COMPUTATION-FRIENDLY SHAPE GRAMMARS

Detailed by a sub-framework over parametric 2D rectangular shapes

KUI YUE, RAMESH KRISHNAMURTI, FRANÇOIS GROBLER*
Carnegie Mellon University, USA
**Construction Engineering Research Laboratory, Champaign, IL, USA*

ABSTRACT: NP-hardness of parametric subshape recognition for an arbitrary number of open terms is proven. Guided by this understanding of the complexity of subshape recognition, a framework for computation-friendly parametric shape grammar interpreters is proposed, which is further detailed by a sub-framework over parametric two-dimensional rectangular shapes. As both the proof of NP-hardness and rectangular sub-framework invoke elements in graph theory, the relationship between shape and graph grammars is also explored.

KEYWORDS: Parametric subshape recognition, NP-hard, graph grammar

RÉSUMÉ : *Il est démontré que la reconnaissance de sous-formes paramétriques pour un nombre arbitraire de termes ouverts est NP-difficile. En se basant sur cette analyse, nous proposons un cadre d'interpréteurs grammaticaux simples à calculer pour les formes paramétriques que nous détaillons davantage dans le cas de formes paramétriques rectangulaires bidimensionelles. Comme la preuve de NP-difficulté et notre sous-cas nécessitent des éléments de la théorie des graphes, nous explorons également la relation entre grammaires de formes et grammaires de graphes.*

MOTS-CLÉS : *Reconnaissance de sous-formes paramétriques, NP-difficulté, grammaires de graphes*

1. INTRODUCTION

As a formalism, shape grammars (Stiny 1980) have been widely applied, in many different fields, to analyzing designs. Computer implementation of a shape grammar interpreter is vital to both research and application. However, implementing such an interpreter is hard, especially when directed at parametric shapes defined by open terms (Chau *et al.* 2004), though there have been notable attempts (e.g., McCormack and Cagan 2002). The central difficulty is parametric subshape recognition. In this respect, finding a proof on how formally hard this can be provides critical theoretical guidance when developing new algorithms where their generality and practicability can be justified.

This paper is set out as follows. We start with a review of the formal definition of a shape grammar, which is then used to analyze the complexity of subshape recognition for parametric two-dimensional rectilinear shapes. It turns out that parametric subshape recognition is computationally expensive even for shapes of a moderate size; indeed, parametric subshape recognition for an arbitrary number of open terms is NP-hard. Guided by this theoretical result, we propose a framework to ensure computability of shape grammars. This is further detailed by examining a sub-framework over parametric two-dimensional rectangular shapes. Both the proof of NP-hardness and rectangular sub-framework invoke elements in graph theory—in fact, there is a connection to graph grammar research, namely, collage grammars (Drewes and Kreowski 1999). As a consequence, the relationship between shape and graph grammars is explored.

2. THE COMPLEXITY OF A PARAMETRIC INTERPRETER

Various shapes have been investigated in shape grammar research. For convenience and accuracy of discussion, we define shapes to belong to one of the eight types identified in Table 1. For example, non-parametric two-dimensional rectilinear shapes are of *Type I*.

TABLE 1. *A CLASSIFICATION OF SHAPES.*

	RECTILINEAR		WITH CURVES	
	2D	3D	2D	3D
Non-parametric	I	II	III	IV
Parametric	V	VI	VII	VIII

A rigorous definition is essential to a theoretical analysis. Here, we adopt Stiny's (1980) formal definition for two-dimensional rectilinear shapes (Types I and V). It is interesting to note that in Stiny (2006), he gives up this definition, because of the concern that in (architectural) design, there is no preexisting fixed vocabulary, as new vocabularies emerge during the course of design. However, from the perspective of any implementation, the vocabulary has to be 'pre-fixed'; that is, the universe of the vocabulary is known *a priori*. Therefore, with respect to implementation, a formal definition remains valid.

2.1. Formal definition of shape grammars

A *shape* is a limited arrangement of straight lines defined in a Cartesian coordinate system with real axes and an associated Euclidean metric. A shape is specified by the *maximal* line representation. A shape is a *subshape* (part) of another shape whenever every line of the first shape is also a line of the second shape. A *labeled shape* consists of two parts: a shape and a set of labeled points. A *parameterized shape* is obtained by allowing the coordinates of the end points of the maximal lines in a given shape to be variables. A *parameterized labeled shape* σ is given by $\sigma = <s,P>$, where s is a parameterized shape, and P is a finite set of labeled parameterized points. A *labeled parameterized point* is a labeled point p where the coordinates of p are variables.

