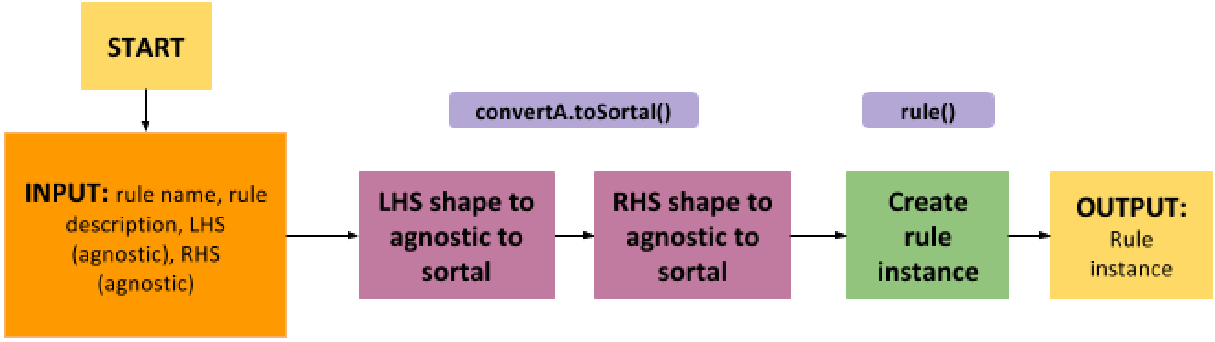
Annex F: More Information on SortalGI functions

This section further explains on how some of the functions available in 'sortalgi'.

- create_rule
- find_rule_appns
- maximalize
- partOf

create_rule

Internally, 'create_rule' has the following steps:


START
INPUT: rule name, rule description, LHS (agnostic), RHS (agnostic)
convertA.toSortal()
LHS shape to agnostic to sortal
RHS shape to agnostic to sortal
rule()
Create rule instance
OUTPUT: Rule instance

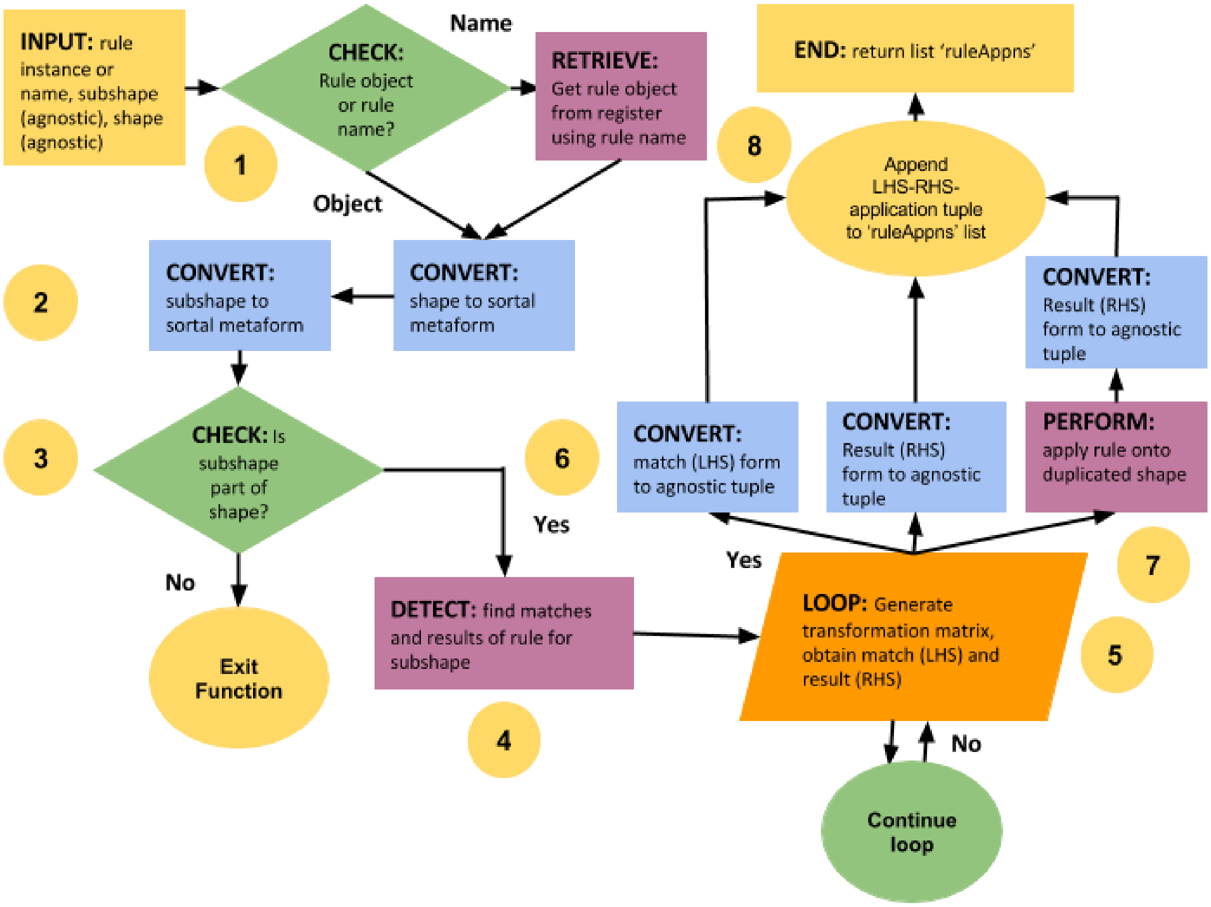
1. Convert agnostic tuple form to sortal metaform for each side.

