Combining Geometries and Descriptions

A shape grammar plug-in for Grasshopper

Bianchi Dy\textsuperscript{1}, Rudi Stouffs\textsuperscript{2}
\textsuperscript{1,2}National University of Singapore
\textsuperscript{1,2}{akidribm|stouffs}@nus.edu.sg

A persistent challenge to the more widespread use of shape grammars in architectural research is the creation of rules and rule sets for application in design contexts, while leaving space for design creativity despite the limitations of a rule-based system. A hybrid of associative and rule-based approaches may alleviate this. We present one such development, a Grasshopper shape grammar plug-in that embeds a rule-based approach within a parametric modelling environment. It supports shape emergence, visual enumeration of rule application results, and the parametric definition of shapes and shape rules even when selecting a non-parametric rule matching mechanism. Grasshopper's ability to handle geometries and text together allows for external descriptions and labels as attributes to points, enabling definition and application of compound, geometric and description rules. Well-known examples from shape grammar literature are implemented using the plug-in, with a focus on rule definition and application in the context of interaction between the parametric modelling environment and the rule-based interpreter, and simultaneous use of geometry, descriptions, and descriptions as attributes in rules.

Keywords: shape grammar, shape grammar interpreter, parametric modelling, Grasshopper, rule-based, descriptions

INTRODUCTION

Research into parametric or associative modelling goes back at least as far as Sutherland's Sketchpad (Sutherland, 1963). Research into shape grammars is only slightly younger (Stiny and Gips, 1971) but does not yet enjoy the same success, with one possible reason being the persistent challenge of structurally creating rules and rule sets for application in design contexts. The hybrid of associative and rule-based approaches may alleviate this challenge.

We present one such development, a Grasshopper shape grammar plug-in that embeds a rule-based approach within a parametric modelling environment. It supports shape emergence, visual enumeration of rule application results, and allows shapes and shape rules to be defined parametrically, even when adopting a non-parametric rule matching mechanism. Additionally, the plug-in supports parametric rule matching and the specification of descriptions, enabling definition and application of descrip-
tion rules in tandem with (parametric) shape rules. This paper focuses on rule application in the context of interaction between the parametric modelling environment and the rule-based interpreter, and the simultaneous use of geometry and descriptions, both external and as attributes to geometry, in rules using the plug-in. Relevant examples are presented.

SORTAL GRAMMARS

Shape grammars come in a variety of forms, with labeled shapes (Stiny, 1980), i.e., line segments with labeled points, as the most basic formalism. “Sortal grammars” integrate a wide variety of forms and act as a class of formalisms for shape grammars, using “sortal structures” as representational building blocks (Stouffs and Krishnamurti, 2001; Stouffs, 2012). These structures are defined as formal compositions of other primitive, sortal structures (also termed “sorts”). Component sorts specify a partial order relationship on their individuals and forms, thus defining both matching and arithmetic operations for rule applications. Shapes may be either objects or attributes in object-attribute relationships, or both (or neither, but this case would not count as a shape grammar).

The organizing structure of sortal grammars enables the implementation of different kinds of shape grammars, not limited to specific formalisms, design languages or contexts. As such, using sortal structures enables a modular implementation of a generalized, flexible shape grammar interpreter for different grammar forms. This sortal shape grammar interpreter (“SortalGI”) has been developed as a library and an API in the Python programming language, with graph-based representation for parametric shapes, and combinatorial enumeration of potential matches. It supports both parametric and non-parametric shape grammars, including line segments, plane segments, points, circular arcs, Bezier curves, labels, weights, colors, enumerative values, and (parametric) descriptions, in 2D and 3D.

The shape grammar interpreter is accessible in a Python development environment; within the Rhino 3D modeling environment [1] through an API; and as a Rhino/Grasshopper plug-in [2], requiring no programming or scripting. Though the entire library is available within the Rhino 3D modeling environment, the current API does impose a few restrictions. The Grasshopper plug-in is currently limited to parametric and non-parametric points, line segments, plane segments, circles, ellipses, Bezier curves, labeled (description) points, non-parametric circular arcs, and descriptions.