A *shape grammar* has four components: (i) S is a finite set of shapes; (ii) L is a finite set of symbols; (iii) R is a finite set of *shape rules* of the form $a \rightarrow b$, where a is a labeled shape in $(S,L)^+$, and b is a labeled shape in $(S,L)^*$; and (iv) I is a labeled shape in $(S,L)^+$ called the *initial shape*.

In *non-parametric shape grammars*, a shape rule $a \rightarrow b$ applies to a labeled shape c when there is a transformation τ such that $\tau(a)$ is a subshape of c. The labeled shape produced by applying the shape rule $a \rightarrow b$ to the labeled shape c under the transformation τ is given by $[c - \tau(a)] + \tau(b)]$.

Parametric shape grammars are extensions of non-parametric shape grammars in which shape rules are defined by filling in the open terms (point variables) of a general schema. A *shape rule schema* $a \rightarrow b$ comprise parameterized labeled shapes, a and b, where no member of the family of labeled shapes specified by a is the empty labeled shape. Whenever specific values are given to all variables of a and b, by an assignment g to determine specific labeled shapes, a new shape rule $g(a) \rightarrow g(b)$ is defined. This shape rule can then be used to change a given labeled shape into a new one in the usual way. That is, the shape rule application is expressed as: $[c - \tau(g(a)] + \tau(g(b)]$.

To implement an interpreter for parametric shape grammars, the crucial step is subshape recognition. That is, detecting the existence of subshape $g(a)$ in c by automatically finding an appropriate assignment for g.

2.2. Preliminary analysis

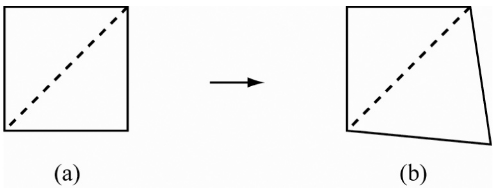
In non-parametric subshape recognition of two-dimensional rectilinear shapes (Type I), the transformation can be determined by matching three distinguishable points of a to three distinguishable points of c (Krishnamurti 1981).

However, in parametric subshape recognition, this is not necessarily the case.

It is possible that the parametric labeled shape, *a,* has a certain number of fixed points (non-open terms). If there are more than three fixed points (distinguishable, by definition), the Krishnamurti 3-point algorithm is still applicable, with about $\binom{n}{3}$ possibilities to test against (new auxiliary points may be computed and used).

For shapes with 1 or 2 fixed points, it is identical to the situation when all points are open as similarity is subsumed by the assignment. When all points are open, shape transformation may not be describable by a matrix. For example, Figure 1a matches Figure 1b under a parametric shape rule, although there is no 3×3 matrix that describes the transformation. As a result, open terms have to be determined, point-by-point, for each candidate subshape in *c*.

FIGURE 1. *EXAMPLE OF PARAMETRIC SUBSHAPE MATCHING.*



In general, when there are *k* open terms, there are $\binom{n}{k}$ possibilities. Even assuming that testing against each possibility costs unit time (it is typically more expensive in reality), when *k* is close to *n/2*, the time complexity is a super-polynomial. To give a concrete example, the possible number of tests is $7.5 \times 10^7$when $k = 5$, $n = 100$; 1.7×10^{13} when, $k = 10$, $n = 100$; and 1.0×10^{29} when $k = 50$, $n = 100$. In practice, a parametric shape rule with $k = 5$, $n = 100$ is not complicated. Note, however, that when *k* is more than *n/2*, the number of possible tests begins to decrease.

2.3. Parametric subshape recognition is NP-hard

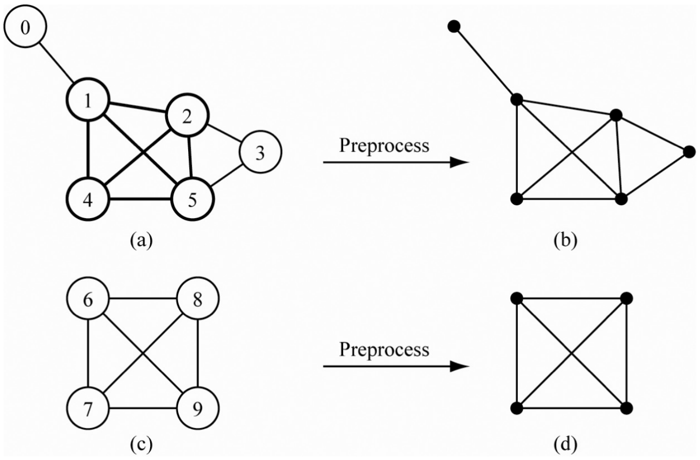
We show that parametric subshape recognition, in general, is NP-hard, by reducing the problem to finding certain cliques in a graph. That is, if we can solve parametric subshape recognition in polynomial time, then we can solve the theoretical clique problem in polynomial time, which is known to be NP-hard (Cormen *et al.* 2004).

