

# SortalGI plug-in for Grasshopper

## User manual

SortalGI version 0.7  
Manual update September 2019  
Written by Rudi Stouffs

### Table of content

1. About the SortalGI plug-in	2
2. Installation and update	3
3. Starting on a SortalGI-based parametric model	5
4. Creating a shape object	7
5. Manipulating a shape object	10
6. Creating a rule	13
7. Applying a rule	16
8. Specifying shape descriptions	20
9. Specifying predicates	27
10. Specifying directives	29
Appendix A. A formal notation for shape descriptions	31
Appendix B. Description functions	35
Appendix C: FAQ	38

## 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.

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 takes three steps. You can find the latest update from Food4Rhino (<http://www.food4rhino.com/app/sortalgi-shape-grammar-interpreter>) or sortal.org (<http://www.sortal.org/downloads/plugin.html>). Installation applies to both Rhino 6 (Windows and Mac) and Rhino 5 (Windows only).

### Step 0: Installing GhPython (Rhino 5 for Windows only)

Only if you are using Rhino 5 (Windows only), you must additionally install GhPython onto Rhinoceros Grasshopper.

- a) Download the GhPython Grasshopper Assembly file (gha) from Food4Rhino: <http://www.food4rhino.com/app/ghpython>
- b) Open Rhino and Grasshopper
- c) In Grasshopper, choose File > Special Folders > Components folder. Save the gha file in this folder
- d) In the finder window, right-click the gha file > Properties and make sure there is no "blocked" text

### Step 1: Installing the SortalGI library (all versions)

This step applies to initial installation as well as every major update (e.g., from v0.5.1 to v0.6.0). Installation of the SortalGI library may be achieved in one of two ways.

- a) If you have not yet done so, download the latest SortalGI update Food4Rhino (<http://www.food4rhino.com/app/sortalgi-shape-grammar-interpreter>) or sortal.org (<http://www.sortal.org/downloads/plugin.html>) and unzip the file.
- b) Windows: Copy-paste the content of the folder 'lib' (the folders 'sortal' and 'site-packages') into the location C:\Program Files\Rhino 6\Plug-ins\IronPython\Lib or equivalent on your computer. Note:
  - i. There may already be a 'site-packages' folder in IronPython\Lib, it suffices to add the content of 'site-packages' to this folder.
  - ii. Adding files to the target folder may require admin access. If you do not have admin access, you can choose a different location to hold these files. You must then add the appropriate module search paths in the following step.
- c) Mac: Copy-paste the content of the folder 'lib' (the folders '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. 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"
  - ii. You **cannot** choose a different location to hold these files. Rhino 6 for Mac does not (yet) support module search paths.

### Step 2: Updating Rhino's Module Search Paths (Rhino for Windows only)

This step only applies to initial installation (Windows only); it needs to be performed only once.

- 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) Add the Plug-ins\IronPython\Lib\site-packages folder of your Rhino installation folder into the 'Module Search Paths'
- e) Switch from the 'Files' tab to the 'Script Engine' tab (in the Python Options window).
- f) Check the 'Frames Enabled' option and click 'OK'
- g) Close Rhino completely

### Step 3: Installing the SortalGI plug-in (all versions)

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 Objects folder.
- c) Copy-paste the content of the folder 'user objects' into this 'UserObjects' folder.

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

In case of a minor update (e.g., from v0.6.0 to v0.6.1), 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 3. Starting on a SortalGI-based parametric model).

### 3. 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 should use CTRL+B in order to ensure the SGI Setup component is executed before all other components.



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 7. 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 6. Applying a rule)
- *text size*: the text size to visualize any labels or shape descriptions that are attributes to geometries resulting from a SortalGI component
- *descriptions*: list of shape description types, each identified by its name (see section 8. Specifying shape descriptions for a specification of descriptions)

#### Outputs:

- *check*: 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.



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:

- *sourceDirectory*: optional source directory where the SortalGI components should be copied from and into the 'UserObjects' folder; if you omit this source directory, 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
- *updateThisFile*: True or False; setting this value to True will execute the SGI Update component; setting it to False will keep it from executing over and over again

Outputs:

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

## 4. Creating a shape object

Creating shape objects using the SGI Shape or SGI dShape components serves different purposes. A shape object can be used to define the left-hand-side or the right-hand-side of a rule. Rule application also requires a shape object as the input shape and, optionally, as the input subshape (see section 7. Applying a rule).

A shape object 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 labels or shape descriptions assigned as attributes. The Text Point, Text Curve and 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.



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. It can also be used to tag a point (see References in section 8. Specifying shape descriptions).