THE SORTALGI GRASSHOPPER PLUG-IN

The SortalGI Grasshopper plug-in makes part of the SortalGI library and interpreter available within Grasshopper. As such, shape rules may be defined parametrically even if the selected rule matching mechanism is non-parametric, as illustrated in the discussion. Text descriptions may also be defined as part of rules or shapes alongside geometry, with dedicated components for single or multiple rule application generations and support for visually enumerating them. All plug-in components are discussed in this section.

Shapes

The plug-in defines both a rule object and a shape object. Grasshopper geometries (and text descriptions) are stored in shape objects as sortal data structures, as Grasshopper geometries do not maintain text objects very well.

- The basic Shape component only accepts points, line segments, surfaces, polylines and labeled points. The resulting shape object and its corresponding geometry are returned.
- A dShape variant also accepts standalone text descriptions (collected through a Panel component). Text descriptions are distinguished by description type; these types are predefined as part of the set-up component (see section Setup). Descriptions are embedded in the shape object; these are not visualized in Rhino or as geometry but are separately accessible as a list of text.
• Both prior components accept an optional reference point, which moves the shape from the reference point to the origin and is useful for drawing the left-hand-side and right-hand-side of a rule beside one another rather than overlapping.

• *Text Point* creates a Rhino text object from a point and a single-line text. This serves as a labeled point (when the text is enclosed within double quotes) or a description point input to *Shape*.

• *S2G* visualizes shape objects as their corresponding geometry. Any descriptions embedded in the shape object that are not part of labeled points are separately returned as a list and can be read using the *Panel* component.

Geometric operations such as *Move* and *Rotate* are redefined as Grasshopper components acting upon shape objects. Each component accepts a shape object and any additional data required to inform and apply the transformation, such as a vector to move a shape along, and a base plane and angle to rotate a shape. There are also components to find the union (*Sum*), intersection (*Product*) or complement (*Difference*) of two shapes.

**Rules**

A rule object may constitute either a shape rule, parametric or non-parametric, or a compound rule combining a shape rule with one or more description rules (Stiny, 1981; Stouffs, 2018).

• The *Rule* component constructs a non-parametric rule object from a rule name and brief explanation of the rule, a left-hand-side shape object and a right-hand-side shape object. Any mismatch in description types, i.e. if present in one rule side and absent from the other rule side, is automatically resolved by adding an empty description of that type to the relevant side’s shape object within the rule. While the right-hand-side of a rule may be left empty, the left-hand-side must have a minimum set of geometry and/or descriptions present.

• The *pRule* takes the same inputs as the *Rule* component but returns a parametric rule object. Parametric rules have different minimum requirements for the left-hand-side geometry from non-parametric rules.

• Both *Rule* and *pRule* outputs are accepted as ‘rule’ inputs and can be deconstructed by the *Rule Info* component, which returns a rule’s left-hand-side and right-hand-side shape objects, and a multi-line text consisting of the rule’s GUID component, rule name and rule description.

Rules can be applied onto shapes by connecting the relevant rule component corresponding to a rule object to the *Apply*, *Apply All*, or *Derive* components (see section Rule Applications). Alternatively, the *Get Rule* component accepts a rule name or names (text separated by line breaks, through the *Panel* component) and returns the corresponding rule object(s). This output may serve as rule input for all rule application components.

**Rule Applications**

The general process for applying a rule starts with detection and matching of the rule’s left-hand-side onto the target shape. Should a match or multiple matches be found, rule application involves subtracting the part corresponding to the match from the target shape, after which the right-hand-side is added under the same transformation as the left-hand-side is matched. This transformation is a similarity transformation (translation, rotation, reflection and/or uniform scaling) for a non-parametric rule. In the case of a parametric rule, matching relies on a combinatorial enumeration of potential matches on the basis of distinguishable elements, that is, the infinite carriers of line segments and their intersection points and the ordering of intersection points (and endpoints) along the infinite carrier. As such, there is no explicit transformation, and sufficient correspon-
dence must exist between the rule’s left-hand-side and right-hand-side in order to allow rule application to succeed.

