Towards a useful grammar implementation: beginning to learn what designers want

Andrew I-kang Li[a],[1] Rudi Stouffs[b]

[a] Kyoto Institute of Technology, Kyoto, Japan

[b] National University of Singapore, Singapore

We present a general shape grammar implementation that supports subshape detection and handles lines and labeled points in three-dimensional space. Its front and back ends are both set in the CAD application Rhinoceros3d. Informal observations of designers using the implementation suggest that they are more interested in producing designs to work with than in using the more specialized features of shape grammars. This in turn suggests that researchers who create such implementations have more to learn about such usages designers use them.

1 Introduction

Since the very first shape grammar implementation [4], the fundamental technical challenge has always been subshape detection [6], the recognition of one shape as being part of another shape, under some allowable transformation. Stouffs and Krishnamurti [13] identified two additional challenges. The first is generality – the rapid development, adaptation, and maintenance of grammar-based systems – or the ability of users to create and run their own grammars.

The second challenge concerns ways of enabling designers to employ grammatical rules in a manner that does not impede their designing. Tapia [15] partly addressed this challenge by paying attention to the user experience, in particular visual interaction. In his implementation, users created shapes, not by typing coordinates, but by drawing the shapes directly in a drawing program, that is, as shapes. In shape grammars, shapes and their component lines and points are domain objects, and now users could manipulate them directly. To apply rules and generate new shapes, they used a standalone interpreter, known as GEdit.

[1] Corresponding author. Tel.: +81 75 724 7657. E-mail address: andrewli@kit.ac.jp.

The first author, Li [8], continued this approach by increasing the types of domain objects that users could manipulate directly: not only lines and shapes, the components of shapes, but also shapes and rules, the components of grammars. He used a commercial modeling application, first AutoCad, then Rhinoceros3d (commonly known as Rhino), in which users could draw shapes and rules and handle newly generated shapes. The difference from GEdit was that the whole grammar – the initial shape and the rules – was a Rhino model and could be saved as a Rhino document. Li called this a *whole-grammar* approach, since the whole grammar was available for thinking through direct manipulation.

This provided two advantages. First, users could now directly manipulate all domain objects: rules, shapes, lines, and labeled points. Second, they could now organize rules and shapes in groups to help their thinking, just as they organize icons on their computer desktops. This visual organization is known as secondary notation.

In addition, Rhino offers a wide range of tools, with which users could pre- and postprocess grammatically created shapes. They could, for example, create complicated shapes with scripts, modify those shapes with grammars, and convert those shapes to solid models for 3d printing. Users could now use grammars in a larger work stream of designing and making.

Li's arrangement had a significant disadvantage, however. Like Tapia's, it consisted of two separate applications: one for creating and manipulating shapes, the other for applying rules and calculating new shapes. Where Tapia had GEdit, Li had Grammar Environment, an application developed by Chau et al. [1].

This split arrangement meant that when users switched tasks, they had to switch applications. And, when switching applications, they also had to transfer shape and rule files. Li provided a library, written in the Python programming language, to facilitate file transfer and similar tasks, but this non-domain work clearly impeded users' ability to focus on domain objects and operations.

We now address this issue by replacing the stand-alone interpreter with one that runs inside Rhino. This new back end is SortalGI, the *sortal* grammar interpreter [12]. It supports subshape detection and is written in the Python programming language, as are the front end scripts, which have been adapted from the previous version. In this integrated version, all scripts run in Rhino (both v5 and v6 for Windows, and v6 for Mac), and users can do their grammatical work entirely within the Rhino environment.

2 The front end

Since the front end is set inside the Rhino environment, it is only one of many functionalities already available there. Its purpose is to enable users to use the back end easily. One of the ways it does so is by rendering the components of the grammar – those Rhino objects drawn by the user – parsable for communication to the back end. And for this, the front end needs to use some of Rhino's organizational capabilities, such as layers. At the same time, we want to keep those very capabilities available to users to the largest extent possible. In anticipation of this tension between structure and freedom, we followed a few working guidelines.

First, shapes are composed of Rhino objects, currently line curves and text dots. That is, the objects themselves are the shape; there is not some symbolic representation between the shape and the user. They are in the foreground, available for manipulation by the user, and persist between work sessions as the record of the grammar.