Inputs:

- *P*: point geometry
- *label*: text specifying the tag, label or shape description of the text point (see section 8. Specifying shape descriptions for a specification of descriptions); multiple values can be combined into a single entry by separating them with a vertical bar

Outputs:

- *G*: the resulting text point



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. It can also be used to tag a curve (see References in section 8. Specifying shape descriptions).

Inputs:

- *C*: curve geometry
- *label*: text specifying the label or shape description of the text curve (see section 8. Specifying shape descriptions for a specification of descriptions); multiple values can be combined into a single entry by separating them with a vertical bar

Outputs:

- *G*: the resulting text curve

## Text Surface



The Text Surface component creates a labelled surface object, 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. It can also be used to tag a surface (see References in section 8. Specifying shape descriptions).

### Inputs:

- *S*: surface geometry
- *label*: text specifying the label or shape description of the text surface (see section 8. Specifying shape descriptions for a specification of descriptions); multiple values can be combined into a single entry by separating them with a vertical bar

### Outputs:

- *G*: the resulting text surface

## SGI Shape



The SGI Shape component creates a shape object 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 objects; 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:

- *S*: the resulting shape object
- *G*: the geometry of the shape object

## SGI dShape



The SGI dShape component creates a shape object from geometry, shape descriptions (see section 8. 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 objects; any part of the geometry not recognized will be ignored
- *D*: one or more shape descriptions, each item preceded by the shape description type (name) 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:

- *S*: the resulting shape object
- *G*: the geometry of the shape object
- *D*: the shape descriptions of the shape object (note that any shape description that is assigned as an attribute to part of the geometry is not included as it already forms part of the geometry)

SGI S2G



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

Inputs:

- *S*: a shape object

Outputs:

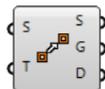
- *G*: the geometry of the shape object
- *D*: the shape descriptions of the shape object (note that any shape description that is assigned as an attribute to part of the geometry of the shape object is not included as it already forms part of the geometry)

## 5. Manipulating a shape object

Following the creation of a shape object, various geometrical operations are available as SortalGI components to act upon a shape object; 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 object and any additional data required to inform and apply the transformation, and returns the resulting shape object as well as the corresponding geometry and shape descriptions. Their operation is quite identical to the corresponding Grasshopper components, except that they act upon a shape object.

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 object along a translation vector. This component is very useful to ensure the visualization of shape objects resulting from rule application do not overlap and are properly spaced (see section 7. Applying a rule).

Inputs:

- *S*: shape object
- *T*: translation vector

Outputs:

- *S*: the resulting shape object
- *G*: the geometry of the resulting shape object
- *D*: the shape descriptions of the resulting shape object

### SGI Rotate Shape



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

Inputs:

- *S*: shape object
- *A*: rotation angle in radians
- *P*: rotation plane

Outputs:

- *S*: the resulting shape object
- *G*: the geometry of the resulting shape object
- *D*: the shape descriptions of the resulting shape object

### SGI Mirror Shape



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

Inputs:

- *S*: shape object
- *P*: mirror plane

Outputs:

- *S*: the resulting shape object
- *G*: the geometry of the resulting shape object
- *D*: the shape descriptions of the resulting shape object

SGI Scale Shape



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

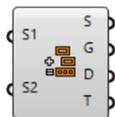
Inputs:

- *S*: shape object
- *C*: center of scaling
- *F*: scaling factor

Outputs:

- *S*: the resulting shape object
- *G*: the geometry of the resulting shape object
- *D*: the shape descriptions of the resulting shape object

SGI Sum



The SGI Sum component sums (combines) two shape objects together.

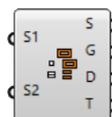
Inputs:

- *S1*: shape object
- *S2*: another shape object

Outputs:

- *S*: the resulting shape object
- *G*: the geometry of the resulting shape object
- *D*: the shape descriptions of the 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 shape objects.

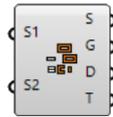
Inputs:

- *S1*: shape object
- *S2*: another shape object

Outputs:

- *S*: the resulting shape object
- *G*: the geometry of the resulting shape object
- *D*: the shape descriptions of the 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 object with respect to another shape object.

Inputs:

- *S1*: shape object
- *S2*: another shape object

Outputs:

- *S*: the resulting shape object
- *G*: the geometry of the resulting shape object
- *D*: the shape descriptions of the 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 component sums (combines) any number of shape objects together.

