SortalGI API for Rhino

User Manual

Manual update 19 August 2018
Written by Bianchi Dy and Rudi Stouffs

Table of Contents

About the SortalGI API

A shape rule combines a specification of recognition and manipulation. A shape rule is commonly specified in the form *lhs* → *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 $s - f(a) + f(b)$.

Two types of rules are distinguished, parametric 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 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.

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 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 API for Rhinoceros provides access to the SortalGI *sortal*/shape grammar interpreter and makes (part of) its functionality available within the Rhino Python programming environment. It allows the user to create and apply shape and description rules by merging the capabilities of Rhinoceros in drawing shapes and storing user data and descriptions onto geometry with the computational capabilities of the shape grammar interpreter.

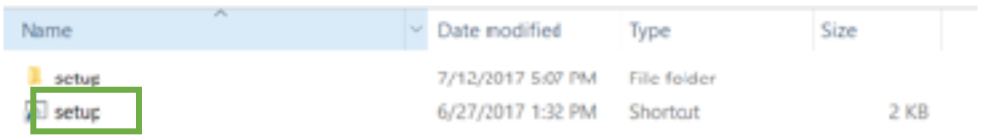
API development by Bianchi Dy
System development by Bui Do Phuong Tung
Under the supervision of Rudi Stouffs

SortalGI Installation and Setup for Rhinoceros

The following sections describe the step-by-step process of installing the SortalGI grammar interpreter and its API onto Rhinoceros.

1. Installing the SortalGI library

Run the 'setup' widget inside the folder 'sortal-setup'. In addition to installing the 'sortal' library, this will install packages such as 'future', 'enum' and 'mpmath' in the right location, which are necessary for compatibility between IronPython and the SortalGI API.



Name	Date modified	Type	Size
setup	7/12/2017 5:07 PM	File folder	
setup	6/27/2017 1:32 PM	Shortcut	2 KB

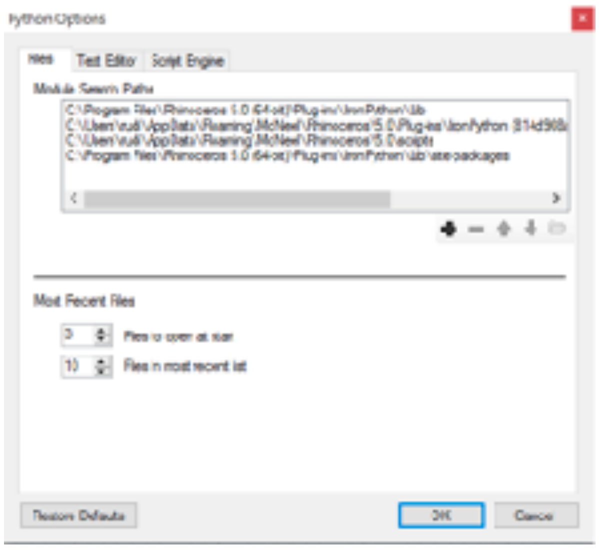
When prompted for Administrator Access by the 'setup' batch file, select 'Yes'. Wait for the packages to finish installing.

Alternatively, if you are unable to run the 'setup' widget, you may manually copy-paste the files in their respective locations:

- site-packages ≫ all files inside
  - Copy-paste the content of the folder 'sortal-setup\site-packages\ into the location C:\Program Files\Rhinoceros 5 (64-bit)\Plug-ins\IronPython\Lib\site-packages or equivalent on your computer
- sortal-packages ≫ sortal
  - Copy-paste the folder 'sortal' (inside 'sortal-setup\sortal-packages) into the location C:\Program Files\Rhinoceros 5 (64-bit)\Plug-ins\IronPython\Lib\ or equivalent on your computer

2. Linking the SortalGI library to Rhinoceros

- ➔ Open Rhino
- ➔ Type `EditPythonScript` in the Rhino command box
- ➔ In the Rhino Python Editor window, select 'Options…' from the Tools menu
- ➔ Add the Plug-ins\IronPython\Lib\site-packages folder of your Rhino installation folder into the 'Module Search Paths'
- ➔ Switch from the 'Files' tab to the 'Script Engine' tab (in the Python Options window)
- ➔ Check the 'Frames Enabled' option and click 'OK'
- ➔ Close Rhino completely and relaunch it for the changes to take effect



Python Options

Files | Text Editor | Script Engine

Module Search Paths

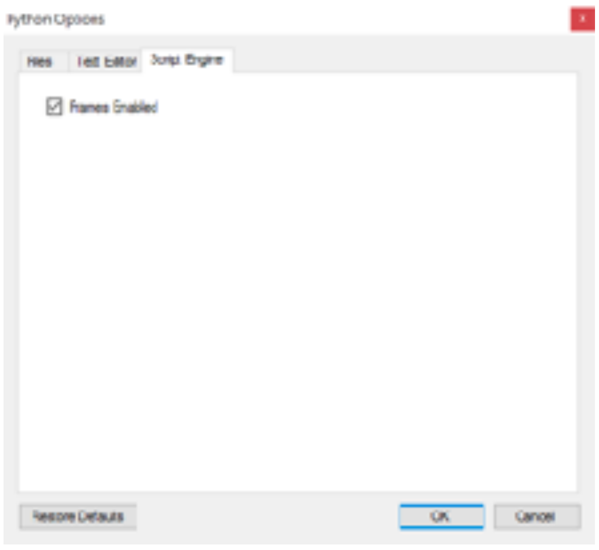
```
C:\Program Files\Rhinoceros 5.0 (64-bit)\Plug-ins\IronPython\Lib
C:\Users\vud\AppData\Roaming\McNeel\Rhinoceros\5.0\Plug-ins\IronPython (814d908a
C:\Users\vud\AppData\Roaming\McNeel\Rhinoceros\5.0\scripts
C:\Program Files\Rhinoceros 5.0 (64-bit)\Plug-ins\IronPython\Lib\site-packages
```

Most Recent Files

3 Files to open at start

10 Files in most recent list

Restore Defaults | OK | Cancel



Python Options

Files | Text Editor | Script Engine

☑ Frames Enabled

Restore Defaults | OK | Cancel

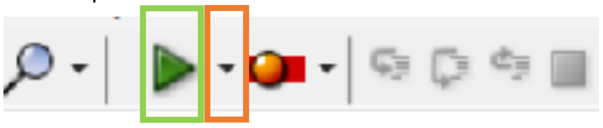
3. Setting up the SortalGI library in Rhinoceros

Before running any SortalGI API or Rhino API methods, it is necessary to set up the SortalGI library first. This involves defining the *sorts* (*sortal* structures) that will be used. There are two ways to do this. Both ways refer to the folder 'sortalgi', which can be found in the folder 'demos\Rhino demo'. All SortalGI API functions can be found in 'sortalgi'.

a. **Run the default setup script** by importing the SortalGI API into your code:

```
import sortalgi as _sgi
_sgi.sortal_setup()
```

To ensure that this script works, save it in the same folder hierarchy level as the 'sortalgi' folder. To run the script, use the **'Reset engine and debug' option** in the Rhino Python Editor, found where the **orange** box indicates in the picture below.



See if you get the response 'Setup complete' in your Python Editor window, in which case the sortal library and API functions are now ready for use. Note that any scripts run after 'sortal_setup()' is called that use Sortal API functions must be run using the green Play button or the 'Run Script (no debug)' option, indicated by the **green** box. This prevents the SortalGI library from refreshing completely and maintains previously set up *sorts*.

The 'sortal_setup' function is explained in Annex D: Legacy Methods, it is considered a legacy function that you can use as is, or you can customize it for your own purpose (see 'sortalgi_init_.py' for some additional explanation and its implementation). Note that if you customize 'sortal_setup' or define your own script to define the *sorts* (*sortal* structures) for the SortalGI grammar interpreter, you must ensure that the highest tier *sort* is a compound *sort* named 'rhino_shapes_c'. All SortalGI API functions refer to the compound *sort* named 'rhino_shapes_c'.

b. **Upload a *Sortal* Description Language (SDL) file** by using the function 'read_sdl' from the API functions:

```
import sortalgi as _sgi
_sgi.read_sdl(<sdl file name>)
```

Where <sdl file name> is the SDL file name with a '.sdl' extension. This input must be a text string, and may include a specification of the location of the SDL file if it is not located within the same folder as the active Python code. However, any slashes must be replaced with double slashes. For example:

```
_sgi.read_sdl(McNeel\\Rhinoceros\\5.0\\scripts\\sortal_3D\\demo scripts\
\rhino demo\\sortalgi\\sortal_lib_api)
```

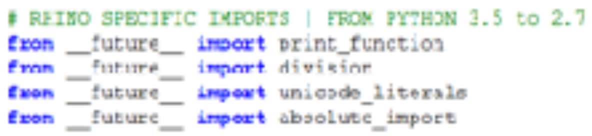
The format and notation of the Sortal Description Language (SDL) is not explained in this document. If you are interested in more information about this, you are welcome to investigate an example SDL file (included in some of the Rhino demos) or you may find some (incomplete) information at http://www.sortal.org/structures/SDL/index.html

If the highest tier *sort* defined inside the SDL file is not a compound *sort* named 'rhino_shapes_c', the 'read_sdl' function will attempt to resolve this by defining (or rewriting) a compound *sort* named 'rhino_shapes_c' to contain any disjunctive *sorts* defined in the SDL file. If a compound *sort* named 'rhino_shapes_c' already existed in the SortalGI grammar interpreter, its existing disjunctive *sorts* will remain unless overwritten.

4. Importing future for Python 3.5 to Python 2.7 compatibility

When writing scripts using the SortalGI API and library in Rhino or Python 2.7, it is necessary to put the following lines of code at the start of every script to import the 'future' package. This enables compatibility between Python 3.5 (which is used to write the SortalGI API and library) and RhinoPython, which uses IronPython (equivalent to Python 2.7). An example from the API itself is shown on the right-hand side below:

```
from __future__ import print_function
from __future__ import division
from __future__ import unicode_literals
from __future__ import absolute_import
```



```
# RHINO SPECIFIC IMPORTS | FROM PYTHON 3.5 to 2.7
from __future__ import print_function
from __future__ import division
from __future__ import unicode_literals
from __future__ import absolute_import
```

API

All functions described in this section (and imported under the comment 'API function imports' in the sortalgi '__init__.py' file) comprise the SortalGI API. Their uses, inputs and outputs are discussed in this section.

Import notation

- The use of 'sgi' in this section refers to the package 'sortalgi', which contains all the methods listed above and is imported in code snippet examples as follows:

  ```
  import sortalgi as sgi
  ```

- The references to 'sc.sticky' in this section, on the other hand, refer to scriptcontext.sticky, which is a dictionary used to store values like the number of predicates/directives of a certain type currently active in a run of the sortal library, or the spacing allowance (in the prevailing workspace units) for drawing shapes in the Rhino viewport. It is imported as follows:

  ```
  import scriptcontext as sc
  ```

- The terms 'shape' and 'form' are used interchangeably in this section.
- The Rhinocommon library is imported in the following variations:

  ```
  import Rhino as r
  import Rhino.Geometry as rg
  ```

Summary of all methods

NAME	PURPOSE
apply_flow	Applies a flow present in the flow register onto a shape present in the form register; returns the result of the flow application as a list of Rhino GUIDs, which are by default hidden from the Rhino viewport; an optional Boolean value called hide may be set to False to render the resulting list of Rhino GUIDs visible in the viewport
check_precision	Returns current precision (as number of decimal places) of sortal library
convert_shape	Converts a sortal shape to the target sort type (this target sort type may be a product of ‘convert_sort’ or retrieved directly from the sort register)
convert_sort	Determines the structure of a shape's top sort type and creates the target equivalent sort type based on user input; all external description types and attributes attached to geometries are reflected in the newly constructed sort (i.e. 2D -> 3D, P2D -> 2D)
create_flow	Creates a flow based on a text string elaborating the order and use of existing rules in the sortal rule register
create_rule	Creates a rule object from the inputs rule name, rule description, LHS shape name and RHS shape name
create_shape	Creates a shape object from the following inputs: shape name, Rhino geometry, target sort type; the target sort type may be left blank if there is only one functioning geometric disjunctive sort active
default_precision	Resets precision (as number of decimal places) of sortal library to 5
draw_rule	Draws the sides of a rule as Rhino geometry, side by side one another; the drawing may be moved to a different location by inputting a reference point (this may be a tuple/list of three numbers or a Rhino geometry point or GUID), where the reference point serves as the new 'origin' for the drawing
draw_shape	Draws the shape retrieved from the form register as Rhino geometry; the drawing may be moved to a different location by inputting a reference point (this may be a tuple/list of three numbers or a Rhino geometry point or GUID), where the reference point serves as the new 'origin' for the drawing
extract_shape	Extracts a sub-shape from a sortal shape based on the target sort type provided by the user
find_rule_appns	Generates the rule applications from a given rule-shape combination; takes as input a rule object or rule name, a sub-shape name (optional) and a main shape name and an optional Boolean value as to whether to hide the resulting Rhino GUIDs corresponding to the results; returns list of lists of GUIDs corresponding to resulting shapes after rule applications, with the GUIDs hidden from the Rhino viewport by default
get_rule_lhs	Returns LHS of rule instance as a list of Rhino GUIDs

NAME	PURPOSE
get_rule_rhs	Returns RHS of rule instance as a list of Rhino GUIDs
get_rule_description	Returns description of rule instance as a text string
maximalize	Maximalizes a sortal shape (called by its name from the form register)/list of Rhino geometries (if the latter, redraws the Rhino geometries and returns them)
move	Moves lists of GUIDs in a list apart from another based on the dimensions of each list's collective bounding box, and the axis of movement provided by the user; the default is to move the shapes to the right (based on the x-axis) and upwards
overwrite_sdl	Overwrites a pre-existing SDL file or overwrites certain rules, shapes or flows in a pre-existing SDL file; this function may also be used to add rules, shapes or flows to a pre-existing SDL file
part_of	Checks if a sub-shape agnostic object/list of Rhino geometries is part of a potentially larger shape agnostic object/set of Rhino geometries
read_sdl	Sets up rules and shapes from SDL file in sortal library; adds shapes to form register; converts SDL file rules and shapes to prevailing 'rhino_shapes_c' sort type; rewrites contents of read SDL file to reflect prevailing 'rhino_shapes_c' structure, if necessary
redraw	Deletes inputted Rhino geometry and replaces them with their geometric counterparts and corresponding labels, descriptions, predicates and directives based on data stored inside the user text of the original geometry
save_sdl	Creates a new SDL file or overwrites a pre-existing one using the inputted file name and the listed rules, shapes and flows
set_flow_description	Changes the description of the given flow object
set_flow_name	Changes the flow name of the given flow instance; if the new flow name matches that of a pre-existing flow, prompts user for overwrite or renaming
set_precision	Sets precision (as number of decimal places) of sortal library to user input
set_rule_description	Changes the description of the given rule object
set_rule_name	Changes rule name of rule instance; if new rule name matches that of a pre-existing rule, prompts user for overwrite or change of rule name input
set_shape_name	Changes name of shape object and updates its name in the form register

apply_all_together

Finds the rule applications of a rule on a shape (an optional subshape input may be used to limit the number of matches found within the shape). It applies all rule applications onto the shape in parallel and returns the sum of the results of the rule applications as a list of Rhino GUIDs. By default, these drawings are hidden from the Rhino viewport.

