or a Description (enclosed within backward quotes)

Outputs:

- *Directive*: Directive text

#### SGI Direction Directive



The SGI Direction Directive component creates a direction directive from one or more line element tags and direction vectors. Any discrepancy between the numbers of inputs is resolved by copying the respective last value.

The direction directive specifies the direction vector of a new line element. The direction vector can be explicated as a coordinate tuple or as a description enclosed within backward quotes. For example, the description ``#plane.normal`` prescribes the normal vector of a tagged plane as the direction vector.

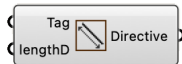
Inputs:

- *Tag*: one or more element Tags of line segments (*rhs* shape)
- *directionD*: one or more direction vectors, each either a coordinate tuple or a Description (enclosed within backward quotes)

Outputs:

- *Directive*: Directive text

#### SGI Length Directive



The SGI Length Directive component creates a length directive from one or more line element tags and length values. Any discrepancy between the numbers of inputs is resolved by copying the respective last value.

The length directive specifies the length of a new line segment.

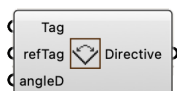
Inputs:

- *Tag*: one or more element Tags of line segments (*rhs* shape)
- *lengthD*: one or more length values, each either a numeric value or a Description (enclosed within backward quotes)

Outputs:

- *Directive*: Directive text

#### SGI Angle Directive



The SGI Angle Directive component creates an angle directive text from one or more target and reference element tags and angle values. Any discrepancy between the numbers of inputs is resolved by copying the respective last value.

The angle directive specifies the angle (in radians) between a new spatial line element and an existing spatial line element.

Inputs:

- *Tag*: one or more element Tags of target line segments (*rhs* shape)

- *refTag*: one or more element Tags of reference line segments (*lhs* or *rhs* shape)
- *angleD*: one or more angle values (expressed in radians), each either a numeric value or a Description (enclosed within backward quotes)

Outputs:

- *Directive*: Directive text

## Appendix A. A formal notation for shape descriptions

The table below presents a formal notation for shape descriptions and the left-hand-side (*lhs*) and right-hand-side (*rhs*) of shape description rules in Extended Backus-Naur-Form (EBNF), including examples. The same non-terminals serve to define the production rules for a description, an *lhs* and an *rhs*. Only when necessary are alternative production rules defined for the same non-terminal; these are then identified by adding the terms *description*, *lhs* and *rhs*, respectively, enclosed within angle brackets ('<...>'), as a prefix to the respective production rule.

<pre>typed-description = type-name ':' description . type-name = identifier . description = description-entity   description-sequence . description-entity = literal   top-level-tuple . description-sequence = '&amp;' description-entity '&amp;' { description-entity '&amp;' } .</pre>
<pre>literal = keyword-literal   number   string . keyword-literal = 'e'   'nil'   'pi'   'true'   'false'. number = [ '-' ] digit-sequence [ '.' digit-sequence ] . digit-sequence = digit { digit } . digit = '0'   '1'   '2'   '3'   '4'   '5'   '6'   '7'   '8'   '9' . string = '"' { string-character } '"' . string-character = any-character-except-quote   '\ ' .</pre>
<p>Example <b>description-entity</b>: "centrally divided, double 1-rafter beam in front and back"</p> <p>Example <b>description-sequence</b>: &amp;e&amp;0&amp;"nothing"&amp;</p>
<pre>top-level-tuple = tuple   unmarked-tuple . tuple = '(' tuple-entities ')'   '&lt;' [ tuple-entities ] '&gt;'   '[' [ tuple-entities ] ']' . &lt;description&gt;tuple-entities = tuple-entity-sequence . &lt;lhs&gt;tuple-entities = tuple-entity-sequence   tuple-expression . &lt;rhs&gt;tuple-entities = tuple-entity-sequence   tuple-expression . tuple-entity-sequence = tuple-entity ( { ',' tuple-entity }   { ';' tuple-entity } ) . &lt;description&gt;tuple-entity = literal   tuple . &lt;lhs&gt;tuple-entity = numeric-expression   string-expression   tuple . &lt;rhs&gt;tuple-entity = numeric-expression   string-expression   tuple   function-returns-tuple . unmarked-tuple = tuple-expression   tuple ( tuple   keyword-literal ) { tuple-entity } .</pre>
<p>Example <b>tuple</b>: ("l:", 10, "c:", (0, 0), "r:", 0)</p> <p>Example <b>unmarked-tuple</b>: &lt;"", "O", "R0", "R1"&gt; &lt;"O", 1, 1, 1&gt; &lt;"R0", 1, 1, 0&gt; &lt;"R1", 1, 0, 1&gt;</p>
<pre>description-rule-side = description-rule-entity   description-rule-sequence . &lt;lhs&gt;description-rule-entity = literal   parameter [ '?' conditional ]   string-expression   top-level-tuple . &lt;rhs&gt;description-rule-entity = numeric-expression   string-expression   function-returns-tuple   tuple-expression . description-rule-sequence = '&amp;' description-rule-entity '&amp;' { description-rule-entity '&amp;' } .</pre>