Second, the Rhino work space is divided between the system and the user. The system needs a place to do its work, and so do users. One way we try to accommodate both system and users is by having the system display rules and all calculated shapes in the positive-*y* half of the three-dimensional virtual work space. The other half is left for users to use as they will. The other way is by assigning rules and calculated shapes to their own, automatically named layers. This way, the front end can identify rules and calculated shapes, and users can also create and name layers for their own use.

Third, commands are implemented as Python scripts, as in the previous, split version. Users invoke a command by running a script. This is an interim measure; in future versions commands will be available through more straightforward means, like menu items.

These working guidelines are intended to make users' experiences easier and more productive. The first step for users is to initialize the Rhino document by running the *initialize* script. This prepares both the back end, by initializing its internal representation and rule register, and the front end, by creating a new layer, *Shape 0,* for the initial shape. Shapes on layers named *Shape n* are automatically scrolled so that results are shown chronologically along the positive *y*-axis with the latest result always displayed just above the *xz*-plane.

If the document already contains a grammar, the front end reads the rules into the rule register, and users can resume using the grammar immediately. If the document is new, then it is time for users to create at least an initial shape and a rule. To create the initial shape, users simply draw it with line curves and text dots in the positive-*x,* positive-*y* quadrant of the space (the upper right quadrant of the *xy*-plane in two dimensions) on the layer *Shape 0.*

To create the rule, users draw below the *xz*-plane on any user-named layer. Then they run the *create rule* script, which prompts them to select: 1) the elements of the left shape; 2) the reference point of the left shape; 3) the elements of the right shape (if any); and 4) the reference point of the right shape (if not empty).

The system then draws the rule in the negative-*x,* positive-*y* quadrant on a new layer *Rule 1*. Around each of the left and right shapes is a three-dimensional frame which indicates the shape's coordinate system (Fig. 1). The elements of the two shapes and both frames are locked in a single group for easy selection for rule application. If users draw a second rule, previously created rules will be scrolled away from the *xz*-plane to make space to draw the new rule. Each new rule is entered into the rule register.

Now users have at least one shape and one rule. They run the *apply rule* script, which prompts them to select: 1) the shape; 2) the subshape (optional); and 3) the rule. By selecting a subshape, users can, for example, apply a rule to a single cell in a matrix.

The system calculates all next shapes and, if any, draws them in a row in the positive-*x,* positive-*y* quadrant, each shape on its own layer, and scrolls them – along with all earlier generations of calculated shapes – away from the *xz*-plane (Fig. 2).

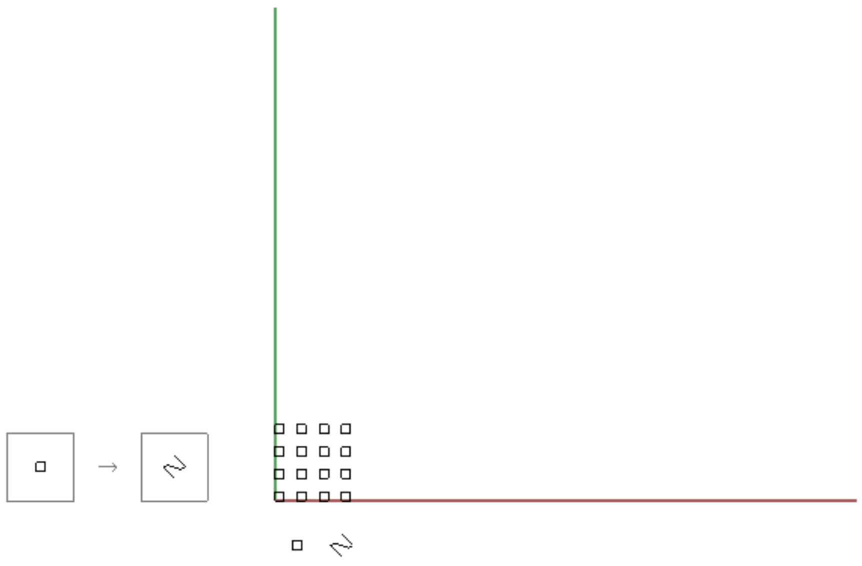

Fig. 1. Two-dimensional (top) view of the work space, containing a grammar before the first rule application. The rule (with frames around the two shapes) is in the upper left quadrant, while the initial shape (the 16 squares) is in the upper right quadrant. The two shapes in the lower right quadrant are the shapes drawn by the user to define the rule.

