WHERE ASSOCIATIVE AND RULE-BASED APPROACHES MEET

A Shape Grammar Plug-in for Grasshopper

RUDI STOUFFS
1 National University of Singapore
1 stouffs@nus.edu.sg

Abstract. We present a shape grammar plug-in for Grasshopper that allows shapes and shape rules to be defined in a parametric manner, even if the rule matching mechanism does not support parametric rules. The plug-in supports shape emergence and provides support for visually enumerating rule applications. We reflect on the interaction between parametric or associative modelling and rule-based generation within the context of using this plug-in.

Keywords. Shape grammar; SGI; parametric modelling; Grasshopper; rule-based.

1. Introduction

Many authors have attempted to define generative design (see Agkathidis (2015) for a synoptical review) and such attempts can be considered from different points of view. From a historical point of view, analytical form finding techniques led to form-finding algorithms and algorithmic design approaches (e.g., Terzidis, 2006), among others. From a popular point of view, generative design may be equated to parametric or associative modelling. While different types of parametric modelling can be distinguished (Janssen and Stouffs, 2015), parametric modelling can generally be considered as a form of both visual programming and dataflow modelling and, as such, as a form of algorithmic design.

Bohnacker et al. (2012) equate algorithms to rules or rule sets, in the context of generative design in the Processing environment. Conceptually this equation makes sense, rules are a convenient way of expressing design moves or actions and their underlying principles in a structured form (Flemming, 1987), especially in the context of design derivation (Woodbury and Burrow, 2006). “Designers preferentially use past successful moves, their own or others, in future projects. One of the long promises of grammars is the ability to explicitly encode such moves” (Woodbury and Burrow, 2006, p.69). However, structurally creating rules and rule sets, and applying these rule sets in design, is far from straightforward.

Both associative and rule-based approaches have advantages and disadvantages and a combination of the two may prove to be more effective. Examples of such hybrid approaches already exist, though generally limited to specific case studies. Most examples of shape grammars in literature do
constitute parametric shape grammars, which may include parametric constraints as associations between different parameters (e.g., Duarte, 2001). However, few implementations exist that provide some flexibility in use. An alternative is embedding rule-based considerations within a parametric modelling environment, e.g., Li et al. (2013) revisit the “rule-based and parametric” system (Li, 2001) described in the Yingzhao Fashi (Li, 1103) into a parametric model that encapsulates some of the rules.

In this paper, we present a shape grammar plug-in for Grasshopper that allows shapes and shape rules to be defined in a parametric manner, even if the rule matching mechanism does not support parametric rules. The shape grammar plug-in supports shape emergence and provides support for visually enumerating rule applications and, to some extent, derivations. We elaborate on the implementation of the shape grammar interpreter and the related plug-in, and present examples of shape computations, shape rule applications and derivations, and their visual enumeration, using the plug-in. The discussion focuses on the interaction between parametric modelling and rule-based generation within the context of this plug-in.

2. Sortal grammars

Grammar formalisms for design come in a large variety (e.g., Stiny, 1980; Stiny, 1981; Carlson et al., 1991; Heisserman and Woodbury, 1994; Duarte and Correia, 2006), requiring different representations of the entities being generated, and different interpretative mechanisms. Shape grammars also come in a variety of forms, even if less broadly. Most examples of shape grammars rely on labeled shapes, a combination of (2D) line segments and labeled points (Stiny 1981). However, even in the original conception of shape grammars (Stiny and Gips, 1972), an iconic shape (made up of curved lines) serves the role of non-terminal marker rather than labeled points, and a colored infill of the resulting shapes is considered part of the generative specification, though not of the shape grammar.

Next to labels, other non-geometric attributes have been considered for shapes. Stiny (1992) proposes numeric weights as attributes to denote line thicknesses or surface tones. Knight (1989; 1993) considers an extension to the shape grammar formalism that allows for a variety of qualitative aspects of design, such as color, to be integrated in the rules of a shape grammar. Though not specific to colors, the resulting grammar is called a color grammar and notions of transparency, opacity and ranking are introduced to regulate the behavior of interacting quality-defined areas or volumes.

Sortal grammars (Stouffs and Krishnamurti, 2001; Stouffs, 2012) take this one step further. Sortal grammars are a class of formalisms for design grammars, utilizing sortal structures as representational structures, where these structures are defined as formal compositions of other, primitive, sortal structures, termed sorts. Sortal grammars 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. In addition, whereas in all other formalisms the augmented shapes have been derived from shapes of
spatial elements by associating symbols, labels or other qualitative aspects to the elements, under a shape-attribute relationship, in sortal grammars, shapes may be either the object or the attribute in the relationship, or both (or neither, though such examples do not constitute spatial grammars as such).