parameter = identifier .  
 identifier = ( letter | underscore ) { ( letter | underscore | digit ) } .  
 letter = 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' | 'J' | 'K' | 'L' | 'M' | 'N' | 'O' | 'P' | 'Q' | 'R' | 'S' | 'T' | 'U' | 'V' | 'W' | 'X' | 'Y' | 'Z' | 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j' | 'k' | 'l' | 'm' | 'n' | 'o' | 'p' | 'q' | 'r' | 's' | 't' | 'u' | 'v' | 'w' | 'x' | 'y' | 'z' .  
 underscore = '\_' .

Example **<lhs>description-rule-entity**:

<"Fixed", var1> <var2, var3> remainder

Example **description-rule-sequence**:

&a1&a2&a3&a4&a5&a6&a7&a8&

conditional = enumeration | equation | range.  
 enumeration = '{' ( number-sequence | string-sequence ) '}' .  
 number-sequence = number { ',' number } .  
 string-sequence = string { ',' string } .  
 equation = comparator comparand .  
 comparator = '=' | '<' | '<=' | '>' | '>=' .  
 comparand = number | '(' numeric-expression ')' | parameter | reference .  
 range = '[' number ',' number ']' .

Example **<lhs>description-rule-entity with enumeration**:

yard?{nil, "default"}

Example **<lhs>description-rule-entity with equation**:

<nrooms?>2, rooms>

numeric-expression = term { addition-operator term } .  
 term = factor { multiplication-operator factor } .  
 factor = base { exponentiation-operator exponent } .  
 exponent = base .  
 base = keyword-literal | number | '(' numeric-expression ')' | function-returns-number | parameter | reference .  
 exponentiation-operator = '^' .  
 multiplication-operator = '\*' | '/' | '%' .  
 addition-operator = '+' | '-' .

Example **numeric-expression**:

vol – pi^2 \* radius \* (length / 2)^2 + 4 / 3 \* pi \* (length / 2)^3

string-expression = string-expression-entity { ',' string-expression-entity } .  
 <lhs>string-expression-entity = literal | parameter [ '?' conditional ] .  
 <rhs>string-expression-entity = base | string | function-returns-string .

Example **<rhs>string-expression**:

"with ".(c + 1). " columns"

Example **<lhs>string-expression**:

"with ".c?=(be21 + be22). " columns"

<lhs>tuple-expression = tuple-append | tuple-prepend .  
 <rhs>tuple-expression = tuple-addition | tuple-extension .  
 tuple-append = { tuple-entity } parameter ( '\*' | '+' ) tuple-entity { tuple-entity } [ tuple-expression ] .  
 tuple-prepend = [ tuple-expression ] { tuple-entity } tuple-entity parameter ( '\*' | '+' ) { tuple-entity } .  
 tuple-addition = [ parameter ] '+' basic-tuple-argument .  
 tuple-extension = { tuple-entity } parameter { tuple-entity } [ tuple-expression ] .