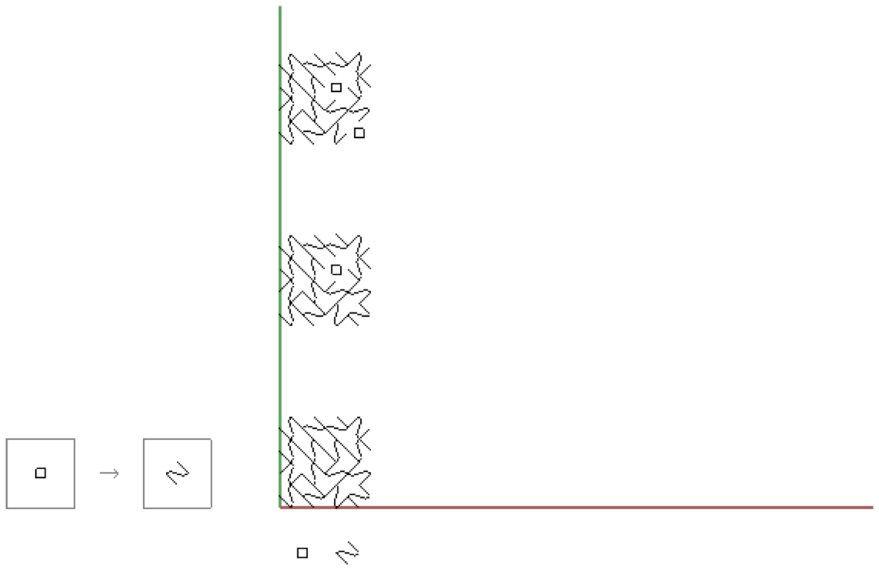

Fig. 2. Two-dimensional (top) view of the work space after 16 rule applications. Only the last 3 generations are shown here; the first 13 generations are scrolled away from the *x*-axis.

3 The back end

The back end consists of the SortalGI shape grammar interpreter library and API, developed in the Python programming language. Beyond lines and labeled points (text dots in Rhino), SortalGI actually supports a variety of different shape grammar forms, including representations and matching algorithms. SortalGI presents a modular implementation of a generalized shape grammar interpreter, utilizing *sortal* structures as representational building blocks. *Sortal* (representational) structures are hierarchically defined as formal compositions of other, primitive, data structures, denoted *sorts* [11]. The main compositional operators are a co-ordinate, disjunctive composition of any number of *sortal* structures, as in a composition of lines and points , and a subordinate, semi-conjunctive composition of a primitive *sort* with any other *sortal* structure under an object–attribute relationship, as in the case of labeled points – points with labels as attributes. Where the fundamental technical challenge is subshape detection, *sortal* grammar formalisms [9,13] benefit from the fact that every composite structure derives its matching mechanism for subshape detection from the respective matching mechanisms of the component structures. This means that, ultimately, the matching mechanism of any composite *sortal* structure derives from its primitive component *sorts*, solely depending on the respective compositional operators. Having implemented the matching mechanisms for each of the primitive *sorts*, any composition thereof has its matching mechanism implemented as well. In this way, the SortalGI interpreter allows for a broad range of shape grammar formalisms to be supported, including many formalisms found in shape grammar literature.

Beyond its representational flexibility, the SortalGI library includes two alternative matching mechanisms for spatial elements: a non-parametric mechanism matching shapes under similarity transformations (translation, rotation, reflection and uniform scaling) and a parametric-associative mechanism matching shapes under some topological constraints as well as associations of perpendicularity and parallelism. The former recognizes shapes based on similarity, a square matches any square, irrespective of its location, orientation or size. Similar for rectangles of a fixed length-to-width ratio. The latter extends the matching mechanism to polygons of a specific arity. A convex quadrilateral matches any other convex quadrilateral polygon, irrespective of its exact shape, on condition that the former have no perpendicular or parallel edges.