2. Create the rule object using the rule sides, rule name and rule description as input for the sortal class 'rule'.

3. Return rule object and the corresponding rule name.

find_rule_appns

This function generates rule applications for a given rule, shape and subshape (optional). The detailed workflow for how 'find_rule_appns' works is described in the following flowchart.


INPUT: rule instance or name, subshape (agnostic), shape (agnostic)
1
CHECK: Rule object or rule name?
Name
Object
RETRIEVE: Get rule object from register using rule name
END: return list 'ruleAppns'
8
Append LHS-RHS-application tuple to 'ruleAppns' list
2
CONVERT: subshape to sortal metaform
CONVERT: shape to sortal metaform
CONVERT: Result (RHS) form to agnostic tuple
3
CHECK: Is subshape part of shape?
6
Yes
CONVERT: match (LHS) form to agnostic tuple
CONVERT: Result (RHS) form to agnostic tuple
PERFORM: apply rule onto duplicated shape
Yes
7
No
Exit Function
DETECT: find matches and results of rule for subshape
LOOP: Generate transformation matrix, obtain match (LHS) and result (RHS)
5
4
No
Continue loop

1. *Checking rule input*

The input for the rule can be either the rule object itself or the rule name. If the input is a rule object, then the function proceeds to the next step. If the input is a string, it is checked against the rule register (a dictionary of rule objects stored in the sortal library) to see if the rule already exists. If the rule name is not present in the rule register, the function does not proceed and an error is raised in the form of a message box.

```
try:
    if chosenRule.type == rule.type:
        pass
except:
    if type(chosenRule) == type('a'):
        if chosenRule in rule.register:
            …
        else:
            message = ('Rule name is not present in the rule register.')
            rs.MessageBox(message)
    else:
        message = ('Rule input is neither a rule object or a rule name.')
        rs.MessageBox(message)
```

2. *Conversion from agnostic to sortal form*

   The agnostic forms (subshapeAgnostic and shapeAgnostic) are converted into sortal metaforms. This example can be found in the function definition of find_rule_appns (sortalgi -> __init__)

```
shapeSortal = convertA.toSortal(shapeAgnostic)
subshapeSortal = convertA.toSortal(subshapeAgnostic)
```

   'convertA.toSortal' is further explained in [Annex B: Conversion Classes](#).