Example **tuple-prepend**:

h1 h2 H\*

Example **tuple-extension**:

a1 last(a1) + (0, 1)

Example **tuple-addition**:

bedrooms + <1, [(“couple”, 0), (“double”, 0), (“single”, 1)]>

function = function-returns-number | function-returns-string | function-returns-tuple .  
 function-returns-number = numeric-function | length-function | string-function-untyped | tuple-  
 function-untyped | vector-function | round-function | random-function .  
 numeric-function = ( 'sqrt' | 'sin' | 'cos' | 'tan' | 'asin' | 'acos' | 'atan' ) ( ' numeric-expression ' ) |  
 'atan2' ( ' numeric-expression ' ; numeric-expression ' ) .  
 length-function = 'length' ( ' ( string-argument | tuple-argument ) ' ) .  
 <lhs>string-argument = string | function-returns-string | parameter | reference .  
 <rhs>string-argument = string-expression .  
 function-returns-string = string-function-returns-string | string-function-untyped | tuple-function-  
 untyped .  
 string-function-returns-string = ( 'left' | 'right' ) ( ' string-argument ' ; numeric-expression ' ) .  
 string-function-untyped = 'eval' ( ' string-argument ' ) .  
 tuple-function-untyped = ( 'first' | 'last' | 'min' | 'max' ) ( ' tuple-argument ' ) | ( 'item' ) ( ' tuple-  
 argument ' ; numeric-expression ' ) .  
 <lhs>tuple-argument = basic-tuple-argument .  
 <rhs>tuple-argument = basic-tuple-argument | tuple-expression .  
 basic-tuple-argument = tuple | function-returns-tuple | parameter | reference .  
 function-returns-tuple = tuple-function-returns-tuple | function-returns-vector | string-function-  
 untyped | tuple-function-untyped .  
 tuple-function-returns-tuple = ( 'unique' | 'segments' | 'pairwise' | 'loops' ) ( ' tuple-argument ' ) |  
 'adjacencies' ( ' tuple-argument ' ; tuple-argument ' ) .  
 function-returns-vector = two-vector-function | proj-vector-function | scale-vector-function |  
 round-function .  
 two-vector-function = ( 'vectoradd' | 'vectorsubtract' | 'dotproduct' | 'crossproduct' ) ( ' ( vector-  
 argument ' ; vector-argument | two-vector-argument ) ' ) .  
 vector-argument = ( ' numeric-expression ' ; numeric-expression [ ' ; numeric-expression ] ' ) |  
 function-returns-vector | parameter | reference .  
 two-vector-argument = ( ' vector-argument ' ; vector-argument ' ) | parameter | reference .  
 proj-vector-function = 'proj' ( ' ( vector-argument ' ; vector-argument ' ; vector-argument | three-  
 vector-argument ) ' ) .  
 three-vector-argument = ( ' vector-argument ' ; vector-argument ' ; vector-argument ' ) | parameter  
 | reference .  
 scale-vector-function = 'vectorscale' ( ' ( vector-argument ' ; numeric-expression | vector-number-  
 argument ) ' ) .  
 vector-number-argument = ( ' vector-argument ' ; numeric-expression ' ) | parameter | reference .  
 vector-function = ( 'mag' | 'angle' ) ( ' ( vector-argument ' ; vector-argument ' ) | ' ( two-vector-  
 argument ' ) ' ) .  
 round-function = 'round' ( ' ( numeric-expression | vector-argument ' ) ' ) .  
 random-function = 'random' ( ' vector-argument ' ) .