Any such perpendicular or parallel edges in the shape to be matched would need to be matched in the target shape. Note that while the SortalGI library adopts a graph-based representation for parametric-associative shapes, unlike other graph-based implementations [5,14,17], 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 nonpolynomial, even with a hypothetical, linear time subshape detection algorithm [18]. In comparison, a combinatorial enumeration, searching for k elements within a set of n (distinguishable) 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 attribute labels to limit the combinatorial explosion.

As such, the SortalGI library supports both parametric-associative and non-parametric shape grammars, including points, line and plane segments, circular and elliptical arcs, quadratic Bezier curves, labels, weights, colors, enumerative values, and (parametric) descriptions, in 2D and 3D. Emergence is naturally supported. Currently, the front end utilizes only the non-parametric mechanism, applied to lines and labeled points. However, it can be extended to include other domain objects as well as the parametric-associative matching mechanism. TNevertheless, even if thehough the entireSortalGI library is available in its entirety within the Rhino modeling environment, the API provided does limit the extent of geometric and non-geometric element types that are supported, due to the need to graphically visualize the data within the Rhino modeling environment.This API has been specifically developed to support the integration of the SortalGI library within Rhino. In particular, it not only acts as a programming interface providing access to the underlying functionality, but also supports the conversion of geometric data from Rhino into the SortalGI interpreter and back. At first, the conversion was done from an agnostic description instead of from Rhino's internal object representation. The agnostic description was conceived as a text-based data structure, not unlike the one that was used with Chau et al.'s [1] engine. The current version of the API no longer accepts the agnostic description and instead relies on Rhino GUIDs referencing Rhino objects. This is possible because the engine runs within Rhino and can query Rhino objects directly using Rhino's Python API.

The API mainly offers functions to create shapes from lists of Rhino GUIDs, to create rules, to determine rule applications and to draw the resulting shapes. The drawing process is an integral part of the conversion from the *sortal* representation back to Rhino GUIDs, in order to generate these GUIDs. However, in order to allow the front end to decide when and where which results should be visualized, the visibility of the resulting Rhino objects is turned off by default.

The library maintains a rule register and every rule is automatically added to the register. For this reason, rule names must be unique. Rules can be retrieved from the register by name. The library also maintains a shape register although that is entirely voluntary and serves little use in the context here described. Additionally, the library also maintains a register of rule applications both to improve rule application performance, when the same rule is applied to the same shape more than once, and to allow for the conception of a derivation tree. There remain many questions as to how such a derivation tree may be accessed and visualized within a platform such as Rhino and, as such, this functionality is currently underdeveloped and unused.

One difficulty encountered has been the precision of geometric information within Rhino. To adjust the precision of computation within the library to the requirements of Rhino, a precision parameter has been created within the SortalGI library that can be set and adjusted through the API.

4 Discussion

Here we discuss our experiences in workshops and a graduate-level course[2] for design students, most of whom had had no previous experience with shape grammars.

[2]CAAD Futures workshop, June 2019, Daejeon, South Korea; Ouroborous workshop, August 2019, National Yunlin University of Science and Technology, Yunlin, Taiwan; eCAADe + SIGraDi workshop, September 2019, Porto, Portugal; Advanced computational design, Fall 2018 and Fall 2019, Kyoto Institute of Technology, Kyoto, Japan.

4.1 The user experience

As for the front end, weWe recall that the proximate motivation for adopting an internal back end was to eliminate the mental friction of repeatedly shifting focus and files between applications. The idea was straightforward, but the effect was as consequential as we had anticipated: users simply spent more time on design tasks. As a result, it was easy to observe that users were interested in grammatically generated shapes as objects for their design work, but not as elements in a structured design space. Users accumulated shapes, periodically evaluated them as individuals on their own account, and then culled them without, for instance, trying to preserve derivational sequences.

And if users were not much concerned by the technical niceties of shape grammars, neither were they willing to forsake other design tools for shape grammars. For example, instead of drawing complicated shapes by hand, they wrote Python scripts to create them. They then applied grammar rules to these shapes. In other cases, they used grammars to create what were essentially design diagrams, consisting of zero-width lines. They exported the diagrams to Illustrator, where they assigned (non-zero) widths to the lines and extracted the edges of those lines as cutting paths.

These observations are entirely consistent with Woodbury's [16] assertion that designers are pragmatic, use whatever tools will help them in their work, and tend to use those tools with less technical sophistication than expert users. This suggests that there is a slight conflict between the authors' interest in domain objects and operations on the one hand and designers' interest in getting the job done on the other. Or perhaps we should say that domain objects and operations are not the same for designers as for the authors, the specialists who created the tool.