3. *Checking for subshape and shape relationship*

   The subshape is checked if it is part of the larger shape, before proceeding with generating matches. This is done using the method <metaform instance>.partOf(<other metaform instance>

```
if subshapeSortal.partOf(shapeSortal):
        …
```

   If the subshape is not part of the larger shape, the function ends. This particular 'partOf' method is a built-in function for sortal objects.

4. *Subshape match detection*

   The following line of code corresponds to this step:

```
subshapeApps = chosenRule.detect(subshapeSortal)
```

   where chosenRule is the rule object (made using 'create_rule' function or retrieved from the sortal library rule register), and subshapeSortal is the sortal metaform of subshapeAgnostic respectively.

   The '<rule name>.detect' method generates match (LHS), result (RHS) and rule application combinations inside the rule object in the sortal library.

5. *Appending relevant match and result pairs*

   If the subshape sortal metaform is part of the shape sortal metaform, a loop structure generates and appends LHS-RHS-application combinations of the subshape (which are less than that of the larger shape) to the list 'ruleAppns', which will be returned as output.

```
for index1 in range(len(subshapeApps)):
    …
```

6. *Converting sortal forms of match and result to agnostic forms*

   If the condition of the if-else statement in step 4 is fulfilled, the transformation matrix, match (LHS), result(RHS) and rule application (the shape after the rule has been performed on it) are converted to agnostic form and then appended as a tuple of four elements to the list ‘ruleAppns’, which is returned. This list may contain more than one combination/element.

```
matrix =
subshapeApps[index1].transformSet.retrieveTransform(affineTransform3D).mat
rix
match = convertS.toAgnostic(subshapeApps[index1].matchForm)
result = convertS.toAgnostic(subshapeApps[index1].resultForm)
```

   ‘convertS.toAgnostic’ changes the sortal metaform to an agnostic tuple; it takes as input a metaform and outputs the corresponding agnostic form. More information about ‘convertS.toAgnostic’ can be found in Annex B: Conversion Classes.

   Further focus is given on generating the post-rule application sortal metaform for the main shape in the next step. This is also appended as part of the LHS-RHS-rule application tuple.

7. *Performing rule on a shape*

   To perform a rule, the following method is used:

   <rule instance name>.detection.applications[index].perform(<metaform instance>

   In ‘find_rule_appns’, the rule is performed and then immediately converted from sortal metaform to agnostic tuple.

```
after =
convertS.toAgnostic(shapeApps[index2].perform(shapeSortal.duplicate()))
```

   Note that the method ‘shapeSortal.duplicate()’ creates a duplicate of the shape sortal metaform to perform the rule on. This is because if the original shapeSortal is used in this part, then it will be altered permanently and the previous LHS-RHS-rule applications will no longer apply to the shape.

8. Appending match-result-after rule application tuple

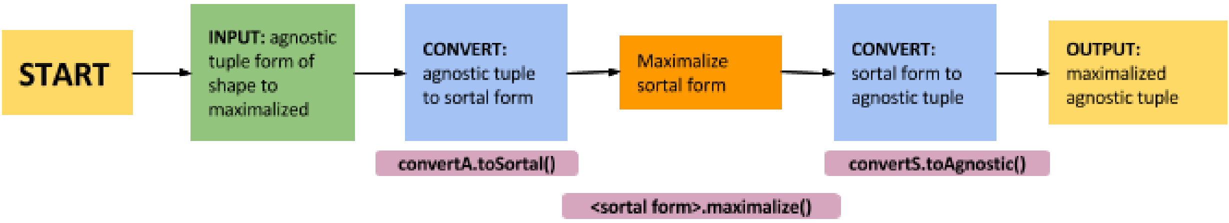
   The LHS-RHS-rule application combination is appended as a tuple to the list ‘ruleAppns’. The final list is returned as the output of the function ‘find_rule_appns’.

```
ruleAppns.append([matrix, match, result, after])
```

maximalize

This function maximalizes an agnostic object. This function first converts the agnostic object to sortal form, maximalizes it using the sortal library function ‘<shape sortal form>.maximalize()’, then converts it back into agnostic tuple form and returns the maximalized agnostic tuple.

