SortalGI plug-in for Grasshopper

User manual

SortalGI version 0.3.x
Manual update 30 May 2018
Written by Rudi Stouffs

Table of content

1. About the SortalGI plug-in

A shape rule combines a specification of recognition and manipulation. 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. 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
System development by Bui Do Phuong Tung
Under the supervision of Rudi Stouffs

2. Installation and update

Installation takes three steps. The first step only needs to be performed once and only if you are using Rhino 5. The second step is also required for any major update (e.g., from v0.2.2 to v0.3.0), while the third step should be performed for any update, though in case of a minor update (e.g., from v0.2.1 to v0.2.2), you may be able to simply use the SGI Update component (see below).

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)

Step 1:Installing GhPython and updating Rhino's Module Search Paths

If you are using Rhino 5, you must install GhPython onto Rhinoceros Grasshopper. This is not necessary if you are using Rhino 6.

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.
e) Type EditPythonScript in the Rhino Command box.
f) In the Rhino Python Editor window, select 'Options...' from the Tools menu.
g) Add the Plug-ins\IronPython\Lib\site-packages folder of your Rhino installation folder into the 'Module Search Paths'.
h) Close Grasshopper and Rhino completely.

Step 2: Installing the SortalGI library

As the SortalGI library may change with a major update (e.g., from v0.2.2 to v0.3.0) and there may be dependencies between the plug-in and such changes, it is strongly advised to re-install the SortalGI library with every major update. This installation 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) (in Windows) Run the setup widget inside the folder 'sortal-setup'.
The computer may prompt for whether to allow the widget to make changes; click 'Yes' or 'Allow'. Wait for it to finish installing the packages.
c) If you are unable to run the 'setup' widget, you may manually copy-paste the files in their respective locations:
    i. Copy the content of the folder 'sortal-setup\site-packages' into the location C:\Program Files\Rhinoceros 5.0 (64-bit)\Plug-ins\IronPython\Lib\site-packages or equivalent on your computer
    ii. Copy the folder 'sortal' (inside 'sortal-setup\sortal-packages') into the location C:\Program Files\ Rhinoceros 5.0 (64-bit)\Plug-ins\IronPython\Lib

Step 3: Installing the SortalGI plug-in

With any update of the SortalGI plug-in, you must update the SortalGI components in Grasshopper. You can:

a) Open Rhino and Grasshopper.
b) In Grasshopper, choose File > Special Folders > User Objects folder.
c) Copy all files from the folder 'components' and paste these into the folder 'UserObjects'.

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.

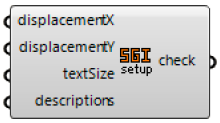
In case of a minor update (e.g., from v0.2.1 to v0.2.2), you can also use the SGI Update component to update the SortalGI components in the Grasshopper Components Tab Panel as well as in the current parametric model (see section 3. Starting on a SortalGI-based parametric model).

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 F5 in order to ensure 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 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 7. 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*: a 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 value indicating success of the setup

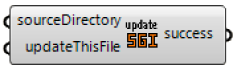
Opening an existing parametric model using SortalGI components

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:

- *sourceDirectory*: an 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

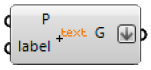
4. Creating a shape object

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, as well as shape descriptions. Points may have labels or shape descriptions assigned as attributes. The Text Point component allows one to assign a label or shape description as a text to a point. 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.

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.

Text Point



The Text Point component creates a labelled point object, that is, a point with a label or shape description as attribute.
Inputs:

- *P*: a point specifying the position of the object
- *label*: a text specifying the label or shape description of the object (see section 8. Specifying shape descriptions for a specification of descriptions)

Outputs:

- *G*: the resulting text object

SGI Shape

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

- *G*: a geometry of points, text points, lines, polylines, (flat) surfaces, meshes and/or boundary representations; any part of the geometry not recognized will be ignored
- *refP*: an 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 a geometry, descriptions and an optional reference point. The descriptions may be omitted, so may be the geometry, though not both at the same time.

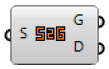
Inputs:

- *G*: a geometry of points, text points, lines, polylines, (flat) surfaces, meshes and/or boundary representations; 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*: an 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:

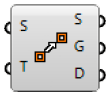
- *G*: the geometry of the shape object
- *D*: the shape descriptions of the shape object (note that any shape description that is assigned as an attribute to part of the geometry of the shape object is not included as it already forms part of the geometry)

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 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).

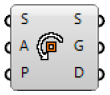
Inputs:

- *S*: a shape object
- *T*: a 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.