Syntax

```
sgi.apply_all_together(chosenRule, shape, subshape = None, refPt =
r.Geometry.Point3d(0,0,0), layerName = 'Default', hide = True prnt = False,
shapeIds = None)
```

Parameters

Required

- chosenRule: Rule name as text string
- shape: Name of main shape as text string as recorded in sortal library's form register pertaining to shape

Optional

- subshape: Name of subshape as text string as recorded in sortal library's form register pertaining to subshape
- refPt: GUID of point, tuple, or Rhino Geometry point of reference point which will serve as the 'origin' with which shapes will be plotted in respect to (in principle, a vector)
- layerName: Name of target layer as text string within Rhino workspace where the GUIDs will be drawn onto
- hide: Boolean value True/False; True (default) – hides Rhino GUIDs from viewport; False – keeps Rhino GUIDs visible in viewport
- prnt: Boolean value True/False; True - prints out description individuals as input-ready text string (to create_shape) as well as printE form of shape after rule application; False (default) - nothing is printed
- shapeIds: List of Rhino GUIDs to clear after new shapes have been drawn

Returns

- appnGeometry: List of lists of Rhino GUIDs (each list correspond to a shape after a certain rule application), if successful; these geometries are hidden from the viewport, by default
- None, if unsuccessful

Warnings & Errors

- TypeError: If initial shape or subshape is not a list of GUIDs or a text string or is empty, or if the input for chosenRule is not a text string
- KeyError:
  - If initial shape or subshape input's name is not present in the form register
  - If rule name is not present in the rule register
  - If 'rhino_shapes_c' is not present in the sort register, i.e. sort types in sortal library has not yet been set up
- Warning: If subshape is not part of shape -> exits function and returns False

Example

```
chosenRule = 'rule_1'
shape = 'shape_1'
subshape = 'shapelineSegments'
ruleAppns = sgi.find_rule_appns(chosenRule, shape, subshape)
```

The function returns a list of lists of Rhino GUIDs, each element list corresponding to a rule application. By default, the shapes are drawn on top of one another and hidden from the Rhino viewport. The function 'move' may be used to space the results out from one another, and rs.ShowObjects may be used on the elements inside 'ruleAppns' to show the shapes.

Pseudocode Snippet

This section illustrates a general overview of the rule application process seen in 'apply_all_together' in the form of a truncated pseudocode describing the contents of the function. Note that some of the notation here may not necessarily reflect available functions in the API. This pseudocode is written such that sections of the pseudocode may be used by the user to create their own rule application 'convenience functions' for interacting with the sortal library. This code snippet assumes that all inputs given are correct.

```
def apply_all_together(ruleName, shapeName, subshapeName = None, refPt =
r.Geometry.Point3d(0,0,0), layerName = 'Default'):
      # Retrieval of sortal objects
      # This section is similar across rule application functions
ruleObject = rule.register[ruleName]
      shapeSortal = form.formRegister[shapeName]
      totalShape = None

      # Determination of which shape to base rule application detection on
      if subshapeName != None:
            subshapeSortal = form.formRegister[subshapeName]
            ruleAppns = ruleObject.detect(subshapeSortal)
      else:
            ruleAppns = ruleObject.detect(shapeSortal)

      # This section will vary based on the intended behavior by the user
# Proceeds with summing up results of rule application if > 0
      # Returns the sum
      if ruleAppns > 0:
            results = []
            for appn in ruleAppns:
                  results.append(appn.perform(shapeSortal.duplicate()))
            totalShape = results[0]
            for result in results[1:]:
                  totalShape.sum(result)
            return totalShape
      else:
            return None
```

apply_flow

Applies a flow operation (a sequence of rules with given instructions for looping, order of application and other conditions) onto a shape. The flow string and the shape it is to be applied are retrieved from their respective registers, and the Rhino GUIDs of the result of the flow are returned (and hidden from the Rhino Viewport). If the user wishes to have the Rhino GUIDs be visible, set the last value ('hide') to False.

Syntax

```
sgi.apply_flow(flowName, shapeName, refPoint = r.Geometry.Point3d(0,0,0), hide
= True)
```

Parameters

- flowName: Name of flow inside flow register to be applied onto shape
- shapeName: Name of shape inside form register that the flow will be applied onto
- refPoint: Reference point to serve as 'origin' point for the final shape after the flow; the shape will be moved to this point; the default origin point is (0,0,0)
- hide: Boolean True/False value if output Rhino GUIDs are to be hidden; True (default) - hides Rhino objects from viewport, False - leaves Rhino objects in viewport

Returns

- finalShapeRhino: Rhino GUIDs (hidden from Viewport)
- None is returned if the flow sequence is unsuccesful

Warnings & Errors

- KeyError: Shape name or flow name does not exist in their respective registers
- TypeError: Reference point input is a not tuple/list of three numbers, a Rhino point geometry or a GUID OR the inputs for shape name and/or flow name are not text strings
- Error: If the number of elements in the reference point tuple/list is not exactly three numbers, or if 'rhino_shapes_c' is not present in the sort register, i.e. sort types in sortal library has not yet been set up
- Warning: If the flow sequence could not be carried out successfully

Example

```
flowName = 'flow_1'
shapeName = 'shape_1'
refPoint = refPoint = r.Geometry.Point3d(0,0,0)
new_shape = sgi.apply_flow(flowName, shapeName, refPoint, hide = False)
```

The contents of ‘new_shape’ will be a list of Rhino GUIDs corresponding to the drawing of the shape in the Rhino viewport. Because ‘hide’ is set to False, the resulting drawing will be visible in the workspace.

check_precision

Returns the precision of comparison (number of decimal places) inside the sortal library (an integer value)

Syntax

```
sgi.convert_to_agnostic(shapeRhino = [], classification = None, descriptions =
'', refPoint = Rhino.Geometry.Point3d(0,0,0))
```

Parameters

- None

Returns

- Integer value corresponding to number of decimal places the sortal library applies onto numbers

Warnings & Errors

- None

Example

```
sgi.check_precision()
```

This returns an integer.

convert_shape

Converts the contents of sortalShape (a sortal shape data structure) to their counterpart geometric sorts in targetSort. All external, standalone descriptions as well as attributes attached to the geometries are carried over, so long as they are reflected in targetSort

Syntax

```
sgi.convert_shape(sortalShape = None, targetSort = None, newName = None)
```

Parameters

- sortalShape: Text string. The name of the shape or the sortal shape data structure to be converted to the target sort in targetSort; this may have a primitive, attribute or disjunctive sort type; if the input has a compound sort type and there is only one functioning geometric disjunctive sort inside the compound sort, then the shape corresponding to this functioning geometric disjunctive sort is extracted and converted to the target sort
- targetSort: Text string. The name of the target sort type or the sort type data structure that is the end goal of the conversion; this may be a primitive, attribute or disjunctive sort; if the target sort type is a compound sort, an error will be raised.

To illustrate possible conversion cases:

a. sortalShape: 'shape_1' with sort type meta3D, targetSort: metaP2D; this will convert the geometric individuals of 'shape_1' to P2D geometric individuals and retain any attribute types carried over to metaP2D. However, if, for example, meta3D has line segments with labels, while metaP2D has primitive line segments only, then the labels attached to line segments in 'shape_1' will not be maintained.

b. sortalShape: 'line1' with sort type lineSegP3D, targetSort: lineSeg2D; this will construct the shape 'line1' as non-parametric, 2D line segments. The same behavior regarding attribute maintenance applies here as in (a).

c. Sort type: ellipticalArc3D, targetSort: 'P3D'; this is not possible, as elliptical arcs are not enabled for parametric behavior in the sortal library.

d. Sort type: meta3D (disjunctive, including pointN3D), targetSort: pointP2D; in this case, only the pointP3D form will be retrieved from meta3D and turned to its non-parametric 2D counterpart and outputted.

e. Sort type: rhino_shapes_c, with only one functioning geometric disjunctive sort (N3D) because the other sort type present is a dummy disjunctive sort composed of two description sorts.

- newName: The name of the converted shape that will be registered in the shape and the form register (text string); if none is given, then the name of the original shape and the target sort are concatenated to create a name for the shape

Returns

- newSortalShape: Returns converted sortal shape with geometric individuals reflecting attribute structure, and non-parametric/parametric behavior of targetSort, if successful
- False: if unsuccessful

Warnings & Errors

- TypeError:
  - If the input for ‘sortalShape’, ‘targetSort’ or ‘newName’ is not a text string,
  - If the input for ‘targetSort’ is a compound sort
- KeyError:
  - If the sortal shape or target sort does not exist in the form/sort register
  - If 'rhino_shapes_c' is not present in the sort register, i.e. sort types in sortal library has not yet been set up
- Warning: If the sort type of sortalShape is already the same as the target sort

Example

The sort type to be used in conversion is first retrieved from the sort register.

```
new_sort = sort.register[‘P3D’]
```

This retrieves a shape with a non-parametric 3D disjunctive sort (housed under the compound sort ‘rhino_shapes_c’) and converts it to the disjunctive sort ‘P3D’. The new shape is given the name ‘shape_2’ in the form register.

```
new_sort = sgi.convert_sort(‘shape_1’, new_sort, ‘shape_2’)
```

convert_sort

Determines and constructs the counterpart sort type contained in sortalStructure, according to the target sort's specified parametric-dimension combination (e.g., non-parametric 2D - N2D, non-parametric 3D - N3D, parametric 2D - P2D, parametric 3D - P3D)

Syntax

```
sgi.convert_to_agnostic(shapeRhino = [], classification = None, descriptions =
'', refPoint = Rhino.Geometry.Point3d(0,0,0))
```

Parameters

- sortalStructure: Name (text string) or actual data structure of sort type to be converted over to new parametric-dimension structure; this can be a disjunctive, attribute or primitive sort
- targetSort: Text string of target sort type; this may be 2D or 3D, non-parametric or parametric
- For example:
- a. sortalStructure: meta3D, targetSort: 'P2D'; this will determine and construct the sort type that mirrors the geometry and their attributes in meta3D, but as parametric 2D geometric sorts
- b. sortalStructure: lineSegP3D, targetSort: 'N2D'; this will construct the 2D line segment geometric sort and bring over any attributes present in lineSegP3D
- c. sortalStructure: ellipticalArc3D, targetSort: 'P3D'; this is not possible, as elliptical arcs are not enabled for parametric behavior in the sortal library

Returns

- targetSortType: New sort type corresponding to targetSort variation, that mirrors base geometry in sortalStructure, but converts them to their counterparts as in targetSort, if successful. If the sortal structure being converted is a disjunctive sort, it is automatically included in 'rhino_shapes_c'; otherwise, it is only stored in the sort register
- None, if unsuccessful

Warnings & Errors

- TypeError: Input for sortalStructure is neither a text string nor a sort type data structure, or input for targetSort is not a text string
- KeyError:
  - If input for sortalStructure is a text string that is not present in the sort register
  - if 'rhino_shapes_c' is not present in the sort register, i.e. sort types in sortal library has not yet been set up

Example

```
original_sort = sort.register['P3D']
new_sort = sgi.convert_sort(original_sort, 'N2D')
```

This returns a new sortal structure based on the sort types in 'original_sort', but enabled with non-parametric 2D sort types

create_flow

Creates a flow from the inputs flowName (name of the flow object), flowDesc (description of the flow object), and flowSeq (text string sequence of rule names and ordering, looping and execution instructions)

Syntax

```
sgi.create_flow(flowName, flowDesc = '', flowSeq = '')
```

Parameters

- flowName: Flow name (text string); this will be used to retrieve the flow object from the flow register

- flowDesc: Flow description (text string, optional); this is used to describe the flow object
- flowSeq: Flow sequence (text string); this text string contains the sequence of rules as well as their ordering, looping and execution instructions

Returns

- newFlow: The flow sortal data structure is returned if the flow is successfully created

Warnings & Errors

- TypeError: If the input for flowName, flowDesc, or flowSeq is not a text string
- ValueError: If the input for flowName or flowSeq is empty, or a rule name specified in flowSeq is not present in the rule register
- KeyError:
  - If a flow object with the same data as flowName already exists in the flow register
  - if 'rhino_shapes_c' is not present in the sort register, i.e. sort types in sortal library has not yet been set up

Example

```
flowName = 'f1'
flowDesc = 'test flow'
flowSequence = 'rul1 (rul2 rul3{2})*'
sgi.create_flow(flowName, flowDesc, flowSequence)
```

create_rule

Creates a new rule instance; it takes as input the rule name, rule description, LHS and RHS agnostic shapes/ shape names/Rhino geometries, predicates (for LHS) and directives (RHS). Since 'create_rule' is linked to the sortal library, the act of creating a rule object checks if a rule with the same name as the new rule being created already exists. If yes, then the rule is not created and the function is exited.

Syntax

```
sgi.create_rule(ruleName, ruleDesc, lhs, rhs, predLHS = None, dirRHS = None,
prnt = False)
```

Parameters

- ruleName: Name of rule to be created (text string)
- ruleDesc: Description of rule to be created (text string)
- lhs: Name of shape to become LHS of rule (text string)
- rhs: Name of shape to become RHS of rule (text string)
- predLHS: Agnostic dictionary of predicates (optional); usually, this is obtained from sgi.create_shape
- dirLHS: Agnostic dictionary of directives (optional); usually, this is obtained from sgi.create_shape
- prnt: Boolean value, corresponding to whether to print if rule was created successfuly, and as to which predicates/directives were added to the rule successfully

Returns

- Sortal rule data structure, if successful
- None, if unsuccessful

Warnings & Errors

- MessageBox: If a rule with the same name already exists in the sortal register, then the user is prompted for whether they would like to overwrite the pre-existing rule or to give the rule to be created a different name or to exit the create_rule function; alternatively, if the geometry within the rule object's LHS is insufficient
- TypeError: If the input for rule name or rule description or LHS shape name or RHS shape name is not a text string
- KeyError:
    - If the shapes corresponding to the inputs for the LHS and RHS shape names are not present in the form register
    - If 'rhino_shapes_c' is not present in the sort register, i.e. sort types in sortal library has not yet been set up

Example

```
ruleName = 'rule_1'
ruleDesc = 'test rule'
lhsName = 'shape_lhs'
rhsName = 'shape_rhs'
newRule = sgi.create_rule(ruleName, ruleDesc, lhsName, rhsName, predicates,
directives)
```

The shapes corresponding to the names ‘shape_lhs’ and ‘shape_rhs’ are retrieved from the form register and are used to create the rule. The function also stores the newly created rule instance in the sortal library. A rule object can be retrieved later by using the property of the sortal library import ‘rule’, called ‘register’:

```
ruleObject = rule.register[ruleName]
```

This returns the sortal rule object.

create_shape

Creates a sortal shape from a collection of Rhino GUIDs. It names the shape according to the input for the shapeName variable in the sortal library. It returns the predicate/directive dictionaries.

Syntax

```
sgi.create_shape(shapeName = None, shapeData = None, descriptions = '',
classification = None, refPoint = rg.Point3d(0,0,0), prnt = False)
```

Parameters

- shapeName: Name of shape (text string)
- shapeData: List of Rhino GUIDs that will compose the sortal shape
- descriptions: Text string of descriptions to include in the sortal shape, e.g.

'label1@("A2", 1, 1234);("A3", 4, 1234)|label2@("A1", 1, 1234)|label2@("A9", 8, 1234)' where there are two description types, 'label1' (followed by an ampersand “@”; 2 individuals separated by a semicolon “;”), and 'label2' (2 individuals); declaration of different description types and their individuals is separated by a vertical dash “|”

- classification: Name of target disjunctive sort type (text string); this sort type must already be present in the sort register under the compound sort 'rhino_shapes_c'; this input may be left blank if there is only one active geometric disjunctive sort inside ‘rhino_shapes_c’ and the other sort is a dummy sort composed of two description sort types
- refPoint: Reference point to serve as 'origin' point for shape; the default origin point is (0,0,0)
- prnt: Boolean value (True/False); True - prints the resulting sortal shape according to the output of the 'printE' command in the sortal library; False (default) - does not print anything

Returns

- predicates: Predicates dictionary
- directives: Directives dictionary

Warnings & Errors

- TypeError: If shapeName input is not a text string, or if shapeData input is not a list of Rhino GUIDs, or if shapeData has any elements that are not Rhino GUIDs inside
- KeyError:
  - If the result Rhino GUIDs to Agnostic Dictionary conversion is empty
  - if 'rhino_shapes_c' is not present in the sort register, i.e. sort types in sortal library has not yet been set up
- Warning: If descriptions input is not a text string

Example

```
shapeLHS = rs.GetObjects('Select LHS shape')
LHSdesc = 'segmentCount@t|platform@(name?="bp", count?>0, leng, wid, layer,
ad_type, ad_count)'
pt1 = rs.GetObject('Select a reference point')
predicates, directives = sgi.create_shape ('shapeL', shapeLHS, LHSdesc, 'P2D',
pt1)
```

default_precision

Sets precision of comparison (number of decimal places) inside sortal library back to default (5).