Internally, the function has the following steps:


START
INPUT: agnostic tuple form of shape to maximalized
CONVERT: agnostic tuple to sortal form
Maximalize sortal form
CONVERT: sortal form to agnostic tuple
OUTPUT: maximalized agnostic tuple
convertA.toSortal()
convertS.toAgnostic()
<sortal form>.maximalize()

1. *Converting agnostic tuple to sortal form*

   The agnostic tuple form input is converted to sortal form via the following function:

   ```
   shapeSortal = convertA.toSortal(shapeAgnostic)
   ```

   where ‘shapeAgnostic’ is an agnostic tuple. ‘shapeSortal’ is the sortal form returned by ‘convertA.toSortal’, which is further explained in Annex B: Conversion Classes.

2. *Maximalizing sortal form*

   The sortal form has a built-in function used to maximalize it. This is called to maximalize the geometry inside the sortal form, and alters the geometry directly:

   ```
   shapeSortal.maximalize()
   ```

3. *Converting sortal form to agnostic tuple*

   ‘shapeSortal’ is altered directly by the built-in maximalize method, so it is used as input to get the agnostic tuple form.

   ```
   shapeAgnostic = convertS.toAgnostic(shapeSortal)
   ```

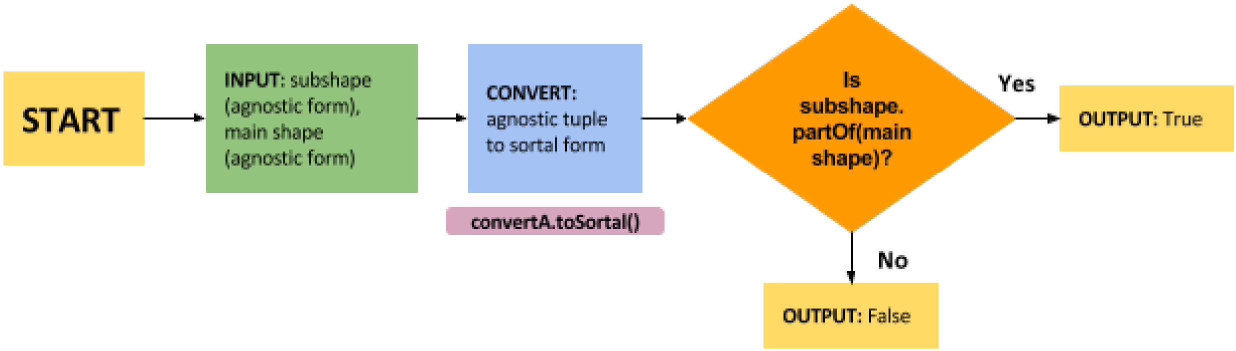
4. Outputting maximalized agnostic tuple

The maximalized agnostic tuple stored in ‘shapeAgnostic’ is returned by the function.

```
return shapeAgnostic
```

partOf

This function checks if an agnostic object is part of another agnostic object. The flowchart describes the steps taken by this function.


START
INPUT: subshape (agnostic form), main shape (agnostic form)
CONVERT: agnostic tuple to sortal form
convertA.toSortal()
Is subshape. partOf(main shape)?
Yes
OUTPUT: True
No
OUTPUT: False

1. Converting agnostic tuples to sortal form

The agnostic tuples are converted to sortal form via the following function:

```
shapeSortal = convertA.toSortal(shapeAgnostic)
```

```
subshapeSortal = convertA.toSortal(subshapeAgnostic)
```

where ‘shapeAgnostic’ and ‘subshapeAgnostic’ are agnostic tuples. The method ‘convertA.toSortal’ is further explained in Annex B: Conversion Classes.

2. Checking if subshape sortal form is part of main shape sortal form

The subshape is checked if it is part of the larger shape using the method <metaform instance>.partOf(<other metaform instance>. For example:

```
result = subshapeSortal.partOf(shapeSortal)
```

If the subshape is part of the larger shape, the function returns True; False, if otherwise. This behavior is used as the returned output for this function:

```
return subshapeSortal.partOf(shapeSortal)
```