There are three plug-in components to apply a rule onto a shape:

- The Apply component returns a single rule application that can be chosen from all possible rule applications by inputting an optional rule application index (the application is randomly selected, if unspecified).
- The Apply All component yields the output from all possible rule applications in a list.
- The Derive component accepts a list of one or more rule objects, possibly retrieved through the Get Rule component. The rules are applied in sequence, each on the result of the preceding rule application. The desired rule application for each step may be chosen by inputting an optional list of rule application indices. If unspecified, the applications are randomly selected. The component returns the list of shape objects resulting from each step of the derivation.

Next to a rule object and a shape object, all application components accept an optional sub-shape object to limit potential matches, with rule application always performed on the entire shape. For Derive, specifically, a list of sub-shapes may be accepted.

All application components also return a translation vector or list thereof for visualizing the resulting shape(s) aside from the original shape. The original shape object is returned in failure cases, such that any subsequent rule application may still apply.

**Setup**

The plug-in includes a Setup component that initializes the sortal grammar interpreter and controls some global settings for Grasshopper geometry such as the text size of labeled points in the Rhino workspace, and the X and Y displacement values used by the Apply and Apply All components to create translation vector(s), such that using a Move component, the resulting shape can be visualized aside from the original shape at the given distance along the X or Y axis. The displacement values are optional; default X and Y values for the translation vector(s) are defined from the bounding box of the original shape.

More importantly, the Setup component takes a list of description names through the Panel component (separated by line breaks) to predefine the active description types that will be used in rules and shapes. If any description type not present in this list is specified in a rule or shape definition, the respective component raises a warning.

**THE PLUG-IN IN ACTION**

To get started using the plug-in, first install the sortal library and plug-in. Afterwards, place Setup as the first component in the Grasshopper file to initialize
the shape grammar interpreter. Ensure that it runs first before all other SortalGI components by pressing CTRL+B while selecting the component.

If any descriptions will be used in the grammar, these description types must be declared by their type name as string input to Setup; each description type is separated with a line break (Fig. 1). This can also be done later, when description types need to be added as more rules are developed. Other optional values such as displacementX, displacementY and textSize can be set to fixed values or sliders at any time. Changing their inputs will trigger recalculation of all plug-in components.

**Shape Definition**

To create a shape, connect one or more components that represent or collect geometry to the G input of the Shape or dShape component. The relevant geometric elements will be converted to their sortal representation and outputted as shape object S.

For dShape, collect any external descriptions in a Panel and connect this to the D input. String values may also be directly stored in D, but they must be surrounded by double quotation marks ("""") to be read as static strings. Description individuals are declared by stating the description type first, followed by a colon (':') and then the description's value (Stouffs, 2018). Multiple shape descriptions of the same type can be written in one line by separating them with a vertical bar ('|') per description individual (Fig. 2).

The output G of both shape components contains the pruned geometry present in the output shape object S, with D containing the external descriptions. Any output geometry from a plug-in component (except for Text Point) may be used as geometry input to any Grasshopper component, allowing interaction between the plug-in and Grasshopper's other components.

**Rule Definition**

The optional reference point input may be used to reposition the shapes defining the rule sides. In Fig.
3, the reference points are indicated by the cross marks. Specifying a reference point moves the geometry from the reference point to the origin. Moving both the left-hand-side and right-hand-side shapes may relate them in space (through overlapping); their location after movement due to the reference point is what is stored in the output shape object S.

Both parametric and non-parametric rules are created using string inputs for the rule name and rule description, and with rule sides defined by either the Shape or dShape component, or from any S output of another plug-in component.

Rule names should not be repeated within the same Grasshopper file. Both Rule and pRule components will raise an error or warning message if the geometry of the left-hand-side is insufficient or if the rule name has been repeated.

**Rule Application**

To apply a rule onto a shape, connect the rule object as output by a Rule or pRule component and any shape object output to any of the rule application components described in the section Rule Applications. Preview geometry of the result(s) is generated in place; however, translation vectors are generated based on the given displacement values or the bounding box of the input shape. The optional sub-shape input may be used to limit the discovery of rule applications to a specific region of the initial shape.

A variety of approaches can be taken for rule application. Apply or Derive can generate step by step results; Apply All generates all potential rule applications for a given rule and initial shape combination. A short derivation using the two rules from Fig. 3 is shown in Fig. 4, using the Derive component. The same derivation can be achieved using a series of Apply components to generate results step by step.