Syntax

```
sgi.default_precision()
```

Parameters

- None

Returns

- Integer value corresponding to number of decimal places the sortal library applies onto numbers (in this case, 5)

Warnings & Errors

- None

Example

```
sgi.default_precision()
```

draw_rule

'draw_rule' draws the sides of a rule in the Rhino workspace. This function supposes that the rule is already present in the rule register.The rule sides are drawn apart from each other, with the space between them dictated by the value of 'fixed_factor' or the size of the bounding box of the LHS GUIDs. When a reference point is given (tuple/list/vector/point), the rule shapes are moved to the reference point. The default reference point is the origin (0,0,0).

Syntax

```
sgi.draw_rule(ruleName, refPt = r.Geometry.Point3d(0,0,0)
```

Parameters

- ruleName: Name of rule object to draw from inside the rule register (text string)
- refPt: Reference point (optional - tuple / list of integers or floats / vectors / points (3Ds)); relocates
- drawings of rule sides with reference point serving as the origin; default origin is (0,0,0)

Returns

- List of lists in the form [lhsRhino, rhsRhino] or [lhsMoved, rhsMoved]: List of lists of Rhino Geometry [[lhs shape GUIDs], [rhs shape GUIDs]]; an empty list is returned if the rule cannot be found in the rule register

Warnings & Errors

- KeyError:
  - If rule name does not exist in the rule register
  - If 'rhino_shapes_c' is not present in the sort register, i.e. sort types in sortal library has not yet been set up
- Warning: Rule name input is not a text string

Example

```
ruleName = 'rule_1'
pt = rs.GetObject('Select reference point for rule drawing')
sgi.draw_rule(ruleName, pt) #
```

draw_shape

Draws the shape in the Rhino workspace. This function assumes that the shape is already present in the form register. When a reference point is given (tuple/list/vector/point), the shape GUIDs are moved to the reference point. The default reference point is the origin. The name of the shape must already be present in the form register for this function to work.

Syntax

```
sgi.draw_shape(shapeName, refPt = r.Geometry.Point3d(0,0,0))
```

Parameters

- shapeName: Name of shape object to draw from inside form register (text string)
- refPt: Reference point (optional - tuple / list of integers or floats / vectors / points (3Ds)); relocates drawing of shape with reference point serving as the origin; default origin is (0,0,0)

Returns

- shapeNewRhino: List of Rhino Geometry [shape GUIDs]; this is returned as an empty list the shape does not exist in the shape register

Warnings & Errors

- Warning: Shape with the inputted shape name does not exist in the form register
- TypeError: Shape name input is not a text string
- KeyError: If ‘rhino_shapes_c’ is not present in the sort register, i.e. sort types in sortal library has not yet been set up

Example

```
shape_name = ‘shape_1’
shape_guids = sgi.draw_shape(shape_name)
```

extract_shape

Searches for a shape inside the input sortal shape with the same sort type and sort level as the target sort type input. For example, if a shape containing only line segments needs to be retrieved and the corresponding sort type to these line segments is called ‘lineSegment-A’, then the target sort type input is ‘lineSegment-A’ and a shape with the sort type ‘lineSegment-A’ is returned if it exists inside the input sortal shape.

Syntax

```
sgi.extract_shape(sortalShape, targetSortType, shapeName = None)
```

Parameters

- sortalShape: Name of shape (text string) or sortal shape data structure
- targetSortType: Name of sort type (text string) or sort type data structure
- shapeName: Name of extracted shape; this is used to register the extracted shape in the form register, if it is successfully extracted

Returns

- result: Sortal shape (actual sortal shape data) corresponding to targetSortType and registered in the form register, if successful
- None, if no such shape with the same sort as the target sort type can be found inside the input

Warnings & Errors

- TypeError: If the input for sortalShape / targetSortType (/shapeName) is neither a text string or a sortal shape data structure/sort type data structure
- KeyError:
    - If sortalShape or targetSortType is a text string input, this error is raised if they do not exist in the form/sort register
    - If ‘rhino_shapes_c’ is not present in the sort register, i.e. sort types in sortal library has not yet been set up
- Warning: If sortalShape does not contain any sort type equal to targetSortType

Example

```
newShape = sgi.extract_shape(‘shape_1’, ‘lineSegment’, ‘lineSegments’)
```

This returns the sortal form with the disjunctive sort ‘lineSegment’ and stores it in the form register under the name ‘lineSegments’.

find_rule_appns

Finds the rule applications of a rule on a shape (an optional subshape input may be used to limit the number of matches found within the shape). It returns a list of lists of GUIDs, with each list corresponding to the result of the rule application on the shape as drawn in the Rhino workspace. By default, these drawings are hidden from the Rhino viewport.

Syntax

```
sgi.find_rule_appns(chosenRule, shape, subshape = None, refPt =
r.Geometry.Point3d(0,0,0), layerName = 'Default', hide = True, prnt = False,
shapeIds = None)
```

Parameters

Required

- chosenRule: Rule name as text string
- shape: Name of main shape as text string as recorded in sortal library's form register pertaining to shape

Optional

- subshape: Name of subshape as text string as recorded in sortal library's form register pertaining to subshape
- refPt: GUID of point, tuple, or Rhino Geometry point of reference point which will serve as the 'origin' with which shapes will be plotted in respect to (in principle, a vector)
- layerName: Name of target layer as text string within Rhino workspace where the GUIDs will be drawn onto

- hide: Boolean value True/False; True (default) – hides Rhino GUIDs from viewport; False – keeps Rhino GUIDs visible in viewport
- prnt: Boolean value True/False; True - prints out description individuals as input-ready text string (to create_shape) as well as printE form of shape after rule application; False (default) - nothing is printed
- shapeIds: List of Rhino GUIDs to clear after new shapes have been drawn

Returns

- appnGeometry: List of lists of Rhino GUIDs (each list correspond to a shape after a certain rule application), if successful; these geometries are hidden from the viewport, by default
- None, if unsuccessful

Warnings & Errors

- TypeError: If initial shape or subshape is not a list of GUIDs or a text string or is empty, or if the input for chosenRule is not a text string
- KeyError:
  - If initial shape or subshape input's name is not present in the form register
  - If rule name is not present in the rule register
  - If 'rhino_shapes_c' is not present in the sort register, i.e. sort types in sortal library has not yet been set up
- Warning: If subshape is not part of shape -> exits function and returns False

Example

```
chosenRule = 'rule_1'
shape = 'shape_1'
subshape = 'lineSegments'
ruleAppns = sgi.find_rule_appns(chosenRule, shape, subshape)
```

The function returns a list of lists of Rhino GUIDs, each element list corresponding to a rule application. By default, the shapes are drawn on top of one another and hidden from the Rhino viewport. The function 'move' may be used to space the results out from one another, and rs.ShowObjects may be used on the elements inside 'ruleAppns' to show the shapes.

get_rule_lhs

Returns the sortal data structure corresponding to the LHS in the given rule input, or if rhino is set to True, creates a drawing of the rule LHS in the Rhino viewport and returns the corresponding list of Rhino GUIDs.

Syntax

```
sgi.get_rule_lhs(name, rhino = True)
```

Parameters

- name: Name of target rule object

- rhino: Boolean value, indicates if side of rule should be drawn and returned as list of Rhino GUIDs; True (default) - returns Rhino GUIDs (hidden from viewport), False - returns sortal form corresponding to target rule side

Returns

- Rule object description (text string); LHS or RHS (as sortal shape or as hidden Rhino GUIDs generated in original location of rule), if successful
- None, if not successful

Warnings & Errors

- TypeError: If rule name input is not a string
- KeyError:
    - If rule name input is not present in the rule register
    - If 'rhino_shapes_c' is not present in the sort register, i.e. sort types in sortal library has not yet been set up

Example

```
lhsGUIDs = sgi.get_rule_lhs('rule_1')
```

This function call returns a list of GUIDs corresponding to the drawing of the LHS in the Rhino viewport.

OR

```
lhsShape = sgi.get_rule_lhs('rule_1', False)
```

This returns the sortal data structure corresponding to the LHS in 'rule_1'.

get_rule_rhs

Returns the sortal data structure corresponding to the RHS in the given rule input, or if rhino is set to True, creates a drawing of the rule RHS in the Rhino viewport and returns the corresponding list of Rhino GUIDs.

Syntax

```
sgi.get_rule_rhs(name, rhino = True)
```

Parameters

- name: Name of target rule object
- rhino: Boolean value, indicates if side of rule should be drawn and returned as list of Rhino GUIDs; True (default) - returns Rhino GUIDs (hidden from viewport), False - returns sortal form corresponding to target rule side

Returns

- Rule object description (text string); LHS or RHS (as sortal shape or as hidden Rhino GUIDs generated in original location of rule), if successful
- None, if not successful

Warnings & Errors

- TypeError: If rule name input is not a string

- KeyError:
    - If rule name input is not present in the rule register
    - If ‘rhino_shapes_c’ is not present in the sort register, i.e. sort types in sortal library has not yet been set up

Example

```
rhsGUIDs = sgi.get_rule_rhs(‘rule_1’)
```

This returns a list of GUIDs corresponding to the drawing of the RHS in the Rhino viewport.

OR

```
rhsShape = sgi.get_rule_rhs(‘rule_1’, False)
```

This returns the sortal data structure corresponding to the RHS in ‘rule_1’.

get_rule_description

Returns the description text of the given rule object, if the rule exists.

Syntax

```
sgi.get_rule_description(name)
```

Parameters

- name: Text string; name of target rule object

Returns

- Text string of rule object description, if successful
- None, if not successful

Warnings & Errors

- TypeError: If rule name input is not a string
- KeyError:
    - If rule name input is not present in the rule register
    - If ‘rhino_shapes_c’ is not present in the sort register, i.e. sort types in sortal library has not yet been set up

Example

```
descriptionText = sgi.get_rule_description(‘rule_1’)
```

maximalize

Accepts the name of a shape object or a list of Rhino GUIDs and returns the maximalized sortal shape or its corresponding list of Rhino GUIDs.

Syntax

```
sgi.maximalize(shape, target = None, rhino = False, hide = False, delete =
False)
```

Parameters

- shape: Shape data in the form of a list of Rhino GUIDs (target disjunctive sort for maximalized must be provided) or the shape name as a text string
- target: Target disjunctive sort for the shape to be maximalized (text string), this is necessary if the input is a list of Rhino GUIDs
- rhino: Boolean value (True/False); True - returns list of Rhino GUIDs corresponding to maximalized shape; False (default) - returns agnostic form, please note that if Rhino GUIDs are given as input, then Rhino GUIDs will be returned
- hide: Boolean value (True/False); True - hides Rhino geometry output from viewport; False (default) - leaves Rhino geometry output visible in viewport
- delete: Boolean value (True/False); if the inputs are a list of Rhino GUIDs, then True deletes the inputs, and False (default) leaves the inputs still in the Rhino viewport

Returns

- maxShape: If a shape name is inputted, then it returns the maximalized sortal data structure of the shape unless the variable rhino is set to True

If a list of Rhino GUIDs is inputted, then it returns a list of Rhino GUIDs corresponding to the maximalized form by default

Warnings & Errors

- TypeError: If type of input is invalid (i.e. not a list of Rhino GUIDs or a shape name text string)
- ValueError: If input is an empty list or if shape name doest not exist in the form register
- KeyError: If 'rhino_shapes_c' is not present in the sort register, i.e. sort types in sortal library has not yet been set up

Example

```
shapeName = 'shape_1'
maximalShape = sgi.maximalize(shapeName)
```

Forgoing the target sort type is allowed so long as there is only one active geometric disjunctive sort within 'rhino_shapes_c'. This function call will return the maximalized sortal data structure corresponding to 'shapeName' in the form register.

OR

```
maximalShapeGUIDs = sgi.maximalize(shapeName, rhino = True, hide = True)
```

This function call will return a list of Rhino GUIDs, but will hide them from the Rhino viewport, as 'hide'

move

Spaces out the shapes in a list based on either the translationVec variable (if a valid input is giving) or based on the bounding box dimensions multiplied by the value of sc.sticky in vertical order. The Rhino GUIDs of the moved shape(s) are returned.

Syntax

```
sgi.move(shapes,  alignment = False, translationVec = False)
```

Parameters

- shapes: List of Rhino GUID lists or list of Rhino GUIDs, translation vector (optional)
- alignment: Vector that will be reduced to unit vector to determine axis of moved shapes (e.g. vertical ascending or horizontal going to the right, etc.)
- translationVec: Single vector or list of vectors, that serve.s as the spacing reference vector/s between shapes. The space between the first and second shapes will follow the first vector element of the list, the space between the second and third shapes will follow the second vector element of the list, and so on.

Returns

- movedShapes: List of Rhino GUID lists after moving the geometries
- None, if unsuccessful

Warnings & Errors

- TypeError: If data type of alignment and/or translation vector is not a tuple or list of three numbers or a Vector3d object
- KeyError: If 'rhino_shapes_c' is not present in the sort register, i.e. sort types in sortal library has not yet been set up
- Warning: If the input for the reference point is invalid (not a tuple or a list of three numbers or a Rhino point geometry)

Example

```
chosenRule = 'rule_1'
shape = 'shape_1'
subshape = 'lineSegments'
ruleAppns = sgi.find_rule_appns(chosenRule, shape, subshape)
sgi.move(ruleAppns,  alignment = r.Geometry.Vector3d(1,0,0), translationVec =
r.Geometry.Vector3d(10,0,0))
```

overwrite_sdl

Stores rules, flows and an initial shape in an SDL file based on the variable fileName, by accepting their name references, and retrieving the relevant sortal objects. If an SDL file with the same name already exists, then the user is prompted if they want to overwrite the file or give a new value to fileName. The input for the file name of the intended SDL file may also include the address of the intended location.

Syntax

```
sgi.overwrite_sdl(fileName, rules = [], shapes = [], flows = [], append =
False, prnt = True)
```

Parameters

- fileName: Text string of file name (for .sdl file; this may include the intended location's full address)
- rules: List of sortal rule object names (optional, this may be an empty list)
- shape: One shape object name (optional, this may be ignored)
- flows: List of sortal flow object names (optional, this may be an empty list)

Note: If including the full address of the target SDL file, the following format must be observed:
'C:\\Users\AKIDRIBM\\AppData\\Roaming\\Grasshopper\\Libraries\\source_code_gh_dev\\'+ <sdl file name> +'.sdl'
where every slash is doubled.