A *clique* in an undirected graph $G = (V, E)$ is a subset $V' \subseteq V$ of vertices, each pair of which is connected by an edge in *E*. In other words, a clique is a complete subgraph of *G*. The size of a clique is its number of vertices. Figure 2c is an example of a clique of size 4. The clique problem is an optimization problem of finding a clique of maximum size in a graph. For example, in the graph of Figure 2a, the maximum clique is 4 (the bolded subgraph).

First, we preprocess graph G (Figure 2a) to get G' (Figure 2b) by treating vertices in G as end points of the incident edges, assigning unique 2D coordinates to all vertices so that no three vertices are collinear, and enforcing all arcs to be straight lines. This can be done in $O(|V|^2)$ time. Note that G' is actually a 2D shape, and we use it as c.

We next generate a complete graph G_k with k vertices (Figure 2c) and similarly preprocess it to obtain G_k' (Figure 2d). This can be done in $O(k^2)$ time. Note that $G_k'$is actually another 2D shape, and we use it as a, with all the points as open terms.

FIGURE 2. *EXAMPLE OF FINDING A CLIQUE OF SIZE 4.*



If there is an algorithm, which is capable of detecting the existence of subshape $g(c)$ in c by automatically finding an appropriate assignment of g in a polynomial time, then we can use the algorithm to detect the existence of subshape $g(G')$ in G' by automatically finding an appropriate assignment of g in a polynomial time, say, T_k. By the particular way that we processed graph G and G_k, the existence of subshape $g(G_k')$ in G' is identical to the existence of G_k in G. In other words, we can use the algorithm to detect the existence of G_k in G in a polynomial time of T_k, plus the preprocessing time.

By doing the above preprocessing and detecting for $k = \{1 \ldots |V|\}$ sequentially until the answer is false, we can find the clique of maximum size in time of $O((O|V|^2) + \sum_{k=1}^{|V|}(T_k + O(k^2)))0$, which is polynomial time. This is a contradiction as the clique problem is known to be NP-hard.

From the above, we can conclude that, in general, it is impossible to design a polynomial algorithm for parametric subshape recognition for shapes of Type V. As a consequence, it is impossible to implement a parametric shape grammar interpreter for shapes of Type V with polynomial time complexity. This conclusion can be extended to shapes with curves (Type VII) and to shapes in 3D

(Types VI and VIII), with the proviso that a straight line is a special case of a curve and 2D is a special case of 3D.

As a result, algorithms, in the literature, particularly, those that deal with parametric shapes fall into two categories. The first category handles special shapes; the second category is more general, with exponential time complexity, which is only practical for shapes of small sizes. The implication for practice is that the best we can do is to design and implement a parametric shape grammar interpreter, which is capable of handling a subset of grammars.

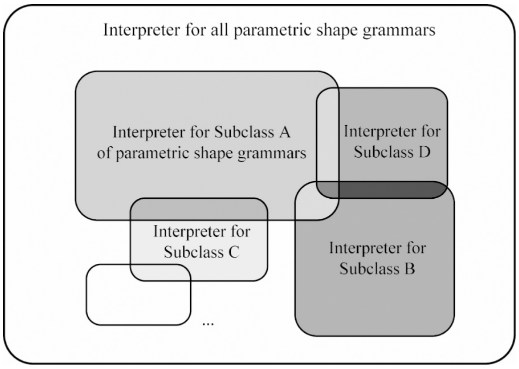
3. COMPUTATION-FRIENDLY SHAPE GRAMMARS

As we have seen, interpreting parametric shape grammars in general is NP-hard. However, there are categories of shape grammars whose implementation is tractable. Shape grammars, which capture certain building styles, generally fall into this category. Consider, for example, the Queen Anne (Flemming 1987) and Prairie house grammars (Koning and Eizenberg 1981). These are examples of parametric shape grammars, in which shape rule application does not depend on emergent shapes. Markers drive shape rule application, and configurations are rectangular or can be approximated as such. Moreover, parameterization is often limited to the height or width of a room, or to the ratio of a room split. Shape rules typically relate to adding a room, to subdividing a room, or to refinements such as adding windows, doors, etc.