Conversely, Fig. 5 illustrates the ability of the plug-in to generate and visualize all possible rule applications with Apply All. For viewing clarity, the multiple results of a single component are drawn separate from one another by connecting the relevant rule application component to the Move component.

**APPLICATIONS AND DISCUSSION**

The value of this shape grammar plug-in stems from its ability to support shape computations, rule applications and derivations, and their visual enumeration, within an interaction between parametric or associative modelling and rule-based generation. Moreover, it supports the definition and application of shape and description rules in tandem. In principle, the plug-in only supports 3D shapes, but 2D is naturally included as a degenerate 3D case. Next, we demonstrate a 3D example, using surfaces to represent 3D blocks.

This example is taken after Stiny (1981) and ex-
Figure 6
Nine rules to create designs made up of blocks from Froebel's building gift 6 (after Stiny, 1981, pp. 260-264).

Figure 7
Panel texts showing definitions of description rules corresponding to geometric rules 1, 2, 4 and 7 in Fig. 6 (after Stiny, 1981, pp. 260-264).

Plores the use of label attributes and external descriptions in tandem with geometry to constrain matching and to record textual information about the design. Specifically, the grammar creates designs made up of blocks from Froebel's building gift 6 (after Stiny, 1981, pp. 260-264). The grammar is composed of nine rules; all of them except rules 8 and 9 include description rules. Fig. 6 shows the shape rules 1 through 8, Fig. 7 shows the description rules corresponding to shape rules 1, 2, 4 and 7. Note that Stiny's grammar consisted of 17 rules as he distinguished rules operating in different directions. Instead, the rules presented here automatically capture directional information. Specifically, the description rules contain references to the corresponding shape rules that allow these to extract information from the shape rule application. However, rules 8 and 9 are additional and are due to extra labels. Note also that while Stiny combines various description entities in a single description rule, separating the entities using the hash sign ('#'), for the sake of clarity, these entities are considered here as separate descriptions and description types.

Fig. 8a shows the initial shape, upon which rule 1 is applied iteratively to obtain Fig. 8b. Due to rotational symmetry, there will be four possible applications for rule 1 at each step. The directionality and location of rule applications for rule 1 is limited by
the presence of the labeled points “a” and “b”. The inclusion of a description of type “column_locations” (Fig. 7) in the initial shape is also necessary to ensure matching, as rule 1 must first recognize the presence of this description type in the shape to match successfully.

Aside from constraining the geometric side of the rules, the labeled points “a”, “b”, and “c” serve as links between the geometry and the descriptions. Rule 1 (Fig. 7 and Fig. 8b) records the sequence of coordinate locations visited by the rule using these labeled points and, hence, the complete, step-by-step history of a labeled shape as it is generated by the grammar. This information is subsequently used in rule 2 to collect certain properties of the shape or design, such as the number of pillars, the number of walls and the location of the rooms. In particular, the contents of “column_locations” help generate the contents of “column_count” (the number of unique coordinate tuples in “column_locations”), “wall_count” (the number of unique segments that can be gathered from “column_locations”), and “room_locations” (the loops of coordinates tuples in “column_locations”) (Fig. 8c).

Rules 4 through 6 similarly track the locations of the ‘stiles’, ‘doors’ and ‘windows’ by storing the locations of two points in a tuple (Fig. 7). Rule 7 draws from multiple descriptions to determine adjacency information, i.e., combining “room_locations”, “door_locations” and “stile_locations” to obtain “adjacency_matrix” (the connections between the rooms and the outside through stiles and doors) (Fig. 8d). Thus, a block-by-block description of the design can be derived based on these descriptions.

Obtaining this information from the descriptions is only possible through additional functions such as length, distance, segments, etc. To this end, “room_locations”, “wall_count” and “column_count” rely on description functions like “loops”, “segments”, and “unique” to compute their right-hand-side values based on the coordinates previously recorded in “columns_locations”. Rule 7 (Fig. 7) illustrates how the adjacency matrix can be obtained in Fig. 8d with the “adjacencies” function. The SortalGI library and plug-in support a number of additional descriptions functions.