3. Sortal grammar interpreter

A sortal grammar interpreter, denoted SortalGI, has been developed in the form of a library and API in the Python programming language (“Sortal.org”, 2017). Using sortal structures as the representational building blocks has allowed for a modular implementation of a generalized shape grammar interpreter for different grammar forms. The SortalGI library supports both parametric and non-parametric shape grammars, including points, line and plane segments, circular and elliptical arcs, quadratic Bezier curves, labels, weights, colors, enumeratives, and (parametric) descriptions, in 2D and 3D. Emergence is naturally supported. Note that the SortalGI library adopts a graph-based representation for parametric shapes, however, different from other graph-based implementations (Grasl and Economou, 2013; Wortmann, 2013; Strobbe et al., 2015), it does not use any sub-graph matching algorithm but instead relies on a combinatorial enumeration of potential matches. In general, graph-based, parametric subshape recognition is non-polynomial, even with a hypothetical, linear time subshape detection algorithm (Wortmann and Stouffs, in press). In comparison, a combinatorial enumeration, searching for $k$ maximal elements within a set of $n$ (distinguishable) maximal elements, yields a tight bound of $O(nk)$. Depending on the size of $k$, this bound is exponential in the worst case, while one can use labels to limit the combinatorial explosion.

The SortalGI library can be accessed and employed in three different ways: firstly, within a Python development environment; secondly, within the Rhino 3D modeling environment (“Rhinoceros”, 2017), in order to make use of Rhino’s graphical capabilities; thirdly, as a Rhino/Grasshopper plug-in (“Grasshopper”, 2017), requiring no programming or scripting. Though the entire library is available within the Rhino 3D modeling environment, the API provided does limit the extent of geometric and non-geometric element types that are supported, partly due to the need to graphically visualize the data within the Rhino 3D modeling environment, and partly due to active developments. The current Grasshopper plug-in further limits the capabilities to non-parametric line segments, labeled (description) points and descriptions, though support for parametric line segments and (labeled) points is under development.

4. The SortalGI Grasshopper plug-in

The SortalGI Grasshopper plug-in encapsulates the SortalGI library and interpreter and makes part of its functionality available within Grasshopper. Specifically, it supports the specification and application of shape rules, using non-parametric line segments, labeled (description) points and descriptions, and the generation of single or multiple rule application results.
4.1. SHAPES

The plug-in defines both a rule and a shape object. The shape object represents a Grasshopper geometry as a sortal data structure. The necessity to do so stems from the fact that Grasshopper geometries do not maintain text objects very well. The Shape component also filters out any geometric information it does not understand and returns the remaining geometry together with the resulting shape object. Additionally, it accepts a reference point for the shape; specifying a reference point moves the shape from the reference point to the origin. This is especially useful when a shape is specified as the left-hand-side or right-hand-side shape of a shape rule. Both sides to the rule can be drawn side by side, rather than one overlapping the other, while reference points ensure the shape geometries properly relate. A TextPoint component simplifies the specification of a text object representing a labelled point. It takes a point and a (single-line) text and creates a text object located at the point. Alternatively, the user can invoke the text command in Rhino to create a text object. Finally, a variant shape component, denoted dShape, embeds descriptions into a shape object. Descriptions are represented as text; these do not form part of the geometry and are not visualized in Rhino, instead, they can be collected into a Panel component for visualization within Grasshopper. Note that descriptions are necessarily typed and description types are pre-defined via the SGISetup component (see section 4.4 Setup). In summary:

- **Text Point** creates a Rhino text object from a point and a single-line text.
- **Shape** takes a geometry of lines, polylines and/or text objects and returns the resulting shape object together with its geometry. It also accepts a reference point; if specified, the shape is moved from the reference point to the origin.
- **dShape** behaves exactly like Shape but also accepts a list of descriptions. The descriptions are embedded into the shape object and additionally returned.
- **S2G** converts any shape object into a geometry of lines and/or text objects. Any descriptions embedded in the shape object that are not part of labelled points are separately returned as a list.