For example, we noticed that there were situations in which users were applying the same rule repeatedly to the same shape, when filling in cells in a matrix, for example. This was inefficient, not only because the user had to run the script for each reapplication, but also because the system was redoing many of the same calculations. We wrote a script for applying a rule repeatedly, and the users received it enthusiastically. It was slow, but it freed users from what was to them unnecessary work. The back end offers what are known as *flows,* which are practically composite shape rules and provide a generalized capability for rule application. Users showed us the importance of this capability, and we expect to develop it further.

4.2 The grammarian's perspective

Similar to the development of the front end interface, the SortalGI library is conceived and developed with shape grammars in mind, rather than simply as a rule engine supporting search and replace. As such, spatial elements are distinguished as either 2D or 3D and as adhering to either a non-parametric or a parametric-associative matching mechanism. Even though it supports multiple grammar forms, the library provides little support for switching between grammar forms. This may not be much of an issue for now, as the interface acts only upon 3D shapes and utilizes only the non-parametric matching mechanism, for now. However, future developments may see the utilization of the parametric-associative matching mechanism as well, possibly requiring the exchange of data between different forms and representations.

Most shape grammars in literature also consider a single formalism to apply for the entire grammar. Even if multiple formalisms are conceived to be used simultaneously, they are generally used in parallel [7], and any exchange of information between these parallel representations relies on description rules to include references to other descriptions or shapes. Such exchange is fully supported in SortalGI [3,10]. Duarte [2] conceives of a discursive grammar that combines two grammars sequentially, a programming grammar generating design briefs based on user and site data and a designing grammar using the design brief(s) to generate designs in a particular style. However, the programming grammar is only a description grammar and any exchange of information between the two grammars is accomplished through description rules operating on the descriptions resulting from the programming grammar.

Acknowledging that a design that originally was conceived in plan or section, or both, might be further elaborated in three dimensions, or the designer might want to use both non-parametric and parametric-associative shape rules intermittently, the SortalGI API does attempt to address these limitations. In particular, it provides support to exchange data between two- and three-dimensional representations, assuming these otherwise correspond to a large degree. Similarly, it supports the exchange of data between representations adhering to the non-parametric and parametric-associative matching mechanisms, again assuming some correspondence between both representations.

4.3 Explorations of extended functionality

Beyond its use within the Rhino modeling environment, as presented in this paper, the SortalGI library can be accessed and employed in (at least) two more ways: firstly, within a Python development environment; secondly, as a Rhino/

Grasshopper plug-in [3], requiring no programming or scripting. These offer access to additional functionality and attempt to address some of the restrictions of the visual front end presented in this paper. For example, the Grasshopper plug-in supports both non-parametric and parametric-associative shape rules to be defined, and for these rules to be applied intermittently. However, it should be noted that the underlying data exchange mechanisms are not entirely general, as it is difficult to establish a general data exchange mechanism to apply between any two *sortal* representations when these representations are unknown in advance and, in the extreme, may not have much in common. We expect the future use of SortalGI to provide more insight to which extent such exchange of data is actually important and desired.

Another matter for further consideration is the relation between the front end and the SortalGI Grasshopper plug-in, which may be considered as two competing interfaces to the SortalGI interpreter, but might also serve as complementary ways of accessing its functionality. Specifically, the integration of the front end with the plug-in could support users in generating complex shapes and rules, while at the same time providing more flexibility in applying rules. Currently, the front end and the plug-in each embody their own initialization set-up which are not entirely compatible. Instead, adopting a common *sortal* representation and ensuring a common initialization may allow users to switch back and forth. Besides this integration, user experiences from either or both interfaces may demonstrate best practices and influence their individual development.

In terms of rule application, the plug-in provides four different components that to some extent reflect ways of adopting the front end as outlined before. The first one applies only a single match, which may be selected by index or, otherwise, randomly.

The second one is to conform to the behavior of the front end. It applies all matches in parallel, returning as many results as there are matches and allowing these all to be visualized separately.