Inputs:

- *L*: list of shape objects

Outputs:

- *S*: the resulting shape object
- *G*: the geometry of the resulting shape object
- *D*: the shape descriptions of the resulting shape object
- *T*: translation vector that can be used to move/displace the resulting shape wrt the original shapes

## 6. 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 shape objects 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, parallel and perpendicular lines 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 object from a left-hand-side (*lhs*) and a right-hand-side (*rhs*), a name and a (optional) brief description. If a shape description type is present as part of one shape object (*lhs* or *rhs*) but absent from the other shape object, an empty shape description of that type is automatically added to the other shape object within the rule.

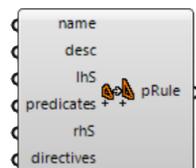
Inputs:

- *name*: rule name; this rule name should be unique
- *desc*: brief description of the rule (optional)
- *lhs*: shape object representing the left-hand-side of the rule
- *rhs*: shape object representing the right-hand-side of the rule

Outputs:

- *rule*: the non-parametric rule object

Sgi pRule



The SGI pRule component creates a **parametric-associative** rule object from a left-hand-side (*lhs*) and a right-hand-side (*rhs*), a name and a brief description, and optional predicates and directives (see sections 9. Specifying predicates and 10. Specifying directives). If a shape description type is present as part of one shape object (*lhs* or *rhs*) but absent from the other shape object, an empty shape description of that type is automatically added to the other shape object within the rule.

Inputs:

- *name*: rule name; this rule name should be unique
- *desc*: brief description of the rule (optional)

- *lhs*: shape object representing the left-hand-side of the rule
- *predicates*: One or more predicates (optional)
- *rhs*: shape object representing the right-hand-side of the rule
- *directives*: One or more directives (optional)

Outputs:

- *pRule*: the parametric-associative rule object

SGI Get Rule 

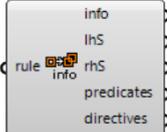
The SGI Get Rule component retrieves a rule object (parametric-associative or non-parametric) by its name.

Inputs:

- *name*: rule name

Outputs:

- *rule*: (non-parametric or parametric-associative) rule object (or null)

SGI Rule Info 

The SGI Rule Info component deconstructs any rule object (parametric or non-parametric) into its left-hand-side and right-hand-side shape objects, a multi-line text containing the rule object's GUID, rule name and description, and its predicates and directives, if any.

Inputs:

- *rule*: rule object

Outputs:

- *info*: multi-line text including the rule's GUID, name and description
- *lhs*: the left-hand-side shape object
- *rhs*: the right-hand-side shape object
- *predicates*: predicates text
- *directives*: directives text

## 7. Applying a rule

Applying a rule to a given shape object 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 (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.

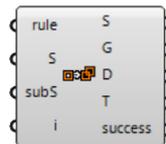
The SortalGI plug-in distinguishes four rule application components: the first one, SGI Apply, applies only a single match (either randomly selected or specified by its index), while the second one, SGI Apply All, applies all matches in parallel, returning as many results as there are matches, and the third one, SGI Apply All Together, applies all matches together, returning a single result that combines (sums) all parallel results. 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 object and an optional subshape object. If specified, the latter must be a subshape, that is, part of, the former. If a subshape object 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 object.

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.

Every resulting shape is accompanied by a translation vector. In the case of SGI Apply, 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, 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 but applies only a single one, either randomly selected or as specified by an index value.

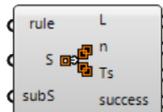
Inputs:

- *rule*: a rule object
- *S*: a shape object to apply the rule to
- *subS*: an optional shape object to restrict matches to; if specified, this shape object must be a subshape, that is, part of, the shape object *S*
- *i*: an 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:

- *S*: the resulting shape object upon rule application; if no match is found then the original shape is returned
- *G*: the geometry of the resulting shape object
- *D*: the descriptions of the resulting shape object
- *T*: a translation vector to allow the resulting 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.

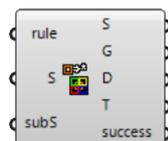
Inputs:

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

Outputs:

- *L*: a list of resulting shape objects corresponding to the number of rule applications found; if no match is found then the original shape is returned
- *n*: the number of matches found, corresponds to the length of the lists *L* and *Ts*
- *Ts*: a list of translation vectors to allow the resulting 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 all possible matches and combines them into a single shape.

Inputs:

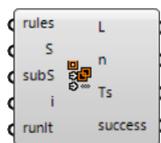
- *rule*: a rule object

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

Outputs:

- *S*: the resulting shape object combining the results from all rule applications; if no match is found then the original shape is returned
- *G*: the geometry of the resulting shape object
- *D*: the descriptions of the resulting shape object
- *T*: a translation vector to allow the resulting 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 rule objects, it applies each in sequence.