This, together with the conclusion drawn in Section 2.3, leads to a paradigm for practical, "general" parametric shape grammar interpreters, as shown in Figure 3. We make the assumption that interpreters for shape grammars belonging to different subclasses, collectively, will cover most parametric shape grammars. It should be noted that the classification is considered to be "better" when the number of subclasses is smaller, and when, simultaneously, the scope covered, collectively, is larger. Possible ways of classifying shape grammars needs further research.

Aside from the internal characteristics of shape grammars, there are other factors that influence their computational tractability, for example, how shape grammars are designed and described. Traditionally, a shape grammar is designed to simply and succinctly describe an underlying building style, with little consideration on how the grammar can be implemented. For example, as is often found in the literature, descriptions of the form "If the back or sides are wide enough, rule 2 can be used…" are inherently counter-computable. As a result, in order to translate this into programming code, shape rules have to be specified in a *computation-friendly* way: that is, shape rules need to be quantitatively specified; furthermore, there should be enough precision in the specification to disallow generation of ill-dimensioned configurations.

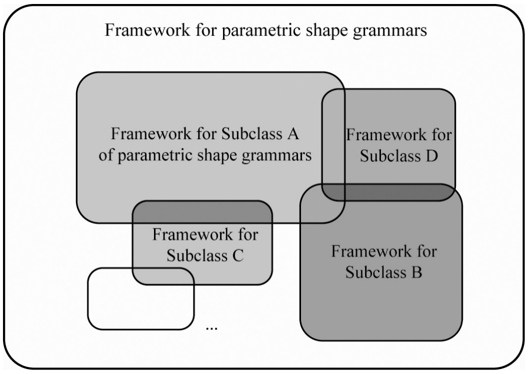
FIGURE 3. *A PARADIGM FOR PRACTICAL "GENERAL" PARAMETRIC INTERPRETERS.*


Interpreter for all parametric shape grammars
Interpreter for Subclass A of parametric shape grammars
Interpreter for Subclass D
Interpreter for Subclass C
Interpreter for Subclass B
...

Closer examination also shows that there may be more than one way to describe a particular shape rule; it is possible that one way is easy to compute, and the other, might be computationally intractable. As a result, it is desirable to design an application program interface (API) as the framework to support the design of shape grammars; then, shape grammars that follow the framework are guaranteed to be computationally tractable. Such a framework requires an underlying data structure, and basic manipulation algorithms. Moreover, for ease of code translation, a meta-language built on top of the basic manipulation algorithms should also be developed. As grammars in different classes typically have differing underlying structures, the appropriate underlying data structure for the framework will be different. Ideally, the interpreter for any subclass of shape grammars can be supported on a single framework. Consequently, the overall framework for parametric shape grammars comprises a series of sub-frameworks, one for each subclass of shape grammars, as shown in Figure 4, which is isomorphic to Figure 3. As the overall framework is capable of ensuring computability, we term shape grammars following such a framework as *computation-friendly*.

In this paper, the concept of the overall framework will be detailed by examining a sub-framework. It is advantageous to select a sub-framework for a subclass with the largest population. It turns out that shape grammars capturing building styles happen to be a good choice. Of the thirty one shape grammar applications reviewed by Chau *et al.* (2004), about half deal with architectural plans. Moreover, conventional buildings, which are buildings with rectangular spaces or dominated as such, are often the subjects. Consequently, we focus on a sub-framework for shape grammars capturing corpora of conventional building types.

***FIGURE 4.** FRAMEWORK FOR PARAMETRIC SHAPE GRAMMARS: ONE SUB-FRAMEWORK FOR EACH SUBCLASS*


Framework for parametric shape grammars
Framework for Subclass A of parametric shape grammars
Framework for Subclass D
Framework for Subclass C
Framework for Subclass B
...