The third one may serve to apply a single shape rule repeatedly to fill cells in a matrix. Applying all matches one after another, it ignores the possibility that the actual result from one application may no longer be able to serve as the input for the next application to match. Note that only in the case of an addition rule does this amount to the exact same behavior as repeating the rule application truly sequentially. In the latter case, if any part of the shape is removed as a result of rule application, then this removal will impact subsequent rule applications, but. This is not the behavior of the third rule application node. Instead, visually, it behaves as if all matches are applied in parallel, combining all the results into a single shape outcome.

Finally, the fourth one can be used to apply a rule repeatedly, although it must be specified how many times. It takes a series of rules as input and applies each rule in sequence, returning all intermediate results as well as the final result. In addition, the plug-in offers a component that does not actually apply the rule, but, instead, returns all the matches. As such, it supports a form of search and extraction based on the left side of the rule. These may then serve as input to one or more rule applications and, as such, improve the efficiency through the application of a divide-and-conquer technique.

5 Conclusion

We presented a general shape grammar implementation that supports subshape detection and handles lines and labeled points in three-dimensional space. Its front and back ends are both set in the CAD application Rhinoceros3d. Informal observations of designers using the implementation suggest that they are more interested in producing designs to work with than in using the more specialized features of shape grammars. Clearly we need to understand better how designers use grammatical tools. Even though they may not aspire to be grammatical specialists like the present authors, they have much to teach us about how to make grammars more useful to them.

References

[1] Chau, H.H., Chen, X.J., McKay, A., and de Pennington, A.: 2004. Evaluation of a 3D shape grammar implementation, *Design computing and cognition '04* (ed. John S. Gero), 357–376, Dordrecht, Kluwer.

[2] Duarte, J.P.: 2001, Customizing mass housing: a discursive grammar for Siza's Malagueira houses, Ph.D. thesis, MIT.

[3] Dy, B. and Stouffs, R.: 2018, Combining geometries and descriptions: a shape grammar plug-in for Grasshopper, *Proceedings of eCAADe 2018,* Volume 2, Lodz, Poland, 499–508.

[4] Gips, J.: 1975, Shape grammars and their uses: artificial perception, shape generation and computer aesthetics, Basel, Birkhäuser.

[5] Grasl, T. and Economou, A.: 2013, From topologies to shapes: parametric shape grammars implemented by graphs, *Environment and planning B: planning and design*, 40(5), 905–922.

[6] Krishnamurti, R. and Stouffs, R.: 1993, Spatial grammars: motivation, comparison and new results. *CAAD futures '93* (eds. U. Flemming and S. Van Wyk), North-Holland, Amsterdam, 57–74.

[7] Li, A.I.: 2001, A shape grammar for teaching the architectural style of the *Yingzao fashi,* Ph.D. thesis, MIT.

[8] Li, A.I.: 2018, A whole grammar implementation of shape grammars for designers, *AI EDAM,* 32(2), 200–207.

[9] Stouffs, R.: 2012, On shape grammars, color grammars and sortal grammars, *Proceedings of eCAADe 2012,* Volume 1, Prague, 479–487.

[10] Stouffs, R.: 2018a, Description grammars: precedents revisited, *Environment and planning B urban analytics and city science*, 45(1) 124–144.

[11] Stouffs, R.: 2018b, Implementation issues of parallel shape grammars, *AI EDAM*, 32, 162–176.

[12] Stouffs, R.: 2018c, http://www.sortal.org/downloads/python.html (last accessed 7 March 2019).

[13] Stouffs, R. and Krishnamurti, R.: 2001, Sortal grammars as a framework for exploring grammar formalisms, *Mathematics and design 2001*, Geelong, Australia, 261–269.

[14] Strobbe, T., Pauwels, P., Verstraeten, R., De Meyer, R. and Van Campenhout, J.: 2015, Toward a visual approach in the exploration of shape grammars, *AI EDAM*, 29(4), 503–521.

[15] Tapia, M.: 1999, A visual implementation of a shape grammar system, *Environment and planning B: planning and design,* 26(1), 59–73.

[16] Woodbury, R.: 2010, *Elements of parametric design,* London, Routledge.

[17] Wortmann, T.: 2013, Representing shapes as graphs, master's thesis, MIT.

[18] Wortmann, T. and Stouffs, R.: 2018, Algorithmic complexity of shape grammar implementation, *AI EDAM*, 32, 138–146.