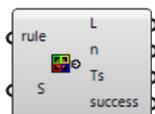
Inputs:

- *rules*: a list of rule objects
- *S*: a shape object to apply the first rule to
- *subS*: an optional shape object to restrict the first match to, or a list of shape objects to restrict consecutive matches to; if specified, the shape object must be a subshape, that is, part of, the input shape of the respective rule object
- *i*: an 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*: True or False indicating whether to execute the component or not

Outputs:

- *L*: a list of resulting shape objects, one for each rule object; if no match is found for a specific rule object then the input shape for that rule is returned
- *n*: the length of the lists *L*, *Ts* and *success*
- *Ts*: a list of translation vectors to allow the resulting shapes to be drawn one next to the other and to the original shape, along the X-axis
- *success*: a 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. Note that depending on the right-hand-side of the rule, identical matches may result corresponding to otherwise distinct rule applications.

Inputs:

- *rule*: a rule object
- *S*: a shape object to match the rule to

Outputs:

- *L*: a list of resulting shape objects corresponding to the rule matches
- *n*: the number of matches found, corresponds to the length of the lists *L* and *Ts*

- *Ts*: a list of translation vectors to allow the resulting 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

## 8. 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 description type name and 'a' is a parameter):  
description: a

### Shape description types

A single shape object or rule object may specify more than one description. For example, one description may be used to constrain rule application while another may serve to count the number of rule applications performed on the shape object. In order to be able to correctly match shape descriptions belonging to the *lhs* and the *rhs* of the rule object, 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 name (type name and description are separated by a colon). Shape description types must be prescribed in the SGI Setup component (see section 3. Starting on a SortalGI-based parametric model).

Multiple 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"

### 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

## 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)

## 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 conditional, 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]

### 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' 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 expressions 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)

## Functions

Functions allow for additional operations on numbers, 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 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, 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:

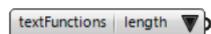
positions: a (`random(0,10,1)`, 0)

SGI Numeric Functions



The SGI Numeric Functions selector offers a list of numeric functions for descriptions.

SGI Text Functions



The SGI Text Functions selector offers a list of text functions for descriptions.

SGI Function Concat



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

Inputs:

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

Outputs:

- *text*: concatenated function name 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 that 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 description (that is part of the same rule). In such case, the parameter name must be preceded by the description type name 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*.

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 name (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 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:labelD.value="1", point3D.value:labelD.value="2")`  
constraint: `a?>=mag(#pt1.value, #pt2.value)`

## SGI Spatial Types



The SGI Spatial Types selector offers a list of spatial types as may be present in shapes. Each spatial type is represented as a numeric value: 0 = point, 1 = line segment, 2 = plane segment, 3 = circle, 4 = ellipse, 5 = circular arc, 6 = quadratic Bezier.

## 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 name or, if specified, an element tag. This combination can be used in a description rule to retrieve the property value. The type name is dependent on the spatial type and whether it applies to a parametric-associative or non-parametric rule.

Inputs:

- *styp*: spatial type name or number (0 = point, 1 = line segment, 2 = plane segment, 3 = circle, 4 = ellipse, 5 = circular arc, 6 = quadratic Bezier)
- *parametric*: True or False value indicating whether the element type name refers to a parametric-associative or non-parametric rule
- *tag*: optional element tag

Outputs:

- *properties*: list of concatenated element tag/type name and property names for a spatial type

Sgi Description Reference



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

Inputs:

- *type*: either a description type name or a spatial type number (0 = point, 1 = line segment, 2 = plane segment, 3 = circle, 4 = ellipse, 5 = circular arc, 6 = quadratic Bezier)
- *parameter*: optional parameter name

Outputs:

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

### Shape element types and their available properties

Every geometric shape element 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	vector tuple* vector tuple* vector tuple* vector tuple* vector tuple* number number	direction vector unit direction vector 'smallest' endpoint 'greatest' endpoint midpoint line length square value of line length
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	vector tuple*	plane normal vector
	circleP3D	center radius diameter circumference area	vector tuple* number number number number	center point radius diameter circumference area of the circle

ellipses	ellipse3D	normal	vector tuple*	plane normal vector
	ellipseP3D	center foci	vector tuple* tuple of vector tuples*	center point list of focal points
		radii	tuple of numbers	list of longer and shorter radii
		area	number	area of the ellipse
circular arcs	arc3D	normal	vector tuple*	plane normal vector
		center radius diameter circumference	vector tuple* number number number	circle center point circle radius circle diameter circle circumference
		start end length angle	vector tuple* vector tuple* number number	endpoint (ccw) endpoint (cw) arc length angle covered by the arc (in radians)
		area	number	area covered by the arc
quadratic Bezier curves	bezier3D	normal	vector tuple*	plane normal vector
	bezierP3D	start 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 point attribute	labelD	value	string	label or description string

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

## 9. Specifying predicates

A predicate serves to express a special condition on the application of a parametric-associative rule. Such condition cannot simply be explicated within the left-hand-side shape. As an example, a predicate may specify that the line segment matching a given line segment must be the shortest line segment within the matching shape. Predicates are only applicable to parametric-associative rules.