Example **function-returns-number**:

length("room")

Example **function-returns-tuple**:

adjacencies(a4, a5 a6)

reference = reference-to-lhs | reference-to-rhs .  
 reference-to-lhs = [ 'lhs.' ] reference-designator ' ( 'value' | parameter | property ) [ ':' filter ] .  
 reference-to-rhs = 'rhs.' reference-designator ' property [ ':' filter ] .  
 reference-designator = identifier .  
 property = identifier .  
 filter = reference-designator ' property filter-operator ( number | vector | string ) .  
 filter-operator = '=' | '<>' | '<=' | '>=' .  
 vector = [ rational ] '(' rational ' ' rational ' ' rational ' ' rational ' ' .  
 rational = [ '-' ] digit-sequence [ '/' digit-sequence ] .

Example **reference-to-lhs**:

indices.value

Example **reference-to-rhs**:

rhs.sections.radius:labels.label="S"



## Appendix B. Description functions

### Numerical functions

function	input	output
abs	1 number	The absolute value of the number
sqrt	1 number	The square root of the number
sin	1 number	The sine value of the angle (in radians)
cos	1 number	The cosine value of the angle (in radians)
tan	1 number	The tangent value of the angle (in radians)
asin	1 number	The inverse sine of the number (in radians)
acos	1 number	The inverse cosine of the number (in radians)
atan*	1 number	The inverse tangent of the number (in radians)
atan2*	2 numbers	The inverse tangent of the ratio (in radians)
todegree	1 number	The value converted from radians in degrees
toradian	1 number	The value converted from degrees in radians
round	1 number	The value rounded to the nearest integer

\*atan versus atan2:

- atan takes 1 input and returns a result from quadrants 1 and 4
- atan2 takes 2 inputs (u, v) that specify a ratio u/v and returns a result from all quadrants

For example:

u	v	$x = u/v$	atan(x)	atan2(u,v)
2	1	2	1.1071487177940904	1.1071487177940904
-2	1	-2	-1.1071487177940904	-1.1071487177940904
2	-1	-2	-1.1071487177940904	2.0344439357957027
-2	-1	2	1.1071487177940904	- 2.0344439357957027

### Text functions

function	input	output
length	1 string	The length of the string
left	1 string and 1 number	The left substring of the specified length
right	1 string and 1 number	The right substring of the specified length

## Tuple functions

function	input	output
length	1 tuple	The number of elements in the tuple
first	1 tuple	The first element of the tuple
last	1 tuple	The last element of the tuple
item	1 tuple and 1 number	The indexed element of the tuple
min	1 tuple	The element of the tuple with minimum value
max	1 tuple	The element of the tuple with maximum value
unique	1 tuple	A tuple of only unique elements
pairwise	1 tuple	A tuple of pairs extracting consecutive elements pairwise from the operand tuple; e.g., (a, b, c, d) -> ((a, b), (c, d))
segments	1 tuple	A tuple of overlapping pairs extracting consecutive elements from the operand tuple; e.g., (a, b, c, d) -> ((a, b), (b, c), (c, d))
loops	1 tuple	A tuple of tuples identifying loops in the operand tuple; e.g., (a, b, c, d, a, e, f, c) -> ((a, b, c, d), (c, d, a, e, f))
adjacencies	2 tuples: a tuple of "enclosures" and a tuple of "connecting" elements	A tuple of tuples representing an adjacency matrix
random	1 tuple: either 2 or 3 numbers	A random number within the range specified by the first two operands; the optional third operand is considered as a step value for the random number generation

## Vector (tuple) functions

function	input	output
round	1 vector tuple*	A vector tuple with each value rounded to the nearest integer
mag	1 or 2 vector tuples*	The distance between the two vectors or the magnitude or length of a single vector
angle	2 vector tuples*	The angle between the two vectors (counterclockwise angle from the first to the second vector) (in radians)
proj	3 vector tuples*: a direction vector, a root vector and a position vector	A vector tuple representing the projection of the position vector on the line specified by the direction vector and root vector
vectoradd	2 vector tuples*	A vector tuple representing the sum of the two vectors
vectorsubtract	2 vector tuples*	A vector tuple representing the difference of the two vectors
vectorscale	1 vector tuple* and 1 number	A vector tuple representing the product of the vector and the scalar
dotproduct	2 vector tuples*	The number resulting from the dot product of the two vectors
crossproduct	2 vector tuples*	A vector tuple representing the cross product of the two vectors