Returns

- If successful, an SDL file of name 'fileName' in the intended location with rule(s) and shape stored in it is created, and True is returned. Otherwise -
- False: Unsuccessful operation

Warnings & Errors

- Warning: If a rule, form (shape) or flow does not exist in their corresponding register, or if the SDL file with the target file name does not exist, or if the SDL file name does not have the '.sdl' extension
- TypeError: If the input for rules, shapes, flows are not lists, or if there is an invalid input (not a text string) within the lists
- KeyError:
  - If a rule, form or flow name within a list does not exist in the corresponding register
  - If 'rhino_shapes_c' is not present in the sort register, i.e. sort types in sortal library has not yet been set up

Example

```
fileName = 'rules_3D.sdl'
rules = ['WHL-1a']
shapes = ['shape_1']
flows = ['f1', 'f2']
sgi.overwrite_sdl(fileName, rules, shapes, flows, append = True, prnt = True)
```

part_of

Checks if a subshape agnostic tuple is part of a possibly larger shape agnostic tuple

Syntax

```
sgi.part_of(subshape, shape)
```

Parameters

- subshape: List of Rhino GUIDs or name of subshape

- shape: List of Rhino GUIDs or name of shape

Returns

- True: If the subshape is part of the shape
- False: If the subshape is not part of the shape

Warnings & Errors

- KeyError:
    - If either shape or subshape name is not available in the form register
    - If ‘rhino_shapes_c’ is not present in the sort register, i.e. sort types in sortal library has not yet been set up
- TypeError: If either shape or subshape input is not a text string

Example

```
result = sgi.part_of(subshapeAgnostic, shapeAgnostic)
```

read_sdl

Opens an SDL file and merges any compound sorts inside with the prevailing iteration of the compound sort 'rhino_shapes_c' in the sort register. It also updates any rules and shapes with outdated sort types not set to the prevailing compound sort 'rhino_shapes_c' and rewrites the SDL file to ensure compatibility with the API.

Syntax

```
sgi.read_sdl(fileName)
```

Parameters

- fileName: SDL file name; if including the full address of the target SDL file, the following format must be observed:
  ```
  'C:\\Users\AKIDRIBM\\AppData\\Roaming\\Grasshopper\\Libraries\
  \source_code_gh_dev\\'+ <sdl file name> +'.sdl'
  ```
  where every slash is doubled.

Returns

- True: if successful
- False: if unsuccessful

Warnings & Errors

- KeyError: If the file name does not exist

Example

```
sgi.read_file(‘rules_3D.sdl’)
```

OR

```
sgi.read_sdl(‘'C:\\Users\AKIDRIBM\\AppData\\Roaming\\Grasshopper\\Libraries\
\source_code_gh_dev\\rules_3D.sdl')
```

redraw

'redraw' redraws an inputted list of Rhino GUIDs with their corresponding tags, predicates, directives, labels and descriptions. The original Rhino GUIDs inputted into the function are deleted, along with any text dots that serve only as tag data and predicate/directive information holders.

Syntax

```
sgi.redraw(rhinoObject, targetSort = 'N3D')
```

Parameters

- rhinoObject: List of Rhino GUIDs to redraw
- targetSort: Name of target disjunctive sort type to classify geometry under (necessary, default is non-parametric 3D disjunctive sort 'N3D')

Returns

- objectIds: List of Rhino GUIDs corresponding to redrawn geometry; the function also deletes theinputted list of Rhino GUIDs, generates new list of GUIDs as outputs

Warnings & Errors

- TypeError:
  - If any elements inside rhinoObject are not Rhino GUIDs
  - If the input for rhinoObject is not a list of Rhino GUIDs
- Error: If the sortal library has not yet been set up, i.e. if the compound sort 'rhino_shapes_c' does not yet exist in the sort register

Example

```
shape_list = rs.GetObjects('Select shape to be redrawn')
shape_list_redrawn = sgi.redraw(shape_list)
```

The target sort type input may be forgone when using ‘redraw’ if there is only one active geometric sort type in the compound sort ‘rhino_shapes_c’. However, in the case of multiple active geometric sort types in ‘rhino_shapes_c’, it is advised that a target sort type for the shape be inputted, as otherwise, the ‘redraw’ function will base the sort type of the shape on the first disjunctive sort it encounters.

save_sdl

Stores rules, flows and an initial shape in an SDL file based on the file name input, by accepting their name references, and retrieving the relevant sortal objects. If an SDL file with the same name already exists, then the user is prompted if they want to overwrite the file or give a new value to the file name input. The input for 'fileName' can also include the address of the intended location.

Syntax

```
sgi.save_sdl(fileName = 'new.sdl', rules = [], shape = '', flows = [])
```

Parameters

- fileName: Text string of file name (for .sdl file; this may include the intended location's full address)

- rules: List of sortal rule object names (optional, this may be an empty list)
- shape: One shape object name (optional, this may be ignored)
- flows: List of sortal flow object names (optional, this may be an empty list)

Returns

- True: If successful; an SDL file of name 'fileName' in the intended location with rule(s) and shape stored in it is created
- False: If unsuccessful

Warnings & Errors

- MessageBox: If an SDL file with the same file name already exists in the location; the user is prompted if they would like to overwrite the file or give the current save_sdl function a new file name input
- Warning: If the rule or shape or flow name does not exist in its corresponding register
- TypeError:
  - If the input for rules or flows is not a list or the input for shape is not a string
  - If there are any non-text inputs within the input for rules or flows
- KeyError:
  - If the shape name does not exist in the form register
  - If 'rhino_shapes_c' is not present in the sort register, i.e. sort types in sortal library has not yet been set up

Example

```
sgi.save_sdl(fileName = 'new.sdl', ['rul_1', 'rul_2'], ['shape_1', 'shape_2'],
['f1', 'f2'])
```

set_flow_description

Changes the description of a flow object in the sortal flow register.

Syntax

```
sgi.set_flow_description(flowName, newDesc)
```

Parameters

- flowName: Name of flow object (text string)
- newDesc: New description for flow object (text string)

Returns

- True: If successful
- False: If successful

Warnings & Errors

- KeyError: If the original flow name does not exist in the sort register

- Error: If 'rhino_shapes_c' is not present in the sort register, i.e. sort types in sortal library has not yet been set up
- Warning: If the input for the new description or the original flow name is not a text string

Example

```
sgi.set_flow_description('f1', 'new description text')
```

set_flow_name

Changes the name of a flow object in the sortal flow register. However, if the new flow name matches that of a pre-existing flow, this will cause an error. If successful, the previous flow name is deleted from the sortal library's flow register and is free to use for new flows.

Syntax

```
sgi.set_flow_name(oldName, newName)
```

Parameters

- oldName: Name of target flow object to be altered (text string)
- newName: New name of target flow object (text string)

Returns

- True: If successful
- False: If unsuccessful

Warnings & Errors

- TypeError: If shapeRhino is not a list of Rhino GUIDs OR if 'descriptions' is not None or a text string
- Warning: If conversion was unsuccessful
- KeyError: If 'rhino_shapes_c' is not present in the sort register, i.e. sortal library has not yet been set up

Example

```
sgi.set_flow_name('f1', 'flow_2')
```

set_precision

Sets precision of comparison (number of decimal places) inside sortal library (integer value).

Syntax

```
sgi.set_precision(new)
```

Parameters

- new: Integer value referring to number of decimal places to indicate precision

Returns

- True: if successful in setting new precision value
- False: if unsuccessful

Warnings & Errors

- None

Example

```
sgi.set_precision(10)
```

The precision in the sortal library is now 10 decimal places.

set_rule_description

Changes the description of a rule object in the sortal rule register.

Syntax

```
set_rule_description(ruleName, newDesc)
```

Parameters

- ruleName: Name of rule object (text string)
- newDesc: New description for rule object (text string)

Returns

- True: If successful
- False: If unsuccessful

Warnings & Errors

- KeyError:
  - If the original rule name does not exist in the sort register
  - If 'rhino_shapes_c' is not present in the sort register, i.e. sort types in sortal library has not yet been set up
- Warning: If the input for the new description or the original rule name is not a text string

Example

```
newRuleDescription = 'new rule description'
sgi.set_rule_description('rul_1', newRuleDescription)
```

set_rule_name

Changes the name of a rule object in the sortal rule register; however, if the new rule name matches that of a pre-existing rule, this will cause an error. If successful, the previous rule name is deleted from the sortal library's rule register and is free to use for new rules.

Syntax

```
sgi.set_rule_name(oldName, newName)
```

Parameters

- oldName: Name of target rule object to be altered (text string)
- newName: New name of target rule object (text string)

Returns

- True: If successful
- False: If unsuccessful

Warnings & Errors

- KeyError:
  - If the original rule name does not exist in the sort register
  - If the new rule name already exists in the sort register or if 'rhino_shapes_c' is not present in the sort register, i.e. sort types in sortal library has not yet been set up
- Warning: If the input for the original rule name or the new rule name is not a text string

Example

```
sgi.set_rule_name('rul1', 'rule_1')
```

set_shape_name

Changes the name of a shape in the sortal form register.

Syntax

```
sgi.set_shape_name(shapeName, newShapeName)
```

Parameters

- shapeName: Name of shape object to be changed (text string)
- newShapeName: New name of target shape object (text string)

Returns

- True: If successful
- False: If unsuccessful

Warnings & Errors

- KeyError: If 'rhino_shapes_c' is not present in the sort register, i.e. sort types in sortal library has not yet been set up
- Warning: If the input for the original shape name or the new shape name is not a text string

Example

```
sgi.set_shape_name('shape_1', 'shape_1_subshape')
```

Rhino Methods

The functions listed in this section are used to add or remove information relating to attributes (weight, color, descriptions), predicates and directives to Rhino geometry. These are collected alongside geometric information when Rhino GUIDs are converted to sortal shape data structures.

For all methods to do with adding labels, descriptions, predicates and directives to the geometry, any specific predicate or directive tag or label or description text already existing in the UserText of the selected geometry is not added again to the data of the geometry.

Import notation

The use of 'rm' in this section refers to the class 'rhino_methods', which contains all the methods listed in this section and is imported in code snippet examples as follows:

```
from sortalgi import rhino_methods as rm
```

The terms 'key name' and 'tag name' refer to the key name of the Rhino geometry, that is used to recognize it in the sortal library when adding predicate and directive data.

Summary of all methods

NAME	PURPOSE
add_bound_line	Adds bound line predicate tag to line segment/polyline; indicated as 'bound' within predicates text ('#( )')
add_color	Adds color to Rhino Object (one or several objects may be selected) using RGB values; this color is not visible in Print Preview Mode
add_description	Adds description to Rhino Object's user text, visualized as text dot with extra 'd( )' enclosing text in the visual text dot assigned to geometry
add_distance	Adds distance directive tag to geometries; first selected geometry may be found in either LHS or RHS, second selected geometry must be found in RHS; range of values for length is inputted to function call; directive is indicated as 'distance' within the directives text ('#( )') in the visual text dot assigned to geometry
add_enum	Adds enumerative value to Rhino Object's user text, not visualized
add_label	Adds label data to Rhino Object's user text, visualized as text dot with extra 'l( )' enclosing text in the visual text dot assigned to geometry
add_longest_line	Adds longest-line predicate tag to line segment/s; indicated as 'longest' within predicates text ('#( )') in the visual text dot assigned to geometry
add_max_line	Adds max-line predicate tag to line segment/s, indicated as ‘max’ within predicates text ('#( )') in the visual text dot assigned to geometry
add_no_label	Adds no-label predicate tag to geometry to constrain matching to geometries without label attributes. This is indicated as 'no_label' within predicates text ('#( )') in the visual text dot assigned to geometry
add_normal	Adds normal directive tag to line segments (if 2D) or plane segments (if 3D). This is indicated as ‘normal ^ <group_name>’ within the directives text ('#( )') in the visual text dot assigned to geometry
add_point_on_line	Adds point-on-line directive tag to line segment/polyline, with the upper and lower bounds for the range of segment values for point placement inputted to function call; this is visually indicated as 'point_on_line ^ <group_name>' within the directives text ('#( )')
add_shortest_line	Adds shortest line predicate tag to line segment/polyline; indicated as 'short' within predicates text ('#( )') in the visual text dot assigned to geometry
add_void	Adds void predicate tag to collection of points/text dots/line segments/a single plane segment that form a closed polygon; indicated as 'void' within predicates text ('#( )'), and indicates which void group it belongs to in the visual text dot assigned to the collection of geometry
add_weight	Adds width (weight) to line or grayscale (weight255) to point or text dot in the visual text dot assigned to geometry
analysis	Analyzes a list of Rhino GUIDs and moves points/extends lines to intersect directly with each other (for lines to intersect with each other, for points to line on lines) in the visual text dot assigned to geometry

NAME	PURPOSE
clear_shape	Clears a list of Rhino objects; takes as input a list/dictionary of Rhino GUIDs OR single Rhino GUID
delete_description	Removes description from Rhino Object's user text and in the visual text dot assigned to geometry; deletes text dot 'attached' to line segment or plane segment, or reverts text dot to a point, if the UserText of the Rhino Object becomes empty after removing the description
delete_label	Removes label from Rhino Object's user text and in the visual text dot assigned to geometry; deletes text dot 'attached' to line segment or plane segment, or reverts text dot to a point, if the UserText of the Rhino Object becomes empty after removing the label
delete_pred_dir	Removes predicate/directive data from Rhino Object and from the visual text dot assigned to the geometry
delete_tag	Removes geometry tag from Rhino Object and from the visual text dot assigned to the geometry
clear_everything	Deletes all Rhino objects in Rhino workspace, and clears all registers in the back-end
tag	Adds or changes the tag data stored in a Rhino geometry and changes the visual text dot 'attached' to the geometry accordingly

add_bound_line

Adds bound line predicate tag to line segment/polyline; indicated as 'bound' within the predicates text ('#( )') in the visual text dot assigned to geometry

Syntax

```
rm.add_bound_line(ends = 2, tagSelf = True)
```

Parameters

- ends: Integer (0, 1 or 2); this indicates whether the line will be bounded on the left side only (0), the right side only (1), or both sides (2)
- tagSelf: Boolean value; True (default) – user may input the desired tag name for the selected geometry; False – method will generate the tag name for the selected geometry based on the currently value inside sc.sticky['keyCount']

Returns

- True: If tagging of geometry/ies with bound line directive is successful; otherwise, a TypeError will be raised

Warnings & Errors

- TypeError:
  - If input for ends is not one of the integers 0, 1 or 2
  - If any of the Rhino geometry in the list of selected objects is not a straight curve (line segment)

Example

```
rm.add_bound_line(ends = 2)
```

add_color

Adds color to Rhino Object (one or several objects may be selected) using RGB values. This color is not visible in Print Preview Mode.

Syntax

```
rm.add_color(r, g, b)
```

Parameters

- r: Integer (value between 0 and 255, inclusive of the two end values) correlating to Red value of (R, G, B) color scale
- g: Integer (value between 0 and 255, inclusive of the two end values) correlating to Green value of (R, G, B) color scale
- b: Integer (value between 0 and 255, inclusive of the two end values) correlating to Blue value of (R, G, B) color scale

Returns

- True: If successful
- False: If unsuccessful

Warnings & Errors

- ValueError: If value of r, g or b is not in between 0 and 255
- TypeError: If any of the input values is not an integer

Example

```
rm.add_color(235, 0, 225)
```

The user is then prompted which objects to change the color of. If multiple objects are selected for one function call, then all these objects will have the same color as one another.

add_description

Adds description data to Rhino Object's user text, visualized as text dot with extra 'd( )' enclosing text in the visual text dot assigned to geometry. The user is first prompted to select the objects to add descriptions to, and then for the description text to add onto each object.

If any items in the description text are to be treated as string literals or as 'labels', then they should be enclosed with double quotes (" ").

Syntax

```
rm.add_description(count = True)
```

Parameters

- count: Boolean value, indicating whether a single object or several objects are to be selected for placing label data onto; True (default) – several objects; False – single object

Returns

- True: If successful; a statement indicating how many objects were given labels is also printed in the Rhino viewport
- False: If no objects were selected

Warnings & Errors

- Warning: If no objects were selected

Example

```
rm.add_description()
```

This allows the user to select multiple objects (as 'count' is set to True by default). Each object will have the function prompting the user for the desired description text.

add_distance

Adds distance directive tag to geometries; first selected geometry may be found in either LHS or RHS, second selected geometry must be found in RHS; range of values for length is inputted to function call; directive is indicated as 'distance' within the directives text ('#( )') in the visual text dot assigned to geometry. Note that if for the value(s) for the range of the distance of the new RHS object, if these inputs exceed 20 characters, then they are not included in the visual text dot text of the geometry.