Given the definition of a shape object, various geometrical operations are redefined as Grasshopper components acting upon shapes; e.g., there is a component to translate/move a shape, rotate a shape, reflect/mirror a shape and scale a shape. Each of these components takes as input a shape object and any additional data required to inform and apply the transformation, and returns the resulting shape object as well as the corresponding geometry and descriptions. Specifically:

- **Move Shape** takes a shape and a vector and moves the shape along the vector (see section 4.4 Setup for a particular use of the Move Shape component).
- **Rotate Shape** takes a shape, a base plane and a rotation angle and rotates the shape about the normal vector of the base plane by the specified angle.
- **Mirror Shape** takes a shape and a mirror plane and mirrors the shape about the base plane.
- **Scale Shape** takes a shape and a scaling factor and scales the shape accordingly. Note that the scaling occurs relative to the origin.

In addition, there are components to union/sum two shapes, intersect/take the product of two shapes, and take the difference of one shape with respect to another.
Each of these components takes two shape objects as input and returns the resulting shape object, as well as the corresponding geometry and descriptions.

- **Sum** combines two shape objects into one.
- **Product** takes two shape objects and returns the common part.
- **Difference** takes two shape objects and returns the complement of the first shape with respect to the second shape.

### 4.2. RULES

A **rule** object may constitute either a shape rule or a compound rule combining a shape rule with one or more description rules. The **Rule** component takes a rule name, a rule description, that is, a brief explanation of the rule, a left-hand-side shape (lhs) object and a right-hand-side (rhs) shape object. If a description type is present as part of one shape object (lhs or rhs) but absent from the other shape object, an empty description of that type is automatically added to the other shape object within the rule.

- **Rule** defines a shape rule from a left-hand-side shape object and a right-hand-side shape object, as well as a rule name and a rule description.
- **Rule Info** deconstructs a rule object into its left-hand-side shape object, right-hand-side shape object, and a multi-line text containing its GUID, name and description.

### 4.3. RULE APPLICATIONS

There are two rule application components, one yields a single rule application outcome, the other yields a list of all rule application outcomes. Both components take as input a rule object, a shape object, and an optional sub-shape object. The sub-shape, if specified, must be a part of the shape; then, the matching is performed on the sub-shape, while the rule application is performed on the shape. In addition, the **Apply** component takes an optional rule application index. Next to the resulting shape object and the corresponding geometry and descriptions, **Apply** returns True or False indicating success or failure, and a translation vector. The translation vector allows the resulting shape to be visualized aside from the original shape (see section 4.4 Setup). The **Apply All** component returns a list of resulting shape objects and the corresponding list of geometries and list of (lists of) descriptions, as well as True or False indicating success or failure, and a list of translation vectors. The translation vectors allow the resulting shapes to be visualized one above the other and aside from the original shape (see section 4.4 Setup). For both components, if no rule application was determined (failure), then the original shape is returned as the result. In summary:

- **Apply** applies a rule onto a shape. It accepts an optional sub-shape object to limit the potential matches, and an optional rule application index. An index value of -1 (default) selects a random rule application, any number outside the index range yields the last one among the rule applications.
- **Apply All** applies a rule onto a shape, returning all rule applications. It, too, accepts an optional sub-shape object to limit the potential matches.
4.4. SETUP

Finally, the SGI Setup component initializes the sortal grammar interpreter and, therefore, must always be added before any other components of the SortalGI plug-in. In addition, it allows for a few global settings:

- SGI Setup initializes the sortal grammar interpreter and takes the following inputs: an X displacement value, a Y displacement (see below), a text size, and a list of description types, each identified by its name. The text size specifies the size of the labels in the visualization of the shape(s) resulting from any SortalGI component.

The X displacement value is used by both the Apply and Apply All components to create the translation vector(s). If specified, the X value of any translation vector equals the given value, such that, using a MoveShape component, the resulting shape can be visualized aside from the original shape, at the given distance along the X axis. The Y displacement value is used only by the Apply All component to create the translation vectors. When multiple shapes result from an Apply All node, the respective translation vectors will automatically ensure an appropriate spacing of the resulting shapes, such that, using a MoveShape component, these can be visualized one above the other. If a Y displacement value is specified, this value will inform the Y value of any translation vector. If no X or Y displacement value is specified to the SGI Setup component, then the respective value of the translation vector will be informed by the bounding box of the original shape.

5. Applications and discussion

The value of this shape grammar interpreter plug-in stems from its ability to support shape computations, shape rule applications and derivations, and their visual enumeration, within an interaction between parametric or associative modelling and rule-based generation. Here, we present some applications that demonstrate these abilities and serve to discuss future work and developments.

Figure 1 demonstrates a short derivation emphasizing emergence in shape computation (Knight, 2003, p. 128). The derivation adopts two rules, one to move a square shape diagonally up and to the right, the other to move an L-shape diagonally down and to the left. Note that due to rotational symmetry, the square can be moved in any diagonal direction, while the L-shape can only be moved in the direction pointed to by the ‘L’. The derivation consists of five consecutive Apply nodes, separated with MoveShape nodes in order to ensure a visual sequence of shapes.