\*A vector tuple is a tuple of two or three numbers; any function accepting (one or more) vector tuples will also accept a single tuple collecting all operands

## Appendix C: A formal notation for flow descriptions

We adapt the notation for regular expressions as a formal notation for flow descriptions. Regular expressions are patterns that are used to match strings by string searching algorithms. Regular expressions are composed of tokens that are combined in a prescribed order, with some variation built into the expression, in order to match a goal string. Similarly, flows are composed of shape or compound rules that are combined in a prescribed order, with some algorithmic variation, in order to produce a valid final shape.

Within the table below we use the term *sub-flow* to denote each and every element within a flow or sub-flow. That is, a sub-flow may be a rule, a flow or a flow structure (sequence, disjunction or negation). A rule is represented by its name, so is a flow. A flow structure is represented either as a sequence of sub-flows within parentheses, as a disjunction of sub-flows within square brackets, or as a sub-flow preceded by the negation symbol '!'.

Metacharacter	Explanation
	A space separates two sub-flows in a sequence or disjunction. In a sequence, if either sub-flow fails to apply, the entire sequence fails to apply. In a disjunction, only one sub-flow needs to succeed for the disjunction to succeed.
(...)	Parentheses enclose a sequence of sub-flows. Sub-flows are attempted to be applied one after the other, each time on the result of the previous application, in the order specified. If one of the sub-flows fails, backtracking will occur.
[...]	Square brackets enclose a disjunction (selection) of alternative sub-flows. Alternatives are attempted to be applied in the order specified. As soon as one application succeeds, subsequent sub-flows are skipped. If no alternative applies, backtracking will occur.
[*...]	Square brackets enclose a disjunction (selection) of alternative sub-flows. When the first character within square brackets is an asterisk, the alternatives are attempted to be applied in a random order instead of in the order specified. As soon as one application succeeds, subsequent sub-flows are skipped. If no alternative applies, backtracking will occur.
!	Success and failure of the succeeding sub-flow are inverted. If the sub-flow fails, the application succeeds, whereas if the sub-flow succeeds, backtracking occurs.
?	The preceding sub-flow may apply zero or one time. A single application is attempted. Success or failure, no backtracking occurs, unless backtracking arrives from a later point to this sub-flow and all alternatives within this sub-flow have been exhausted.
*	The preceding sub-flow may apply zero, one or more times. The iteration proceeds until the sub-flow fails to apply. No backtracking occurs, unless backtracking arrives from a later point to this sub-flow and all alternatives within this iterative sub-flow have been exhausted.