Syntax

```
rm.add_distance(dist1 = None, dist2 = None, tagSelf = True)
```

Parameters

- dist1: Float, integer or description text string; minimum value of distance of LHS geometry (first object selected) from RHS geometry (second object selected)

- dist2: Float, integer or description text string; maximum value of distance of LHS geometry (first object selected) from RHS geometry (second object selected)
- tagSelf: Boolean value; True (default) – user may input the desired tag name for the selected geometry; False – method will generate the tag name for the selected geometry based on the currently value inside sc.sticky['keyCount']

Returns

- True: If tagging of geometry/ies with bound line directive is successful; otherwise, a TypeError will be raised
- False: If no objects were selected

Warnings & Errors

- TypeError:
  - If no values are inputted for either 'dist1'
  - if either of the objects selected by the function are not straight curves (line segments) or points or text dots
- ValueError: If input values for either 'dist1' or 'dist2' are not float numbers, integers or description text strings

Example

```
rm.add_distance(10, 20)
```

OR

```
rm.add_distance(15)
```

add_embeds

Adds the embeds predicate to two Rhino Objects' user text. The user is prompted to select two geometries; the first one is the container geometry in which the second geometry must be embedded inside. Based on the constraints of the sortal library, the first geometry must be either a plane segment or a line segment. If the former, only points and line segments are allowed in the selection of the second geometry. If the latter, then only points are allowed.

Syntax

`rm.embeds()`

Parameters

- None

Returns

- True: If successful
- False: If no object was selected

Warnings & Errors

- None

Example

`rm.embeds()`

The text displayed in the visual text dots of the selected geometry show the key name / tag name of the Rhino geometry they are paired with, with respect to the embeds predicate.

add_enum

Adds enumerative value to Rhino Object's user text. The user is first prompted for the object to add the enumerative value to, and then is prompted for the single enumerative value. The latter is stored in the UserText of the geometry. This is not visualized in the visual text dot attached to the geometry.

Syntax

`rm.enum()`

Parameters

- None

Returns

- True: If successful
- False: If no object was selected

Warnings & Errors

- None

Example

`rm.enum()`

add_label

Adds label data to Rhino Object's user text, visualized as text dot with extra 'l( )' enclosing text in the visual text dot assigned to geometry. The user is first prompted to select the objects to add labels to, and then for the label text to add onto each object.

Syntax

```
rm.add_label(count = True)
```

Parameters

- count: Boolean value, indicating whether a single object or several objects are to be selected for placing label data onto; True (default) – several objects; False – single object

Returns

- True: If successful; a statement indicating how many objects were given labels is also printed in the Rhino viewport
- False: If no objects were selected

Warnings & Errors

- Warning: If no objects were selected

Example

```
rm.add_label()
```

This allows the user to select multiple objects (as ‘count’ is set to True by default). Each object will have the function prompting the user for the desired label text.

add_longest_line

Adds longest-line predicate tag to line segment/s. This is indicated as 'longest' within predicates text ('#( )') in the visual text dot assigned to geometry.

Syntax

```
rm.add_longest_line(tagSelf = True)
```

Parameters

- tagSelf: Boolean value; True (default) – user may input the desired tag name for the selected geometry; False – method will generate the tag name for the selected geometry based on the currently value inside sc.sticky[‘keyCount’]

Returns

- True: If successful
- False: If no objects were selected

Warnings & Errors

- TypeError: If any of the objects selected by the function are not straight curves (line segments)

Example

```
rm.add_longest_line()
```

The user is then prompted to select which Rhino objects to add the predicate tag to.

add_max_line

Adds max-line predicate tag to line segment/s. This is indicated as 'max' within predicates text ('#( )') in the visual text dot assigned to geometry.

Syntax

```
rm.add_max_line(tagSelf = True)
```

Parameters

- tagSelf: Boolean value; True (default) – user may input the desired tag name for the selected geometry; False – method will generate the tag name for the selected geometry based on the currently value inside sc.sticky['keyCount']

Returns

- True: If successful
- False: If no objects were selected

Warnings & Errors

- TypeError: If any of the objects selected by the function are not straight curves (line segments)

Example

```
rm.add_max_line()
```

The user is then prompted to select which Rhino objects to add the predicate tag to.

add_no_label

Adds no-label predicate tag to geometry to constrain matching to geometries without label attributes. This is indicated as 'no_label' within predicates text ('#( )') in the visual text dot assigned to geometry.

Syntax

```
rm.add_no_label(tagSelf = True)
```

Parameters

- tagSelf: Boolean value; True (default) – user may input the desired tag name for the selected geometry; False – method will generate the tag name for the selected geometry based on the currently value inside sc.sticky['keyCount']

Returns

- True: If successful
- False: If no objects were selected

Warnings & Errors

- None

Example

```
rm.add_no_label()
```

The user is then prompted to select which Rhino objects to add the no-label tag to.

add_normal

Adds normal directive tag to line segments (if 2D) or plane segments (if 3D). This is indicated as 'normal ^ <group_name>' within the directives text ('#( )') in the visual text dot assigned to geometry. Note that if for the value(s) for the range of the length of the new normal line, if these inputs exceed 20 characters, then they are not included in the visual text dot text of the geometry.

Syntax

```
rm.add_normal(rhsTag, length, targetCoords = None, tagSelf = True)
```

Parameters

- rhsTag: Text string that will be the tag name of the new normal line
- length: A single integer, float, description string or a list/tuple of two integers, floats, description strings (the two elements need not necessarily be the same data type); if a list/tuple is given as input, then the first element is considered the lower bound of the range for the length of the normal line segment to be generated, and the second element the upper bound of the same range
- targetCoords: List/tuple of two numbers (<x, y> coordinates) or Rhino GUID (point or text dot) that will represent the directional vector that the normal line segment will be perpendicular to; this directional vector must exist in the chosen geometry, be it a line segment or a plane segment
- tagSelf: Boolean value; True (default) – user may input the desired tag name for the selected geometry; False – method will generate the tag name for the selected geometry based on the currently value inside sc.sticky['keyCount']

Returns

- True: If successful
- False: If no objects were selected

Warnings & Errors

- TypeError:
  - If the input for 'rhsTag' is not a text string
  - If the input for length is neither a single integer, float, description string nor a list/tuple of two integers, floats, description strings (the two elements need not necessarily be the same data type)
- Value Error:
  - If there are more than two elements in the input for length of the new normal line segment
  - If any of the objects selected by the function are not straight curves (line segments)

Example

The following function call and inputs adds a directive to generate a new normal line segment with length 10 based on a pre-existing line segment with the directional vector <20, 15> present inside it. The new normal line segment will have the tag name 'lineSegment-2'.

```
rm.add_normal('2', 10, (20,15))
```

Alternatively, the function call and inputs adds a directive to generate a new normal line segment with its length between 10 and 15 based on a pre-existing plane segment. The new normal line segment will have the tag name 'lineSegment-2'.

```
rm.add_normal('2', [10, 15])
```

add_point_on_line

Adds point-on-line directive tag to line segment(s)/polyline(s). The upper and lower bounds for the range of segment values for point placement is inputted to function call. Afterwards, the user is prompted to select the line segments which will have this directive added to their UserText. This is visually indicated as 'point_on_line ^ <group_name>' within the directives text ('#( )').

Syntax

```
rm.add_point_on_line(seg1 = None, seg2 = None, tagSelf = True)
```

Parameters

- seg1: A float value (0 < x < 1) that serves as the lower bound of the segment range where the new point will lie on the line segment
- seg2: A float value (0 < x < 1) that serves as the upper bound of the segment range where the new point will lie on the line segment
- tagSelf: Boolean value; True (default) – user may input the desired tag name for the selected geometry; False – method will generate the tag name for the selected geometry based on the currently value inside sc.sticky['keyCount']

Returns

- True: If successful
- False: If no objects were selected

Warnings & Errors

- TypeError: If any of the objects selected by the function are not straight curves (line segments)
- ValueError: If either 'seg1' (lower bound) or 'seg2' (upper bound) are not float data types within the range (0 < x < 1)

Example

```
rm.add_point_on_line(0.1)
```

add_shortest_line

Adds shortest-line predicate tag to line segment/s. This is indicated as 'shortest within predicates text ('#( )') in the visual text dot assigned to geometry.

<u>Syntax</u>

```
rm.add_shortest_line(tagSelf = True)
```

<u>Parameters</u>

- tagSelf: Boolean value; True (default) – user may input the desired tag name for the selected geometry; False – method will generate the tag name for the selected geometry based on the currently value inside sc.sticky['keyCount']

<u>Returns</u>

- True: If successful
- False: If no objects were selected

<u>Warnings & Errors</u>

- TypeError: If any of the objects selected by the function are not straight curves (line segments)

<u>Example</u>

```
rm.add_shortest_line()
```

The user is then prompted to select which Rhino objects to add the predicate tag to.

add_void

Adds void predicate tag to collection of points/text dots/line segments/plane segments that form a closed polygon. The user is prompted to select the geometry that will compose the void area. This may be composed of points, line segments or a plane segment (mesh, BREP, surface). This is indicated as 'void' within predicates text ('#( )'), and indicates which void group it belongs in the visual text dot assigned to the geometry.

<u>Syntax</u>

```
rm.add_void(sortType = [], detectIntersection = False)
```

<u>Parameters</u>

- sortType: List of text strings, where each element is the name of a sort type that should not be present inside the area delineated by the void coordinates
- detectIntersection: Boolean value; True – checks if geometry given as input for void coordinates is self-intersecting and does not proceed with adding void predicate tag to geometry if yes; False – does not check if input geometry is self-intersecting

<u>Returns</u>

- True: If successful
- False: If no objects were selected

Warnings & Errors

- ValueError:
  - If a sort type listed in the input 'sortType'
- TypeError:
  - If the input for 'sortType' is not a text string or a list of text strings
- Warning:
  - If no geometry was selected
  - If the selected geometry were all points and do not form a polygon (at least a triangle) or there are not enough points to form a polygon (at least three non-collinear points are required)
  - If the selected geometry were all line segments and there are not enough line segments to form a polygon (at least three non-collinear line segments are required)
  - If detectIntersection is True, then if any of the lines intersect one another

Example

```
rm.add_void(['lineSegment-A3D', 'point-V3D'])
```

This means that the area of the polygon delineated by the void coordinates as vertices cannot have any individuals with the sort type 'lineSegment-A3D' or 'point-V3D' inside.

add_weight

Adds width (weight) to line or grayscale (weight255) to point or text dot in the visual text dot assigned to geometry. If several objects are selected, then the function goes through each object in the list of Rhino GUIDs and highlights them with yellow, to help the user recognize which Rhino object is currently being altered. The user is prompted for the target weight (either line width, if the Rhino object is a line segment, or gray scale, if the Rhino object is a point or a text dot)

Syntax

```
rm.add_weight(several = True)
```

Parameters

- several: Boolean value: True (default) – the user may select several objects to add a weight attribute to; False – the user may select only one object to add a weight attribute to

Returns

- True: If successful
- False: If no objects were selected

Warnings & Errors

- TypeError:
  - If input for intended line width is not within the range 0.0 (mm) to 2.0 (mm) or is not a float data type
  - If input for intended gray scale value (for points and text dots) is not within the range 0 to 255 or is not a positive integer

- Warning:
    - If no geometry was selected

Example

```
rm.add_weight()
```

The user will then be prompted to select the objects whose line width or color will be altered. If several objects are selected, the function goes through them one by one, highlighting the object in question in yellow and prompting the user for the desired line width or gray scale value. Once the object has been altered, it is reverted to its original color, if it is a line segment.

analysis

Analyzes a list of Rhino GUIDs and moves points/extends lines to intersect directly with each other (for lines to intersect with each other, for points to line on lines) in the visual text dot assigned to geometry. The precision value is x < 1% of the length of the line or Rhino's unit setting. The GUIDs in the list are modified to fit more accurately with each other (points lying on lines, line segment intersection with other line segments).

Syntax

```
rm.analysis(objects)
```

Parameters

- objects: List of Rhino GUIDs to be analyzed

Returns

- objects: List of Rhino GUIDs after modification, if successful
- False: If number of Rhino GUIDs is less than two (2)

Warnings & Errors

- Warning: if number of Rhino GUIDs in input list is less than two (2)

Example

```
shape = rs.GetObjects()
shape_modified = rm.analysis(shape)
```

clear_shape

Clears a list of Rhino objects. This method takes as input a list/dictionary/tuple of Rhino GUIDs or a single Rhino GUID and deletes it from the Rhino workspace.

Syntax

```
rm.clear_shape(guids)
```

Parameters

- guids: List of Rhino GUIDs to be deleted from workspace

Returns

- True: If Rhino GUID/s is/are successfully deleted from viewport
- False: If no objects were selected

Warnings & Errors

- TypeError: If input is not a list/tuple/dictionary or a single Rhino GUID
- Warning: If no objects were selected

Example

```
shape = rs.GetObjects('Select shapes to clear from Rhino viewport')
rm.clear_shape(shape)
```

delete_description

Removes description data from the UserText of the Rhino Object and from the visual text dot assigned to geometry, if any are present. The user is first prompted for the object/s to remove descriptions from. Afterwards, the function goes through every object and displays what descriptions are currently present in the UserText of the object. The user is then prompted for an integer input corresponding to the description data they wish to remove. If the Rhino Object is a text dot and no text is left in the UserText after the removal of the label, the text dot is changed to a point.

Syntax

```
rm.delete_description()
```

Parameters

- None

Returns

- True: if successful
- False: If no objects were selected

Warnings & Errors

- ValueError: If the input number given by the user does not correspond to any description displayed by the function
- TypeError:
  - If any object in the inputted list is not a point, text dot, curve, BREP, surface or mesh
  - If the input for which predicate/directive data to remove is not an integer
- Warning: If no objects were selected

Example

```
rm.delete_description()
```

The function then displays which descriptions are currently present in the Rhino object and prompts the user to enter an integer value corresponding to the description they wish to remove from the UserText of the Rhino GUID.

delete_label

Removes label data from the UserText of the Rhino Object and from the visual text dot assigned to geometry, if any are present. The user is first prompted for the object/s to remove labels from. Afterwards, the function goes through every object and displays what labels are currently present in the UserText of the object. The user is then prompted for an integer input corresponding to the label data they wish to remove. If the Rhino Object is a text dot and no text is left in the UserText after the removal of the label, the text dot is changed to a point.

Syntax

`rm.delete_label()`

Parameters

- None

Returns

- True: if successful
- False: If no objects were selected

Warnings & Errors

- ValueError: If the input number given by the user does not correspond to any label displayed by the function
- TypeError:
    - If any object in the inputted list is not a point, text dot, curve, BREP, surface or mesh
    - If the input for which predicate/directive data to remove is not an integer
- Warning: If no objects were selected

Example

`rm.delete_label()`

The function then displays which labels are currently present in the Rhino object and prompts the user to enter an integer value corresponding to the label they wish to remove from the UserText of the Rhino GUID.

delete_pred_dir

Removes predicate/directive data from the UserText of the Rhino Object and from the visual text dot assigned to geometry, if any are present. The user is first prompted for the object/s to remove predicates/directives from. Afterwards, the function goes through every object and displays what predicates/directives are currently present in the UserText of the object. The user is then prompted for an integer input corresponding to the predicate/directive data they wish to remove. If the Rhino Object is a text dot and no text is left in the UserText after the removal of the label, the text dot is changed to a point.