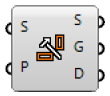
Inputs:

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

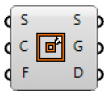
- *S*: a shape object

- *P*: a 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*: a shape object
- *C*: a center of scaling
- *F*: a 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*: a 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

SGI Product

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

Inputs:

- *S1*: a 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

SGI Difference

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

Inputs:

- *S1*: a 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

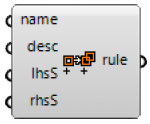
6. Creating a rule

A rule is conceptually specified in the form *lhs* → *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 (see section 6. Creating a rule).

SGI Rule



The SGI Rule component creates a rule object from a left-hand-side (*lhs*) and a right-hand-side (*rhs*), a name and a 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.

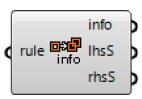
Inputs:

- *name*: a rule name; this rule name should be unique
- *desc*: a brief explanation of the rule
- *lhs*: a shape object representing the left-hand-side of the rule
- *rhs*: a shape object representing the right-hand-side of the rule

Outputs:

- *rule*: the rule object

SGI Rule Info



The SGI Rule Info component deconstructs a rule object into its left-hand-side and right-hand-side shape objects and provides a multi-line text containing the rule object's GUID, name and description.

Inputs:

- *rule*: a rule object

Outputs:

- *info*: a multi-line text including the rule's GUID, name and description
- *lhsS*: the left-hand-side shape object
- *rhsS*: the right-hand-side shape object

7. Applying a rule

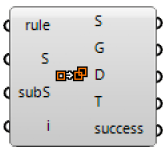
Applying a rule to a given shape object involves determining a transformation under which the left-hand-side (*lhs*) of the rule is a part of the given shape. That is, rule application involves two steps: recognition and manipulation; recognition implies matching the *lhs* of the rule under some transformation 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* specifies a square, the rule will match any square in the given shape independent of its location, rotation or scale (a similarity transformation). However, these matches may also apply to the same part in different ways. Again, if the *lhs* 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 two 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. Both 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.

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. The extent of the translation vector is specified by the *displacementX* and *displacementX* 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).

SGI Apply



The SGI Apply component.
Inputs:

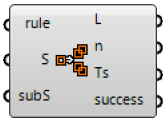
- *rule*: a rule object
- *S*: a shape object to apply the rule to
- *subS*: an optional shape object to restrict matches to; if specified, this shape object must be a subshape, that is, part of, the shape object *S*

- *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.

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 matches 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

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 description 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 'π', 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 (vectorsubstract) 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)

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. In order to reference shape data, you must refer to the element type name (see Shape element types below). However, this will only work if there is only one element of the specific type, otherwise the reference will be ambiguous. In the case of points, you can disambiguate the point by additionally specifying its label, provided the point has a label and the label is unique (see example below).

Example querying the positions of two points with given labels:

constraint: a?>=mag(point.value:labelD.value="1", point.value:labelD.value ="2")

Shape element types

Type	**Name**
points	point
Line segments	lineSeg
Plane segments	planeSeg
Labels/descriptions as point attribute	labelD

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.

typed-description = type-name ':' description . type-name = identifier . description = description-entity | description-sequence . description-entity = literal | top-level-tuple . description-sequence = '&' description-entity '&' { description-entity '&' } .
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 | '\' '"' .
Example **description-entity**: "centrally divided, double 1-rafter beam in front and back" Example **description-sequence**: &e&0&"nothing"&
top-level-tuple = tuple | unmarked-tuple . tuple = '(' tuple-entities ')' | '<' [tuple-entities] '>' | '[' [tuple-entities] ']' . <description>tuple-entities = tuple-entity-sequence . <lhs>tuple-entities = tuple-entity-sequence | tuple-expression . <rhs>tuple-entities = tuple-entity-sequence | tuple-expression . tuple-entity-sequence = tuple-entity ({ ',' tuple-entity } | { ';' tuple-entity }) . <description>tuple-entity = literal | tuple . <lhs>tuple-entity = numeric-expression | string-expression | tuple . <rhs>tuple-entity = numeric-expression | string-expression | tuple | function-returns-tuple . unmarked-tuple = tuple-expression | tuple (tuple | keyword-literal) { tuple-entity } .
Example **tuple**: ("l:", 10, "c:", (0, 0), "r:", 0) Example **unmarked-tuple**: <" ", "O", "R0", "R1"> <"O", 1, 1, 1> <"R0", 1, 1, 0> <"R1", 1, 0, 1>
description-rule-side = description-rule-entity | description-rule-sequence . <lhs>description-rule-entity = literal | parameter ['?' conditional] | string-expression | top-level-tuple . <rhs>description-rule-entity = numeric-expression | string-expression | function-returns-tuple | tuple-expression . description-rule-sequence = '&' description-rule-entity '&' { description-rule-entity '&' } .
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 | 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 .
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 ‘)’)
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

*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
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