Defining the shapes and shape rules in this example parametrically, especially their ratios and di-
dimensions, e.g., using sliders, allows the same grammar to apply to varying block dimensions. In general, altering the dimensions of the defining rule shapes may change the number of rule applications or remove the match between the initial shape and a rule.

Fig. 9 and Fig. 10 demonstrate the application of parametric rules in a short derivation based on the ice ray lattice grammar by Stiny (1977). There are five (parametric) rules in this example (Fig. 9). The first rule identifies the initial shape (a quadrilateral) and labels its vertices. The succeeding four rules all behave similarly in that they split the left-hand-side polygon, be it a quadrilateral, triangle or pentagon, into two smaller polygons contained by the original shape.

As the input polygons to the rules are parametrically defined in Grasshopper, altering their shape may affect the result by triggering a recalculation of the rule applications and, thereby, return completely different results from prior computations or potentially increase or decrease the number of matches, thus altering the derivation completely. Changing the locations of the start and end points of the right-hand-side line segments (“ice rays”) will also trigger a recalculation. Specifically, computing new ice rays uses a point-on-line feature of the sortal library to generate new line segments. A new point is randomly generated within a certain range of the line segment and its precise location cannot be predicted ahead of time. This feature allows for creativity in that the derivation is no longer deterministic, but exploratory. However, one drawback is that prior results are lost, as the new applications do not retain any history of previous runs.

The role of labels and descriptions in the examples is mainly to limit rule application, or to contain information not readily expressed by the geometry. Previous iterations of shape grammar interfaces have generally had limited capabilities in combining geometries and descriptions into shapes. The ability of Grasshopper to handle text and geometry together allows for combining shapes and descriptions in shape or rule objects, and as attributes of points (label descriptions attached to points as created by the Text Point component). Furthermore, the two description types may be related, e.g. the label value of a labeled point may be used in the left-hand-side of a description rule, thus enhancing the expressiveness of rule application. This ability is used to great effect in the description type “index” in Fig. 9, which generates the integer label for new polygon vertices, to prevent the polygons from merging and the ice rays from intersecting one another. It is increased with every successful application to create distinguishing polygon attributes; the labeled points at polygon vertices indicate which polygon(s) the vertices are part of (see Fig. 9). The letter value of the label varies based on the polygon type, with “A” for quadrilaterals, “B” for triangles and “C” for pentagons. Labeled points lying on vertices shared by two or more polygons will have multiple labels, separated from each other by a vertical bar (’|’).

CONCLUSION
We have presented a shape grammar plug-in for Grasshopper that supports both 2D and 3D geometries as well as non-parametric and parametric behaviors. Shapes and rules may be defined paramet-
rically even in cases where the rule matching mechanism is non-parametric. The plug-in also allows for compound, geometric and description rules to be defined and applied in tandem on shapes. Despite or thanks to its rule-based system, the plug-in provides room for exploration and unexpected outcomes, especially when using parametric rules. There is certainly more opportunity to investigate additional components that can further support exploration and may facilitate generation and synthesis on a larger scale, e.g., an urban scale.

REFERENCES
Stiny, G. 1980, 'Introduction to shape and shape grammars,' Environment and Planning B: Planning and Design, 7, pp. 343-351
Stouffs, R. 2012 ‘On shape grammars, color grammars and sortal grammars,’ Proceedings of eCAADe 2012, Prague, pp. 479-487
Stouffs, R. 2017 ‘A practical shape grammar for Chinese ice-ray lattice designs,’ 2nd International Workshop on Cultural DNA, Daejeon, South Korea
Stouffs, R. 2018, 'Description grammars: A general notation,' Environment and Planning B: Planning and Design, 45(1), pp. 106-123
Sutherland, I. 1963, Sketchpad: A man-machine graphical communication system, Ph.D. Thesis, MIT
[1] https://www.rhino3d.com/

ACKNOWLEDGMENTS
This work received funding support from Singapore MOE’s AcRF start-up grant, WBS R-295-000-129-133. The authors would like to thank Bui Do Phuong Tung for his development work on the SortalGl library.

Figure 10
Rule application of the rules from Fig. 9., demonstrating both the potential rule applications from the same rule, and generative variations from the same derivation.