Figure 2 demonstrates an enumeration of potential rule applications (using Apply All nodes) for the second and third rule application steps from Figure 1. There are twenty potential rule applications for the second step, moving each of the five squares composing a cross-like shape in four diagonal directions. There are another twenty potential rule applications for the second step, moving an L-shape diagonally. Eight shapes resulting from the second step have two L-shapes; four additional shapes have exactly one L-shape. MoveShape nodes are used to ensure a proper visual layout.
Figure 3 illustrates the potential interaction between the parametric environment offered by Grasshopper and the (non-parametric) grammar interpreter made available through the SortalGI plug-in. The derivation is exactly the same as in Figure 1, however, the input shapes to both rules have been parametrically altered from a square and an L-shape the size of three squares into a rectangle and an L-shape similarly made up of rectangles with the same ratio. Introducing a slider allows one to parametrically alter the ratio of the rectangle, thereby altering the shape(s) resulting from the derivation, without altering the derivation itself. Note that when the ratio of the rectangle is altered to 1.5:1, the derivation breaks down as the cross-shape resulting from the first rule application step can now also be interpreted as composed of two intersecting rectangles. This
increases the number of matches, such that the selected match no longer yields the expected result.

Figure 2. Grasshopper model (left) demonstrating an enumeration (right) of potential rule applications for the second and third rule application steps from Figure 1. Note that the warning in the model relates to the fact that not all results from the second derivation step yield potential applications in the third step.

Figure 3. The same short derivation as in Figure 1, however, the square shape has been replaced with a rectangular shape; the derivation has otherwise not been altered.

The examples above have omitted any use of labels or descriptions, as they were not necessary to demonstrate the potential of adopting a rule-based approach
within a parametric modelling environment. Both labels and descriptions mainly serve to constrain rule application, a feature that becomes more important in larger grammars (e.g., where rules are applied in stages and each stage adopts its own labels to constrain rules to apply only during this stage) or in parametric shape grammars (where the number of matches can be potentially much larger). While the plug-in supports any number of rules, managing a large number of rules within a dataflow modelling environment will result in an overly complex associative model. Providing a rule application component that accepts a rule name rather than a rule object (or a component that returns a rule object given a rule name) may mitigate this problem to some extent. However, we acknowledge that the adoption of a rule-based mechanism within a parametric modelling environment lends itself better to smaller applications or explorations. We believe that extending the kinds of explorations that can be achieved, e.g., by supporting various types of curves or allowing for parametric shape rules, is a more compelling endeavor. Currently, we are developing support for parametric shape rules. While a shape grammar is either distinguished as parametric or non-parametric, within the context of this plug-in, such distinction seems less relevant and the ability to combine parametric and non-parametric shape rules within a same derivation may be particularly enticing.

It is worth noting that the **Apply** and **Apply All** components only allow for sequential rule applications. In the case of additive rules, **Sum** may serve to combine multiple, alternative rule applications into a single shape as if resulting from a parallel application of the same or different rules. However, in general, parallel rule application would increase the applicability of a rule-based generative approach within a parametric or associative modelling environment. Parametric models are often used to “populate” a surface using a parametric component. Applying a parallel rule application component in combination with a parametric rule matching mechanism, would allow for such behavior to be encoded in rules. Additionally allowing for a parallel rule application component to accept multiple, alternative rules, for example with rule selection guided by one or more shape component attributes, would further increase the applicability of the shape grammar plug-in. This is a topic for future work.

### 6. Conclusion

We have presented a shape grammar plug-in for Grasshopper that allows shapes and shape rules to be defined in a parametric manner, combining the strengths of a parametric or associative modelling approach with rule-based generation. The shape grammar plug-in supports shape emergence and provides support for visually enumerating rule applications and, to some extent, derivations.

### Acknowledgements

This work received funding support from Singapore MOE’s AcRF start-up grant, WBS R-295-000-129-133. I would like to thank Bui Do Phuong Tung and Bianchi Dy for their development work on the SortalGI library and plug-in, respectively.
References


Li, A.: 2001, A shape grammar for teaching the architectural style of the Yingzao fashi, Ph.D. Thesis, MIT.


Wortmann, T.: 2013, Representing shapes as graphs, Master’s Thesis, MIT.

Wortmann, T. and Stouffs, R.: in press, Algorithmic complexity of shape grammar implementation, AI EDAM.