+	The preceding sub-flow may apply one or more times. The iteration proceeds until the sub-flow fails to apply. Backtracking only occurs if the sub-flow fails at the very first time, unless backtracking arrives from a later point to this sub-flow and all alternatives within this iterative sub-flow have been exhausted.
{ <i>n</i> }	The preceding sub-flow may apply exactly <i>n</i> times. The iteration proceeds until <i>n</i> applications or until the sub-flow fails to apply. Backtracking occurs if fewer than <i>n</i> applications succeed or, upon backtracking from a later point to this sub-flow, if all alternatives within this iterative sub-flow have been exhausted.
{ <i>n</i> ,}	The preceding sub-flow may apply <i>n</i> or more times. The iteration proceeds until the sub-flow fails to apply. Backtracking occurs if fewer than <i>n</i> applications succeed or, upon backtracking from a later point to this sub-flow, if all alternatives within this iterative sub-flow have been exhausted.
{ <i>n,m</i> }	The preceding sub-flow may apply any number of times between <i>n</i> and <i>m</i> . The iteration proceeds until <i>m</i> consecutive applications or until the sub-flow fails to apply. Backtracking occurs if fewer than <i>n</i> applications succeed or, upon backtracking from a later point to this sub-flow, if all alternatives within this iterative sub-flow have been exhausted.
?+	The preceding sub-flow may apply zero or one time. No backtracking occurs. When backtracking arrives from a later point to this sub-flow, rather than backtracking within the sub-flow or its iteration, backjumping takes place to the point before this sub-flow.
*+	The preceding sub-flow may apply zero, one or more times. The iteration proceeds until the sub-flow fails to apply. No backtracking occurs. When backtracking arrives from a later point to this sub-flow, rather than backtracking within the sub-flow or its iteration, backjumping takes place to the point before this sub-flow.
++	The preceding sub-flow may apply one or more times. The iteration proceeds until the sub-flow fails to apply. Backtracking only occurs if the sub-flow fails at the very first time. When backtracking arrives from a later point to this sub-flow, rather than backtracking within the sub-flow or its iteration, backjumping takes place to the point before this sub-flow.
{ <i>n</i> }+	The preceding sub-flow may apply exactly <i>n</i> times. The iteration proceeds until <i>n</i> applications or until the sub-flow fails to apply. Backtracking occurs if fewer than <i>n</i> applications succeed. When backtracking arrives from a later point to this sub-flow, rather than backtracking within the sub-flow or its iteration, backjumping takes place to the point before this sub-flow.
{ <i>n</i> ,}+	The preceding sub-flow may apply <i>n</i> or more times. The iteration proceeds until the sub-flow fails to apply. Backtracking occurs if fewer than <i>n</i> applications succeed. When backtracking arrives from a later point to this sub-flow, rather than backtracking within the sub-flow or its iteration, backjumping takes place to the point before this sub-flow.

$\{n,m\}+$	The preceding sub-flow may apply any number of times between $n$ and $m$ . The iteration proceeds until $m$ applications or until the sub-flow fails to apply. Backtracking occurs if fewer than $n$ applications succeed. When backtracking arrives from a later point to this sub-flow, rather than backtracking within the sub-flow or its iteration, backjumping takes place to the point before this sub-flow.
??	The preceding sub-flow may apply zero or one time. Application will be skipped at first. When backtracking arrives from a later point to this sub-flow, application will be tried. Backtracking occurs if all alternatives within this sub-flow have been exhausted.
*?	The preceding sub-flow may apply zero, one or more times. Application will be skipped at first. When backtracking arrives from a later point to this sub-flow, application be tried or, eventually, repeated. Backtracking occurs if all alternatives within this iterative sub-flow have been exhausted.
+?	The preceding sub-flow may apply one or more times. A single application will be tried at first. If successful, application may be repeated, but only upon backtracking from a later point to this sub-flow. Backtracking occurs if a single application fails or all alternatives within this iterative sub-flow have been exhausted.
$\{n\}?$	The preceding sub-flow may apply exactly $n$ times. The iteration proceeds until $n$ applications or until the sub-flow fails to apply. Backtracking occurs if fewer than $n$ applications succeed or, upon backtracking from a later point to this sub-flow, if all alternatives within this iterative sub-flow have been exhausted.
$\{n,\}$ ?	The preceding sub-flow may apply $n$ or more times. The iteration proceeds until $n$ applications or until the sub-flow fails to apply. If successful, additional applications may be tried, but only upon backtracking from a later point to this sub-flow. Backtracking occurs if fewer than $n$ applications succeed or if all alternatives within this sub-flow have been exhausted.
$\{n,m\}?$	The preceding sub-flow may apply any number of times between $n$ and $m$ . The iteration proceeds until $n$ applications or until the sub-flow fails to apply. If successful, additional applications may be tried, but only upon backtracking from a later point to this sub-flow, and never more than $m$ . Backtracking occurs if fewer than $n$ applications succeed or if all alternatives within this sub-flow have been exhausted.
?*	The preceding sub-flow may apply zero or one time. The one application may be skipped randomly. Backtracking only occurs if all alternatives within this sub-flow have been exhausted.
**	The preceding sub-flow may apply zero, one or more times. The iteration proceeds a selected random number of times or until the sub-flow fails to apply. When backtracking arrives from a later point to this sub-flow, fewer or additional applications may be tried as well, in this order. Backtracking occurs if all alternatives within this iterative sub-flow have been exhausted.