### SGI Maxline Predicate

The SGI Maxline Predicate component creates a maxline predicate text from one or more line element tags. The maxline predicate stipulates that any line segment matching the tagged line segment must use its full extent to match the line segment.

Inputs:

- *line*: one or more element tags of line segments

Outputs:

- *text*: predicate text

### SGI Bound Predicate

The SGI Bounds Predicate component creates a bound predicate text 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. 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:

- *line*: one or more element tags of line segments
- *start*: one or more Boolean values specifying whether the startpoint of the (respective) line segments must be a boundary point or not
- *end*: one or more Boolean values specifying whether the endpoint of the (respective) line segments must be a boundary point or not

Outputs:

- *text*: predicate text

### SGI Void Predicate

The SGI Void Predicate component creates a void predicate text from one or more polygonal geometries and, optional, spatial types. 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 text explicates the vertices by their coordinates, 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*: one or more polygonal geometries; each may be expressed as points, line segments, closed polyline, flat surface or boundary representation
- *stype*: one or more spatial type names or numbers (0 = point, 1 = line segment, 2 = plane segment, 3 = circle, 4 = ellipse, 5 = circular arc, 6 = quadratic Bezier)

Outputs:

- *text*: predicate text

Sgi Shortest-Line Predicate



The SGI Shortest-Line Predicate component creates a shortest line predicate text from one or more line element tags.

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 multiple inputs, the matched lines identified as the shortest lines must all have the same length.

Inputs:

- *lines*: one or more element tags of line segments

Outputs:

- *text*: predicate text

Sgi Longest-Line Predicate



The SGI Longest-Line Predicate component creates a longest line predicate text from one or more line element tags.

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 multiple inputs, the matched lines identified as the longest lines must all have the same length.

Inputs:

- *lines*: one or more element tags of line segments

Outputs:

- *text*: predicate text

## 10. 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 a 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 text from one or more 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 parameter value can be explicated as a numeric value between 0 and 1 or as a description text enclosed within backward quotes. For example, the description ``random(0.3, 0.7)`` prescribes a random value between 0.3 and 0.7.

Inputs:

- *line*: one or more element tags of line segments
- *value*: one or more parameter values, each either a numeric value or a description text (enclosed within backward quotes)

Outputs:

- *text*: directive text

### SGI Distance Directive



The SGI Distance Directive component creates a distance directive text from one or more target and reference element tags and distance values. 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
- *Line-point distance*: the new line must run through an existing point, line endpoint or line intersection point
- *Point-line distance*: the new point must be on another existing line not parallel to the reference line
- *Point-point distance*: the new point must be on an existing line

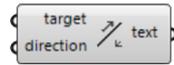
Inputs:

- *target*: one or more element tags of target points or line segments (*lhs* shape)
- *reference*: one or more element tags of reference points or line segments (*rhs* shape)
- *dist*: one or more distance values, each either a numeric value or a description text (enclosed within backward quotes)

Outputs:

- *text*: directive text

### SGI Direction Directive



The SGI Direction Directive component creates a direction directive text 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 text enclosed within backward quotes. For example, the description ``#plane.normal`` prescribes the normal vector of a tagged plane as the direction vector.

Inputs:

- *target*: one or more element tags of line segments (*rhs* shape)
- *direction*: one or more direction vectors, each either a coordinate tuple or a description text (enclosed within backward quotes)

Outputs:

- *text*: directive text

### SGI Length Directive



The SGI Length Directive component creates a length directive text 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:

- *target*: one or more element tags of line segments (*rhs* shape)
- *length*: one or more length values, each either a numeric value or a description text (enclosed within backward quotes)

Outputs:

- *text*: 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 l-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 ‘)’ .  
<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 = [ '-' ] 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

### String 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
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
round	1 vector tuple*	A vector tuple with each value rounded to the nearest integer
mag	2 vector tuples*	The distance between the two vectors
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: FAQ

*1. 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.

*2. 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)