Syntax

`rm.delete_pred_dir()`

Parameters

- None

Returns

- True: if successful
- False: If no objects were selected

Warnings & Errors

- ValueError: If the input number given by the user does not correspond to any label displayed by the function
- TypeError:
  - If any object in the inputted list is not a point, text dot, curve, BREP, surface or mesh
  - If the input for which predicate/directive data to remove is not an integer
- Warning: If no objects were selected

Example

```
rm.delete_pred_dir()
```

The function then displays which predicates/directives are currently present in the Rhino object and prompts the user to enter an integer value corresponding to the label they wish to remove from the UserText of the Rhino GUID.

delete_tag

Removes geometry tag (usually encased inside 'line( )', 'point( )', or 'plane( )') from Rhino Object and from the visual text dot assigned to the object, if any are present. The user is prompted for the object/s to remove tags from, and the tag is removed from both the visual text dot of the Rhino Object as well as its UserText data. If the Rhino Object is a text dot and no text is left in the UserText after the removal of the label, the text dot is changed to a point.

Syntax

```
rm.delete_tag()
```

Parameters

- None

Returns

- True: if successful
- False: If no objects were selected

Warnings & Errors

- Warning:
  - If no objects were selected
  - If the object does not have any 'tag' key in its UserText dictionary

Example

```
rm.delete_tag()
```

The user is then prompted to select objects to remove tags from.

clear_everything

Deletes all Rhino objects in Rhino workspace, and clears all registers (sort, rule, flow, form, sdlParser.forms, sdlParser.flows, sdlParser.rules) in the back-end.

Syntax

```
rm.clear_everything()
```

Parameters

- None

Returns

- True: if successful

Warnings & Errors

- None

Example

```
rm.clear_everything()
```

tag

Adds a tag to a Rhino object or changes the tag stored in a Rhino object and changes the visual text dot assigned to the Rhino object accordingly. Data within the Rhino object's user text pertaining to the directive 'normal', and that is reliant on the tag of the Rhino object, are also changed.

Syntax

```
sgi.tag()
```

Parameters

- tagSelf: Boolean value; True (default) – user may input the desired tag name for the selected geometry; False – method will generate the tag name for the selected geometry based on the currently value inside sc.sticky['keyCount']

Returns

- True: if successful

Warnings & Errors

- None

Example

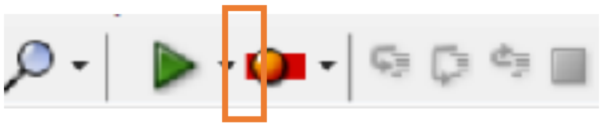
```
sgi.tag()
```

The user is then prompted to select objects whose tags they wish to change or remove. If no input is given when the user is prompted to enter the new tag of a Rhino object, then the tag data item is deleted completely from the Rhino object.

FAQ

1. *I get empty sortal shapes whenever I use functions like 'create_shape' or 'create_shape_ag'. What do I do?*

There are two options you may do in this case. The first one is to rerun your code by using the 'Reset and Debug' option in the Rhino Python compiler, pictured below in the orange box:



The second option is to shut down Rhinoceros completely, re-open the Rhino and Python files, and rerun your code.

2. *The edges of some of my line segments do not meet each other or I have generated shapes where the points do not exactly lie on the lines. How can I fix this?*

Due to the nature of Rhinoceros, the dimensions of the last few digits of its measurements may fluctuate and thus affect computations in the Sortal Library, since the back-end precision is fixed. You may change the precision of comparison (the number of decimal places the back-end will consider when performing operations) using 'set_precision'. Alternatively, you may set the dimension of your Rhinoceros application to a larger base dimension (e.g. cm instead of mm).

3. *Can I get some help?*

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

Annex A: About Sortal Structures

This annex explains about *sortal* structures, or *sorts*, the various types of *sorts*, the various behavioral categories and provides an overview of all data types or characteristic individuals that exist to define *sorts*.

Sortal structures and behavioral categories

Sortal* structures**, also denoted as ***sorts, are representational structures defined as formal compositions of other, primitive, *sortal* structures. While the terms *sortal* structures and *sorts* may be used interchangeably, the term *sortal* structure emphasizes the formal compositional character of the representational structure, while the term *sort* refers to the universe of entities (called *individuals*) as represented by the structure.

- *Sortal* structures are class structures, specifying either a single data type or a composition of other class structures. For instance, data types such as points, labels, and lines all define *sorts*.
- A *sortal* structure can also be considered as a hierarchical structure of properties, where each property specifies a data type (a *primitive sort*). Properties can be collected (a *disjunctive sort*) and a collection of one or more properties can be assigned as an attribute to another property (an *attribute sort*).
- A ***sortal* description** is a description of a data construct, corresponding to a *sortal* structure, and expressed as an individual or *form* (or *metaform*) of a *sort*.
- An **Individual** is the basic element of a *sort*, that is, an instance of the class structure. For example, a point is an individual of the *sort* of points. Every *sort* also allows for a **nil** value or individual. When taking the complement of an individual with respect to another individual, or determining the common part of two individuals, the result may be *empty* or nil.
- A **form** is a collection of individuals of the same *sort*, e.g., a set of points.
- A **metaform** is a collection of forms corresponding to the different component *sorts* of a *disjunctive sort*, e.g., a set of points and line segments.

We distinguish four types of *sorts*: *primitive sorts*, *attribute sorts*, *disjunctive sorts*, and *compound sorts*.

- A **primitive *sort*** specifies a single data type. An individual of a primitive *sort* has a data value of the specified type.
- An **attribute *sort*** is a subordinate, semi-conjunctive composition of a primitive *sort* (its base) with any other *sort* (its weight) under the object-attribute relationship. An individual of an attribute *sort* is an individual of the base *sort* (the *associate individual*) that is assigned a *form* (a collection of zero, one or more individuals) of the weight *sort* as an attribute (the *attribute form*). If the attribute form is empty, it may be omitted and the individual is treated as an individual of the base *sort* only, rather than of the attribute *sort*.
- A **disjunctive *sort*** is a co-ordinate, disjunctive composition of any number of *sorts*. A form of a disjunctive *sort* is a composition of forms from the respective component *sorts*, and is called a *metaform*. The representation of each component *sort* in the composition of forms is optional.
- A **compound *sort*** is a co-ordinate, disjunctive composition of (disjunctive) *sorts*. The distinction between disjunctive and compound *sorts* recognizes the fact that a single *sortal* structure may be adopted to represent a collection of 'drawings'. Where the compound *sort* represents the collection of drawings, each (disjunctive) component *sort* represents a single drawing. In the case of shape matching for rule application, shape elements from the same component *sort* match under the same transformation, while shapes from different component *sorts* match under separate transformations.

Each *sort* may be specified a name, for the purpose of semantic disambiguation. This is a requirement for every primitive *sort* (or aspect).

Each primitive *sort* is defined by its *characteristic individual* and its *behavioral category*. In addition, the definition of a primitive *sort* may include one or more arguments, constraining the possible data entities this *sort* may represent. For example, the definition of a *sort* of weights may include the specification of an upper bound for the numeric weight values as argument.

- The **characteristic individual** defines the representational aspect of a primitive *sort*, specifically, the representation of its individuals' data values and behavioral methods. It is specified in its class implementation. Examples of characteristic individuals are points, line segments and labels.
- The **behavioral category** of a primitive *sort* specifies the operational behavior of its forms and is assigned in a categorization of the characteristic individuals. Specifically, the behavioral category prescribes the behavior of forms under common arithmetic operations (sum, difference and product/ intersection), their canonical (maximal) form, and when a form is part of another form.

The behavior (of forms) of a composite *sort* (whether an attribute or a disjunctive *sort*) derives from the behavior of its component *sorts* depending on the compositional relationship.

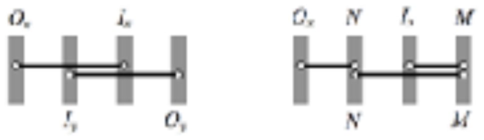
We distinguish six behaviors for primitive *sorts*: *discrete*, *ordinal*, *interval*, *cyclical*, *areal* and *custom*, and their respective forms.

- A **discrete** form is a form with a discrete operational behavior, corresponding to a mathematical set: an individual is part of another individual, only if these are identical; a form is part of another form, if every individual of the first form is also an individual of the second form. The operations of sum, difference and product on forms correspond to set union, difference and intersection, respectively: under the operation of sum, forms are merged and duplicate individuals are removed; under the operation of product, only identical individuals contribute to the result. A discrete form is maximal if no two individuals are identical.
  In other words, if x and y denote two forms of a *sort* with discrete behavior, and X and Y denote the respective sets of individuals, then ($x : X$ specifies X as a representation of x)
  $$x : X \wedge y : Y \;\Rightarrow\; \begin{array}{l} x \leq y \Leftrightarrow X \subseteq Y \\ x + y \;:\; X \cup Y \\ x - y \;:\; X \,/\, Y \\ x \cdot y \;:\; X \cap Y \end{array}$$
  In the case of an attribute *sort*, an individual is part of another individual, only if these are identical and the former's attribute form is part of the latter's attribute form. Under the operation of sum, identical individuals have their attribute forms combined under the (corresponding) operation of sum. Under the operation of product, only identical individuals contribute to the result; their attribute forms combine under the (corresponding) operation of product. The resulting attribute form may be empty.
- An **ordinal** form is a form with an ordinal operational behavior: an individual is part of another individual, only if its ordinal value is smaller than or equal to the latter's ordinal value. Since, for any two ordinal values, one is always less than or equal to the other, an ordinal form is maximal if it contains only a single individual. Thus, a form is part of another form, if the former's individual is part of the latter's individual. Under the operations of sum and product, the resulting form's individual has as ordinal value the largest and, respectively, smallest of both ordinal values.
  Two variant ordinal forms are distinguished from the operation of difference. In both variants, the difference of a smaller ordinal value with respect to a larger or equal ordinal value is nil. In variant 1, the difference of a larger ordinal value with respect to a smaller ordinal value is the larger ordinal value, whereas in variant 2, it is the numeric difference of the two ordinal values.
  $$x : \{m\} \wedge y : \{n\} \Rightarrow \quad x \leq y \Leftrightarrow m \leq n$$

$x + y$: $\{\max(m, n)\}$
$x - y$: $\{\}$ if $m \le n$, else $\{m\}$ [1]
$\{\}$ if $m \le n$, else $\{m - n\}$ [2]
$x \cdot y$: $\{\min(m, n)\}$

In the case of an attribute *sort*, an individual is part of another individual, if its ordinal value is smaller than or equal to the latter's ordinal value and its attribute form is part of the latter's attribute forms. Under the operations of sum and product, the resulting form's individual has as ordinal value the largest and, respectively, smallest of both ordinal values, while its attribute form is the result of the respective operation on both attribute forms.

- An **interval** form is a form with a one-dimensional embedding operational behavior: an interval is part of another interval if it is embedded in the other interval; a form is part of another form if every interval in the first form is embedded in an interval in the second form. Under the operation of sum, forms are merged and adjacent (on the same carrier) or overlapping intervals are combined into a single interval; under the operation of product, the result is composed of the common parts of overlapping intervals. An interval form is maximal if no two intervals are adjacent (on the same carrier) or overlap.
Let $B[x]$ denote the set of boundary elements of a form x of intervals and, given two interval forms x and y of the same *sort*, let I_x denote the set of boundary elements of x that lie within y, O_x the set of boundary elements of x that lie outside of y, M the set of boundary elements of both x and y where the respective intervals lie on the same side of the boundary element, and N the set of boundary elements of both x and y where the respective intervals lie on opposite sides of the boundary element; then
$x : B[x] \wedge y : B[y] \quad \Rightarrow \quad x \le y \Leftrightarrow I_x = \emptyset \wedge O_y = \emptyset \wedge N = \emptyset$
$x + y$: $B[x + y] = O_x \cup O_y \cup M$
$x - y$: $B[x - y] = O_x \cup I_y \cup N$
$x \cdot y$: $B[x \cdot y] = I_x \cup I_y \cup M$



In the case of an attribute *sort*, an interval is part of another interval if it is embedded in the other interval and the former's attribute form is part of the latter's attribute form. Under the operation of sum, overlapping intervals are split at the mutual boundary points and identical parts are combined into one, with the attribute forms combined under the (corresponding) operation of sum. Adjacent intervals (on the same carrier) that have identical attribute forms are also combined. Under the operation of product, the result is composed of the common parts of overlapping intervals, with the attribute forms combined under the (corresponding) operation of product. The resulting attribute form may be empty. An interval form is maximal if no two intervals overlap and if adjacent intervals (on the same carrier) have non-identical attribute forms.

- A **cyclical** form is a form with a one-dimensional cyclical embedding operational behavior. This behavior is quasi-identical to the one-dimensional embedding operational behavior for interval forms as described above, except that for an interval form, all intervals can be ordered based on the starting point of the interval in order to simplify the process of identifying adjacent and overlapping intervals. In the case of the cyclical embedding operational behavior, such ordering must necessarily take into account that the first and last intervals may also be adjacent or overlap.

- An **areal** form is a form with a two-or-higher-dimensional embedding operational behavior. An areal form behaves similar to an interval form: an areal is part of another areal if it is embedded in the other areal. Under the operation of sum, forms are merged and areals that overlap or share boundary (on the same carrier) are combined into a single areal. An areal form is maximal if no two areals overlap or share boundary (on the same carrier).
Let $B[x]$ denote the form of boundary segments of a form x of areals (e.g., if x is a form of plane

segments, $B[x]$ will be a form of line segments) and, given two areal forms x and y of the same *sort*, let I_x denote the form of boundary segments of x that lie within y, O_x denote the form of boundary segments of x that lie outside of y, M the form of boundary segments of both x and y where the respective areals lie on the same side of the boundary segment, and N the *form* of boundary segments of both x and y where the respective areals lie on opposite sides of the boundary segment; then

$$\begin{aligned} x : B[x] \wedge y : B[y] \quad \Rightarrow \quad & x \leq y \Leftrightarrow I_x = 0 \wedge O_y = 0 \wedge N = 0 \\ & x + y \;:\; B[x + y] = O_x + O_y + M \\ & x - y \;:\; B[x - y] = O_x + I_y + N \\ & x \cdot y \;:\; B[x \cdot y] = I_x + I_y + M \end{aligned}$$

In the case of an attribute *sort*, an areal is part of another areal if it is embedded in the other areal and the former's attribute form is part of the latter's attribute form. Under the operation of sum, overlapping areals are split at their mutual boundaries and identical parts are combined into one, with the attribute forms combined under the (corresponding) operation of sum. Areals that share boundary (on the same carrier) and have identical attribute forms are also combined. An areal form is maximal if no two areals overlap and if areals that share boundary (on the same carrier) have non-identical attribute forms.

- A **color** form is a form with a custom, ordinal-like behavior. The specification of a color sort requires the specific behavior to be specified, i.e., whether the sum of two color values is the average RGB value, the maximum RGB value, the sum of the RGB values, or defined as a function of the respective alpha values.
- An **enumerative** form is a form with a custom, ordinal-like behavior. An enumerative value is a value from among an enumerated set. The specification of an enumerative *sort* requires the enumeration of the values as well as their mutual ranking. The enumeration values are specified as a set of identifiers, and their ranking as an array of enumeration values resulting from the addition of every combination of two values (ordered as a matrix, corresponding the original enumeration ordering). For instance, given an enumeration of black and white (in that order), a ranking array of black, black, black and white would mean black dominates white as any addition of two values, except for white and white, results in black. The product of two enumerative values is always nil, unless the two enumerative values are identical. An enumerative *sort* supports the specification of qualitative aspects in "color grammars" (Knight 1989; 1993).

Data types and characteristic individuals

The table below specifies all characteristic individuals available in the Python *sortal* library. Geometric types can be specified within a two-dimensional or three-dimensional space; their characteristic individuals are also distinguished in the context of parametric and non-parametric rules (e.g., point3D versus pointP3D). Do note that not all characteristic individuals are available through the SortalGI API.