$+^*$	The preceding sub-flow may apply one or more times. The iteration proceeds a selected random number of times (at least one) or until the sub-flow fails to apply. When backtracking arrives from a later point to this sub-flow, fewer or additional applications may be tried as well, in this order. Backtracking occurs if a single application fails or all alternatives within this iterative sub-flow have been exhausted.
$\{n\}^*$	The preceding sub-flow may apply exactly $n$ times. The iteration proceeds until $n$ applications or until the sub-flow fails to apply. Backtracking occurs if fewer than $n$ applications succeed or, upon backtracking from a later point to this sub-flow, if all alternatives within this iterative sub-flow have been exhausted.
$\{n,\}^*$	The preceding sub-flow may apply $n$ or more times. The iteration proceeds a selected random number of times (at least $n$ ) or until the sub-flow fails to apply. When backtracking arrives from a later point to this sub-flow, fewer or additional applications may be tried as well, in this order. Backtracking occurs if fewer than $n$ applications succeed or if all alternatives within this iterative sub-flow have been exhausted.
$\{n,m\}^*$	The preceding sub-flow may apply any number of times between $n$ and $m$ . The iteration proceeds a selected random number of times (at least $n$ and at most $m$ ) or until the sub-flow fails to apply. When backtracking arrives from a later point to this sub-flow, fewer or additional applications may be tried as well, in this order. Backtracking occurs if fewer than $n$ applications succeed or if all alternatives within this iterative sub-flow have been exhausted.

## Appendix D: FAQ

1. *Setup component is red and the error says "Solution exception:No module named mpmath"*

This is an installation issue. mpmath is a python module that is referenced by the plug-in. The module should be found in \Program Files\Rhino 6\Plug-ins\IronPython\Lib\site-packages or equivalent on your computer. Check if you have installed it by copying it (and all other supporting modules) into the site-packages folder. Also, check the Rhino module search paths (Rhino Python Editor window Tools/Options). You must add the site-packages folder to the module search paths.

2. *Setup component is red and the error says "Solution exception:cannot import open from io"*

The 'sortal' library contains a subfolder named 'io'. The Rhino python library already contains a file 'io.py'. When both end up in the same location, Rhino will confuse the subfolder (module) with the file. It is important, when installing 'sortal' into C:\Program Files\Rhino 6\Plug-ins\IronPython\Lib, to copy the folder 'sortal' here, not just the content of 'sortal'.

3. *I'm applying a simple parametric-associative rule to all the faces of a mesh, but some faces are not matched*

This might be an issue of precision of the Rhino/GH data. The SortalGI engine tries to address issues of precision by performing approximate coordinate comparisons, but this doesn't always solve the problem. While Rhino adopts 12 significant figures, small errors may occur when entering geometries, even when using the snap function. Therefore, the default precision adopted by the SortalGI engine when operating within Rhino is 8 significant figures. However, the SGI Setup component sports a precision input that you can adjust the value of in order to try to improve upon the result. Empirical evidence has shown that a precision of 6 to 8 significant figures tends to provide the best results. Even then, a few faces might still evade the matching process, especially in the case of quadrilaterals, or higher degree polygons. An additional rule applying to a copy of one of these faces can help to close the gap.

4. *I get a warning or error that makes no sense to me. What can I do?*

Please recompute the Grasshopper model (F5) or reconnect an input to the SGI Setup component to force this component to recompute. This may resolve the issue; sometimes, a disconnect may occur between the Grasshopper model and the SortalGI engine, which may result in a warning or error with little or no relation to the actual data.

5. *Can I get some help?*

You can post a message on the SortalGI forum (<http://sortal.org/feedback/>) or e-mail [stouffs@sortal.org](mailto:stouffs@sortal.org)