SortalGI	Data type	Space	Characteristic individual	Behavior/form
✓	points	2D	point2D	discrete
			pointP2D	
		3D	point3D	
			pointP3D	
	lines	2D	line2D	discrete

	– unbounded	3D	line3D	
✓	line segments – bounded and half-bounded	2D	lineSegment2D	interval
			lineSegmentP2D	
		3D	lineSegment3D	
			lineSegmentP3D	
	planes – unbounded	2D	plane2D	discrete
		3D	plane3D	
✓	plane segments – bounded, rectilinear	2D	planeSegment2D	areal
			planeSegmentP2D	
		3D	planeSegment3D	
			planesegmentP3D	
✓	circles – closed, planar	2D	circle2D	discrete
			circleP2D	
		3D	circle3D	
			circleP3D	
✓	circular arcs – planar	2D	circularArc2D	cyclical
		3D	circularArc3D	
✓	ellipses – closed, planar	2D	ellipse2D	discrete
			ellipseP2D	
		3D	ellipse3D	
			ellipseP3D	
✓	elliptical arcs – planar	2D	ellipticalArc2D	cyclical
		3D	ellipticalArc3D	
✓	Bezier curves – quadratic	2D	bezier2D	interval
			bezierP2D	
		3D	bezier3D	
			bezierP3D	
✓	labels – alphanumeric		label	discrete
✓	numeric values		numeric	discrete
✓	weights – non-negative, numeric		weight	ordinal[1]
			rWeight	ordinal[2]
✓	enumerated values		enumerative	custom
✓	color values – RGB or HSV		color	custom
✓	shape descriptions		description	discrete

Annex B: About Shape Rules and Description Rules

This annex explains about shape rules and description rules.

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.

Shape rules

Two types of rules are distinguished, parametric 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 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 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 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 other spatial data types, e.g., plane segments.

Descriptions and description rules

Descriptions follow a strict format that allows them to be interpreted and matched by the *sortal* library.

Parametric descriptions

Descriptions are parametric in nature, that is, when adopted as the left-hand-side (*lhs*) of a description rule, a 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 description rule, a 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, descriptions that do not form part of a description rule should not contain any parameters or parameter references, otherwise matching will necessarily fail.

Example ('description' is the description *sort* name and 'a' is a parameter):

```
description: a
```

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 'π', 1 and 0, respectively.

Examples:

```
status: true
list: e
```

Description tuples

While 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 further.

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(point3D.value:labelD.value="1", point3D.value:labelD.value
="2")
```

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 rules (pRule). Note that circular arcs are not yet available within parametric 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	vector tuple*	direction vector
		unitDir	vector tuple*	unit direction vector
		start	vector tuple*	‘smallest’ endpoint
		end	vector tuple*	‘greatest’ endpoint
		midpoint	vector tuple*	midpoint
		length	number	line length
		squareLength	number	square value of line length
plane segments	planeSeg3D	normal	vector tuple*	normal vector
	planesegP3D	area	number	plane area
		outer	tuple of vector tuples*	list of outer boundary vertices
circles	circle3D	normal	vector tuple*	plane normal vector
	circleP3D	center	vector tuple*	center point
		radius	number	radius
		diameter	number	diameter
		circumference	number	circumference
		area	number	area of the circle
ellipses	ellipse3D	normal	vector tuple*	plane normal vector
	ellipseP3D	center	vector tuple*	center point
		foci	tuple of vector tuples*	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	vector tuple*	circle center point
		radius	number	circle radius
		diameter	number	circle diameter
		circumference	number	circle circumference
		start	vector tuple*	endpoint (ccw)
		end	vector tuple*	endpoint (cw)
		length	number	arc length
		angle	number	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	vector tuple*	1st control point
		controlPoint	vector tuple*	2nd control point
		end	vector tuple*	3rd control point
		vertex	vector tuple*	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.

A formal notation for descriptions

The table below presents a formal notation for descriptions and the left-hand-side (*lhs*) and right-hand-side (*rhs*) of 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 | 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”

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

Annex C: Description of Sortal Imports

This annex describes the functions of *sortal* imports. These are mostly seen in the source code files of SortalGI API and Rhino methods.

IMPORT	DESCRIPTION
attributeSort	An attribute *sort* specifies a subordinate, semi-conjunctive composition of a primitive *sort* (its base) with any other *sort* (its weight/label) under the object-attribute relationship. An individual of an attribute *sort* is an individual of the base *sort* (the *associate individual*) that is assigned a *form* (a collection of zero, one or more individuals) of the weight/label *sort* as an attribute (the *attribute form*). If the attribute form is empty, it may be omitted, and the individual is treated as an individual of the base *sort* only, rather than of the attribute *sort*. An attribute *sort* may have a name assigned. The attributeSort class represents an attribute sort additionally by its base and weight *sorts*. The canonical version of an attribute *sort* is the unnamed attribute *sort* of the canonical versions of the base and weight *sort*.
bezierCurve2D, bezierCurve3D, bezierCurveP2D, bezierCurveP3D	These classes serve as base objects for quadratic Bezier curves, that contain their properties, behaviors and methods. P2D and P3D indicate that parametric behavior is enabled in the class.
circle2D, circle3D, circleP2D, circleP3D	These classes serve as base objects for circles, that contain their properties, behaviors and methods. P2D and P3D indicate that parametric behavior is enabled in the class.
circularArc2D, circularArc3D	These classes serve as base objects for circular arcs, that contain their properties, behaviors and methods. P2D and P3D indicate that parametric behavior is enabled in the class.
color	This class serves as base object for color values specified in RGB or HSV space, that contain their properties, behaviors and methods. An additional argument may specify the kind of ordinal behavior, i.e., whether the sum of two color values is the average RGB value, the maximum RGB value, the sum of the RGB values, or defined as a function of the respective alpha values.
compoundSort	The compoundSort class represents a disjunctive *sort* as an ordered set of component *sorts* (disjunctive *sorts* only). In its canonical version, component *sorts* cannot themselves be compound *sorts*.
coordinate	A class structure to contain and handle numerical data such as the <x.y.z> values of vectors. It is specifically built to allow for easy conversion between numerical data types such as float **<->** integer.

IMPORT	DESCRIPTION
description	Compound data entity with discrete behavior. A description may contain tuples, strings, numbers and certain literals. When part of a description rule, a description may be parametric and contain expressions. A description sort supports the specification of description functions/grammars (Stiny 1981; Stouffs 2016a; 2016b)[2].
disjunctiveSort	A disjunctive *sort* specifies a co-ordinate, disjunctive composition (under the operation of sum) of any number of primitive and attribute *sorts.* A form of a disjunctive *sort* is a composition of forms from the respective component *sorts* and is called a *metaform*. The representation of each component *sort* in the composition of forms is optional. The disjunctiveSort class represents a disjunctive *sort* as an ordered set of component *sorts*. In its canonical version, component sorts cannot themselves be disjunctive sorts. If any component sort is an unnamed disjunctive sort, instead, its components become part of the disjunctive composition. If any component sort is part of another, unnamed component sort, the former component sort is not included in the composition. A disjunctive sort may have a name assigned.
ellipse2D, ellipse3D, ellipseP2D, ellipseP3D	These classes serve as base objects for ellipses, that contain their properties, behaviors and methods. P2D and P3D indicate that parametric behavior is enabled in the class.
ellipticalArc2D, ellipticalArc3D	These classes serve as base objects for elliptical arcs, that contain their properties, behaviors and methods.
enumerative	This class serves as base object for enumerative values, that contain their properties, behaviors and methods.
flow	A sequence of rule objects along with instructions for the ordering of rule objects, the number of times a rule should be looped, e.g.: 'r1 r2(3) r5+ {r1 r2 r4*}'
form	A form is a collection of one or more individuals from the same sort, e.g., a collection of points defines a form of the sort of points. Forms can be collected into metaforms or assigned as an attribute form to another individual. – A metaform is a collection of forms in accordance to a disjunctive sort. – An attribute form is a form, or metaform, that is assigned as an attribute to another individual, in accordance to an attribute sort. A form is completely specified by its *sort* and its collection of individuals (or in the case of a metaform, its collection of forms).

IMPORT	DESCRIPTION
label	A label is an alphanumerical data entity with a discrete behavior, i.e., the value of the sum of two labels is the collection of both labels, unless both labels are identical, in which case is either label. The Label class extends on the Individual class. It defines the characteristic individual for labels. A label is represented as a string. This characteristic individual accepts no parameters. It specifies an {sortal.map. ExactMap} as default. Forms of labels adhere to a discrete behavior.
line2D, line3D	A line is a linear, connected, non-bounded planar curve with a discrete behavior. The Line class extends on the Individual class. It defines the characteristic individual for lines. A line is represented as a direction vector and a position vector specifying the root of the line. This characteristic individual accepts no parameters. It specifies a {sortal.map. similarityMap} as default. Forms of lines adhere to a discrete behavior.
lineSegment2D, lineSegment3D, lineSegmentP2D, lineSegmentP3D	lineSegment2D: A bounded (and half-bounded) line segment in two-dimensional space, with interval behavior. lineSegment3D: A bounded (and half-bounded) line segment in three-dimensional space, with interval behavior. A line-segment is a connected and bounded segment of a line with an interval behavior. The line defines the co-descriptor of the line-segment, the boundary of the segment is defined by the start and end positions of the line-segment. Vectors or points may be used to define the start and end positions. The lineSegment class extends on the Line class and implements the Interval interface. It defines the characteristic individual for line-segments. P2D and P3D indicate that parametric behavior is enabled in the class. A line-segment is represented as a line with two rational scalars specifying the tail and head relative to the line's root. This characteristic individual accepts no parameters. It specifies a {sortal.map. similarityMap} as default. Forms of line-segments adhere to an interval behavior.
planeSegment2D, planeSegment3D, planeSegmentP2D, planeSegmentP3D	planeSegment2D: A bounded, rectilinear plane segment in two-dimensional space, with areal behavior. planeSegment3D: A bounded, rectilinear plane segment in three-dimensional space, with areal behavior. These refer to classes that serve as base objects for plane segments, that contain their properties, behaviors and methods. P2D and P3D indicate that parametric behavior is enabled in the class.

IMPORT	DESCRIPTION
point2D, point3D, pointP2D, pointP3D	point2D: A point in three-dimensional space, with discrete behavior. point3D: A point in two-dimensional space, with discrete behavior. A point is a 0-dimensional geometric data entity with a discrete behavior. P2D and P3D indicate that parametric behavior is enabled in the class. The point class extends on the Individual class. It defines the characteristic individual for points. A point is represented as a position vector. This characteristic individual accepts no parameters. It specifies a {sortal.map. similarityMap} as default. Forms of points adhere to a discrete behavior.
primitiveSort	Specifies a single data type. An individual of a primitive *sort* has a data value of the specified type.
rule	Rule class that takes as input the rule description, LHS and RHS sides of rule.
similarityMap	Mapping function used to distinguish that two individuals of the same sort type (e.g. two line segments ls1 and ls2, or two points p1 and p2) are of the same sort type despite being in different locations or having different coordinates, and that they can be mapped together.
sdlParser	Class of methods that handles reading and writing SDL files. This class also stores forms, rules, flows and sort types under the following dictionaries: sdlParser.forms, sdlParser.rules, sdlParser.flows, sdlParser.sorts
sort	Sorts can be considered as class structures, specifying either a single data type or a composition of other class structures. For instance, data types such as points, labels, and lines all define sorts.
vector2D, vector3D	A vector specifies a position in a two-dimensional Cartesian space. If normalized, it only specifies a direction. The Vector class defines a vector as a pair of {coordinate}'s and a w factor to reflect the vector's infinity characteristic A Vector object is never modified after creation; thus, it can be used multiple times.
weight, rWeight	A weight specifies a value for the shade of black to white of a plane or a line segment, or perhaps the width of a line. Weight can be defined from a range of 0 to the maximum value. A special version of weight, called rWeight, performs addition and subtraction of weights arithmetically.

Annex D: Legacy Methods

The functions listed in this annex contain legacy behavior, i.e., they accept agnostic shape dictionaries as input or produce them as output. These will eventually be phased out from the API but may be used to understand the underlying structure of data conversion from Rhino GUIDs to *sortal* individuals. The only exception to this is 'sortal_setup'. This is rather a convenience function that helps to setup the *sortal* library. However, it makes assumptions that may not fit every user, as such it is only meant as an example and not part of the SortalGI API.

Summary of all methods

NAME	PURPOSE
draw_result	Draws one rule application from list of rule applications, based on the index number given by the user; this function goes together with find_rule_appns_ag as it extracts the agnostic dictionary of the rule application from the output of find_rule_appns_ag
draw_results	Draws all rule applications from list of rule applications; this function goes together with find_rule_appns_ag as it extracts the agnostic dictionary of the rule application from the output of find_rule_appns_ag
find_rule_appns_ag	Generates the rule applications from a given rule-shape combination; takes as input rule object or rule name, subshape name/agnostic dictionary and mainshape name/agnostic dictionary; returns list of <transformation matrix, match, result, shape after application> as agnostic dictionaries
create_shape_ag	Creates a shape object from the following inputs: shape name, shape agnostic object/ Rhino geometry, target sort type; shape is registered in form register under its name
convert_to_agnostic	Converts a list of Rhino GUIDs to their agnostic dictionary counterpart, based on the target sort type specified by the user; the target sort type input may be left blank if there is only one functioning geometric disjunctive sort active in 'rhino_shapes_c'
maximalize_ag	Maximalizes an agnostic object/set of Rhino geometries (if the latter, redraws the Rhino geometries)
sortal_setup	Sets up default sortal library background data, i.e. geometric sorts and their attributes; this must be run at the start whenever using the sortal library; to enable external descriptions, input a list of strings, where each element is the name of a description sort type the user wishes to create

draw_result

Draws a single result (identified by the index input 'choice') in the Rhino workspace. The default is the first item of the ruleAppns list. When a reference point is given (tuple/list/vector/point), the shape GUIDs are moved to the reference point. The default reference point is the origin.

Syntax

```
sgi.draw_result(choice = 0, ruleAppns = [], refPoint =
Rhino.Geometry.Point3d(0,0,0), hide = True, layerNameMain = 'Default', shapeIds
= [])
```

Parameters

- choice: Index (integer) of chosen result from list of rule applications (tuple of agnostic dictionaries coming from find_rule_appns_ag)
- ruleAppns: List of rule application tuples of agnostic dictionaries (Non-parametric: (transformation matrix, match, result, shape after application), Parametric: (match, result, shape after application)) to retrieve the shape to be drawn from
- refPoint (optional): Tuple / list of integers or floats / vectors / points (3Ds)),
- hide: Boolean Value; True (default) hides the resulting Rhino geometries; False leaves them visible in the Rhino Viewport
- layerNameMain: name of layer to draw Rhino GUIDs of result on,
- shapeIDs: List of shape GUIDs of original shape (if these are to be deleted and replaced)

Returns

- List of Rhino Geometry [shape GUIDs]: If successful
- None: If unsuccessful or inputs are incorrect

Warnings & Errors

- Warning: Choice input is an integer greater than number of available rule applications
- ValueError: No rule applications entered, OR target layer name is not present in layer names, OR shapeIds is not a list OR value of hide is not a Boolean value
- TypeError: choice input is not an integer OR ruleAppns is not a list OR data type of reference point is not a tuple/list of three numbers/Rhino point
- KeyError:
  - If the number of elements in the reference point tuple or list is greater than / less than 3
  - If 'rhino_shapes_c' is not present in the sort register, i.e. sortal library has not yet been set up

Example

```
rule_name = ‘test_rule’
selected_application_index = 0
ruleAppns = sgi.find_rule_appns_ag(rule_name, ‘mainShape’, subshape = None,
Rhino.Geometry.Point3d(0,0,0))
newGUIDs = sgi.draw_result(selected_application_index = 0, ruleAppns, hide =
False, layerName = ‘TestLayer’)
```

This will generate the new GUIDs of a single rule application drawn in the Rhino workspace. Since 'hide' is set to False, the result will be visible in the viewport, and stored on the layer called 'TestLayer'.

draw_results

Draws all the resulting agnostic objects of the ruleAppns list in the Rhino workspace. Each result's drawing is drawn on top of one another, with the location of the plotting based on the location of the original shape the rule was applied onto.

When a reference point is given (tuple/list/vector/point), the shape GUIDs are moved to the reference point, but are still drawn on top of one another. The default reference point is the origin. The Rhino GUIDs are, by default, hidden as soon as they are drawn in the Rhino viewport.

Syntax

```
sgi.draw_results(rule_Appns, refPoint = Rhino.Geometry.Point3d(0,0,0), hide =
True, layerNameMain = 'Default', shapeIds = [], prnt = False)
```

Parameters

- ruleAppns: List of rule applications as agnostic objects in tuples resulting from use of find_rule_appns_ag
- refPoint: Rhino GUID, tuple or Rhino point geometry where results will be plotted in reference to
- hide: Boolean value; True (default) - hide Rhino geometries from viewport, False - show Rhino geometries in viewport
- layerNameMain: Text string; layer name of layer where resulting geometries are to be placed
- shapeIDs: List of Rhino GUIDs to delete and to be replaced by Rhino GUIDs coming from draw_results
- prnt: Boolean value; True - print description individuals belonging to resulting shape, False (default) - description individuals not printed

Returns

- List of Rhino Geometry [GUIDs]: If successful; geometries are hidden from the viewport itself
- None, if function is unsuccessful or inputs are incorrect

Warnings & Errors

- ValueError: No rule applications entered OR target layer name is not present in layer names OR shapeIds is not a list OR value of hide is not a Boolean value
- TypeError: If ruleAppns is not a list OR data type of reference point is not a tuple/list of three numbers/Rhino point
- KeyError:
    - if the list of rule applications is empty
    - If 'rhino_shapes_c' is not present in the sort register, i.e. sortal library has not yet been set up

Example

```
rule_name = 'test_rule'
rule_Appns = sgi.find_rule_appns_ag(rule_name, 'mainShape', subshape = None,
Rhino.Geometry.Point3d(0,0,0))
new_GUIDs = sgi.draw_results(rule_Appns, Rhino.Geometry.Point3d(0,0,0), hide =
True)
```

This will generate all the ruleAppns in the Rhino workspace and draw them on top of one another, but they will immediately be hidden from the viewport as 'hide' is set to True. To show the GUIDs:

```
rs.ShowObjects(new_GUIDS)
```

find_rule_appns_ag

Finds rule applications from a subshape and shape, or from a set of GUIDs. It returns a list of tuples of agnostic dictionaries, corresponding to the LHS shape (the match within the main shape), the RHS shape (the result after the match shape has been replaced with the RHS of the rule), and the entire shape after the rule has been applied onto it. In cases where the rule is non-parametric, the transformation matrix is also returned; if the rule is parametric, no transformation matrix is returned.

Syntax

```
sgi.find_rule_appns_ag(chosenRule, shape, subshape = None, refPoint =
Rhino.Geometry.Point3d(0,0,0))
```

Parameters

- chosenRule: Rule name as text string
- shape: Name of main shape as text string as recorded in sortal library's form register OR agnostic dictionary pertaining to shape OR Rhino GUIDs composing shape (predicates and directives will not be considered)
- subshape: Name of subshape as text string as recorded in sortal library's form register (optional) OR agnostic dictionary pertaining to subshape OR Rhino GUIDs composing subshape (predicates and directives will not be considered)
- refPt: GUID of point, tuple, or Rhino Geometry point of reference point which will serve as the 'origin' with which shapes will be plotted in respect to (in principle, a vector; optional)

Returns

- List of applications (in tuple form) as based on match generated by subshape or main shape, if no subshape input is given
- An empty list, if no rule applications are found
- Each element of the list is a tuple in the following format:

  a. Non-parametric:

    (<transformation matrix>, <LHS match in agnostic form based on subshape>, <RHS result in agnostic form>, <Main Shape after rule application in agnostic form>)

  b. Parametric:

(<LHS match in agnostic form based on subshape>, <RHS result in agnostic form>, <Main Shape after rule application in agnostic form>)

Warnings & Errors

- ValueError: If rule name is not present in the rule register or if the input for chosenRule is neither a rule object nor a rule name
- TypeError: If initial shape or subshape is not a list of GUIDs or a text string or an agnostic dictionary or is empty
- KeyError: If initial shape or subshape input's name is not present in the form register or if 'rhino_shapes_c' is not present in the sort register, i.e. sortal library has not yet been set up
- Warning: If subshape is not part of shape -> exits function and returns False

Example

```
rule_name = 'test_rule'
mainshape_name = 'new_shape'
subshape_name = 'new_shape_sub'
refPoint = rs.GetObject('Select reference point')
rule_appns = sgi.find_rule_appns_ag(rule_name, mainshape_name, subshape_name,
refPoint)
```

Depending on if the rule corresponding to 'rule_name' is parametric or non-parametric, the rule_appns list will contain either three or four items per element of the list, corresponding to each rule application.

An empty list is returned if no rule applications can be found.

create_shape_ag

Creates a sortal shape from a collection of Rhino GUIDs. It registers the shape according to the input for the shapeName variable in the sortal library. The function returns the predicate/directive dictionaries.

Syntax

```
sgi.create_shape_ag(shapeName = None, shapeData = None, descriptions = '',
classification = None, refPoint = rg.Point3d(0,0,0), prnt = False)
```

Parameters

- shapeName: Name of shape (text string)
- shapeData: List of Rhino GUIDs or an agnostic dictionary that will compose the sortal shape; if an agnostic dictionary is given as input, then any input to the descriptions string will be ignored.
- descriptions: Optional text string of descriptions to include in the sortal shape (if these are not yet present in the agnostic dictionary), e.g.

  ```
  'label1@("A2", 1, 1234);("A3", 4, 1234)|label2@("A1", 1, 1234)|
  label2@("A9", 8, 1234)'
  ```

  where there are two description types, 'label1' (followed by an ampersand "@"; 2 individuals separated by a semicolon ";"), and 'label2' (2 individuals); declaration of different description types and their individuals is separated by a vertical dash "|"

- classification: Name of target disjunctive sort type (text string); this sort type must already be present in the sort register under the compound sort 'rhino_shapes_c'
- refPoint: Reference point to serve as 'origin' point for shape; the default origin point is 0,0,0
- prnt: Boolean value (True/False); True - prints the resulting sortal shape according to the output of the 'printE' command in the sortal library, False (default) - does not print anything

Returns

- A sortal shape using the disjunctive sort type specified in the 'classification' variable is created in the form register and stored under the contents of 'shapeName'.
- This function returns the predicates and directives dictionaries obtained from the Rhino geometry input 'shape_data'. They will have the following keys inside:
- predicates: Predicates dictionary

  ```
  {'max': [], 'shortest': [], 'longest': [], 'bound': [], 'nolabel':[]}
  ```
- directives: Directives dictionary

  ```
  {'pointOL':[], 'distance':[], 'normal':[]}
  ```

Warnings & Errors

- TypeError: If shapeName input is not a text string, or if shapeData input is not an agnostic dictionary or a list of Rhino GUIDs, or if shapeData has any elements that are not Rhino GUIDs inside
- Warning: If descriptions input is not a text string
- KeyError: If 'rhino_shapes_c' is not present in the sort register, i.e. sortal library has not yet been set up

Example

```
shape_data = rs.GetObjects('Select shape')
shape_name = 'test_shape'
descriptions = 'label1@("A2", 1, 1234);("A3", 4, 1234)|label2@("A1", 1, 1234)|
label2@("A9", 8, 1234)'
classification = 'N3D'
```

This uses the non-parametric 3D disjunctive default sort type within rhino_shapes_c (called 'N3D') if sgi.sortal_setup() was used to do sortal library setup or a similarly named sort exists in the sort register.

```
predicates, directives = sgi.create_shape_ag(shape_name, shape_data,
descriptions = '',  classification)
```

convert_to_agnostic

Converts a list Rhino GUIDs to an agnostic dictionary based on the structure of the target sort named in the 'classification' variable (this sort type must have already been created and should be present in the register). If descriptions are given, then it will include those descriptions into the agnostic dictionary. By default, the origin is the reference point. However, the list of Rhino GUIDs may be processed with an optional reference point other than the origin

Syntax

```
sgi.convert_to_agnostic(shapeRhino = [], classification = None, descriptions =
'', refPoint = Rhino.Geometry.Point3d(0,0,0))
```

Parameters

- shapeRhino: List of Rhino GUIDs
- classification: Target sort type for the shape
- descriptions: Optional text string of descriptions to include in the sortal shape (if these are not yet present in the agnostic dictionary), e.g.

  ```
  'label1@("A2", 1, 1234);("A3", 4, 1234)|label2@("A1", 1, 1234)|
  label2@("A9", 8, 1234)'
  ```

  where there are two description types, 'label1' (followed by an ampersand "@"; 2 individuals separated by a semicolon ";"), and 'label2' (2 individuals); declaration of different description types and their individuals is separated by a vertical dash "|"
- refPoint: Reference point which the coordinates of the shape will be subtracted from; this functions as the 'origin' of the shape, and can be used for defining shapes that compose rule sides so that they can be drawn side by side one another in the Rhino viewport

Returns

Four outputs, in the following order:

- Agnostic object form of [Rhino GUIDS + description type; classification is used as the key for the dictionary if there is more than one type of disjunctive sort within 'rhino_shapes_c' that is not a dummy disjunctive sort] as agnostic dictionary
- Dictionary of GUIDs corresponding to geometry stored in agnostic object, according to their target sort type
- Dictionary of predicates
- Dictionary of directives

Warnings & Errors

- TypeError: If shapeRhino is not a list of Rhino GUIDs OR if 'descriptions' is not None or a text string
- Warning: If conversion was unsuccessful
- KeyError: If 'rhino_shapes_c' is not present in the sort register, i.e. sortal library has not yet been set up

Example

```
shape_data = rs.GetObjects('Select shape')
classification = 'P3D' # parametric 3D disjunctive sort
descriptions = 'label1@("A2", 1, 1234);("A3", 4, 1234)|label2@("A1", 1, 1234)|
label2@("A9", 8, 1234)'
refPoint = Rhino.Geometry.Point3d(10,10,0)
shape_agnostic, shape_guids_dict, predicates, directives =
sgi.convert_to_agnostic(shapeData, classification, descriptions, refPoint)
```

maximalize_ag

Maximalizes an agnostic dictionary or list of Rhino GUIDs. In the case of inputting a list of Rhino GUIDs, the output is automatically a list of the maximalized Rhino GUIDs.

Syntax

```
sgi.maximalize_ag (shape, target = None, rhino = False, hide = False, delete =
False)
```

Parameters

- shape: Shape data in the form of a list of Rhino GUIDs (target disjunctive sort for maximalized must be provided) OR an agnostic dictionary OR the shape name as a text string
- target: Target disjunctive sort for shape to be maximalized (text string)
- rhino: Boolean value (True/False); True - returns list of Rhino GUIDs corresponding to maximalized shape; False (default) - returns agnostic dictionary form
- hide: Boolean value (True/False); True - hides Rhino geometry output from viewport; False (default) - leaves Rhino geometry output visible in viewport
- delete: Boolean value (True/False); if the inputs are a list of Rhino GUIDs, then True deletes the inputs, and False (default) leaves the inputs still in the Rhino viewport

Returns

- maxShape: Maximalized agnostic dictionary/list of Rhino geometries

Warnings & Errors

- TypeError: If type of input is invalid (i.e. not a list of Rhino GUIDs, an agnostic dictionary or a shape name text string)
- ValueError: If input is an empty list or if shape name doest not exist in the form register
- KeyError: If 'rhino_shapes_c' is not present in the sort register, i.e. sortal library has not yet been set up

Example

The function call below returns the maximalized agnostic dictionary version of 'shape_agnostic'. If there is only one active geometric disjunctive sort in 'rhino_shapes_c', then the variable target can be left as its default value, as below:

```
shape_agnostic, shape_guids_dict, predicates, directives =
sgi.convert_to_agnostic(shapeData, classification, descriptions, refPoint)
sgi.maximalize_ag (shape_agnostic)
```

This function call draws the maximalized shape corresponding to the data inside 'shape_agnostic' in the Rhino viewport, and hides them afterwards; it returns the list of GUIDs corresponding to this maximalized shape.

```
sgi.maximalize_ag (shape_agnostic, rhino = True, hide = True)
```

This function call draws the maximalized shape corresponding after convert the data inside 'shape_rhino' to sortal data, and leaves the new drawing visible in the viewport; it returns the list of GUIDs corresponding to this maximalized shape, and deletes the Rhino GUIDs of the input list 'shape_rhino'

```
shape_rhino = rs.GetObjects('Select shape data')
```

```
sgi.maximalize_ag(shape_rhino, hide = False, delete = True)
```

sortal_setup

Sets up the sort types in the sortal library, as well as any external description sort types, according to the list of string inputs in the variable 'descriptions'. The code in this function also serves as an example on how to create sorts through hardcode, from primitive to compound sort types and to imbue geometric primitive sorts with attributes.

It may be run without any inputs; this sets up a default compound sort with two disjunctive sorts inside, one of which is a dummy disjunctive sort composed of two dummy description sorts. This dummy disjunctive sort is removed from the compound sort 'rhino_shapes_c' once other sort types are introduced, when using 'read_sdl' to import SDL file data.

All geometries (points, line segments, plane segments, circles, ellipses, circular arcs, Bezier curves) are enabled in this default disjunctive sort as non-parametric geometries, if no input is given. Due to the limitations of the library, parametric 2D and 3D circular and elliptical arcs are not available.

Labels, weights, descriptions (one type called 'desc'), weight255 (for grayscale), enumeratives and colors are enabled as attributes to all the non-parametric geometries. As for external descriptions, they may be added to the default setup by inputting a list of text strings corresponding to the names of the intended description sort types.

The names of the default sort types in any resulting agnostic dictionary that follows from using sgi.sortal_setup are as follows:

- Default: “N3D”
- Compound sort: rhino_shapes_c
- Disjunctive sort: rhino_shapes
- Attribute sorts under disjunctive sort:
  - Points - 'points'
  - Line segments - 'line_segments'
  - Plane Segments - 'plane_segments'
  - Circles - 'circles'
  - Ellipse - 'ellipses'
  - Circular arcs - 'arcs'
  - Bezier Curves - 'bezier_curves'
  - Attributes enabled for all sorts: label, description, color, weight, weight255, enumerative

Syntax

```
sgi.sortal_setup(descriptions = [], parametric = False, dimension = True)
```

Parameters

- descriptions: List; list of description sort type names

- parametric: Boolean; indicates if sort types are to be non-parametric or parametric; default is False (non-parametric)
- dimension: Boolean; indicates if sort types are to be 2D or 3D; default is False (3D)

Returns

- True: If successful
- False: If unsuccessful

A sort based on the conditions given to 'sortal_setup' is created in the back end and stored under the compound sort 'rhino_shapes_c'. If the geometric disjunctive sort is the only one active, it is stored with a dummy disjunctive sort composed of two dummy description sorts in the compound sort. The following names are given to the disjunctive sorts based on the given parametric-dimension combinations:

- N3D – non-parametric, 3D
- N2D – non- parametric, 2D
- P3D – parametric, 3D
- P2D – parametric, 2D

Warning/Errors:

- None

Example

When creating a parametric 3D sort with no external descriptions

```
sgi.sortal_setup([], True)
```

OR when enabling external description types

```
sgi.sortal_setup(['description_1', 'description_2'])
```

This will include two description sort types, one named 'description_1' and the other named 'description_2' in the resulting disjunctive sort called 'N3D'.