4. A SUB-FRAMEWORK OVER PARAMETRIC 2D RECTANGULAR SHAPES

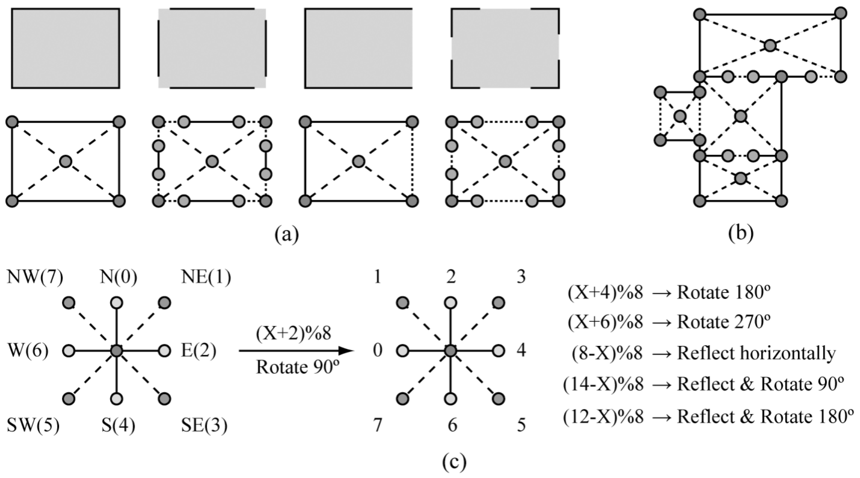
Shape grammars that capture corpora of conventional building types belong to a special subset. Here, shape rules are parametrically specified in such a way as to make implementation tractable. Spaces (rooms) are central to buildings —whence, to shape grammars that describe building styles. Such grammars generally start with a rough layout; details, such as openings and staircase, are added at a subsequent stage. There are two main ways of generating a layout; space subdivision and space aggregation. Combination of the two ways is possible. Parametric subshape recognition is, typically, of searching a special room under certain constraints—actually, label matching.

4.1. Graph-like data structure

The interpreter needs a data structure to represent layouts with rectangular spaces; that is, the data structure contains topological information of the spaces, as well as concrete geometry data. A rectangular space is specified by a set of walls in such a way that the space is considered rectangular by the human vision system. In Figure 5a, among other variations, a space can be specified by four walls jointed to one another, four disjoint walls, three walls, or framed by four corners.

A graph-like data structure is used to record such variations. There is a boundary node for each corner of the rectangular space, as well as a node for each end of a wall. Nodes are connected by either a wall edge (solid line) or an empty edge (dotted line). A central node represents the room corresponding to the space, and connects to the four corners by diagonal edges (dashed lines). It is needed for manipulating boundary nodes, such as dividing a wall through node insertion, deleting a wall by changing its edge type to empty, and so on. Additionally, information about the room is recorded in the room node, e.g., a staircase within the space. Unlike traditional graph data structures, the angle

FIGURE 5. *GRAPH-LIKE DATA STRUCTURE FOR RECTANGULAR SPACES.*



at each corner is set to be a right angle. A node has at most eight neighbors. A set of such graph units can be combined to represent layouts comprising rectangular rooms (Figure 5b).

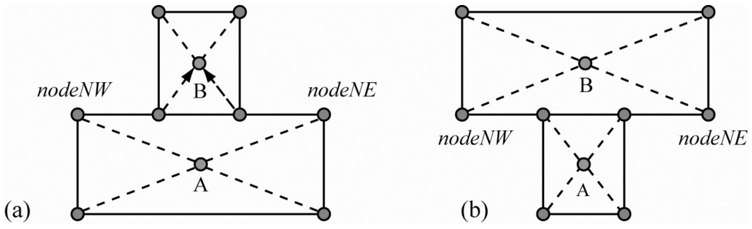
It is necessary for the data structure to support geometric transformations. For layout of rectangular spaces, applicable transformations are translation, rotation, reflection, glide reflection, and scale (uniform and non-uniform). Moreover, rotations are multiples of 90° and reflections are about the either horizontal or vertical.

The transformations are easily implemented on the data structure through manipulating indices. Each neighbor of a node is assigned an index from 0 to 7; indices are transformed using simple modulo arithmetic (Figure 5c). For example, index+2 (modulo 8), rotates, ccw, neighbor vertices through 90°. Other rotations and reflections are likewise achieved. By viewing the original neighbor relationship for each node with the transformed indices, we obtain the same transformation of the whole graph. Moreover, we need manipulate only the interior layout instead of the left side of a shape rule. This gives the same result, albeit simpler. Thus, we only need to consider how to apply shape rules in the case of translation, which is automatically applicable to the configuration under different possible transformations.

4.2. Common functions and meta-language

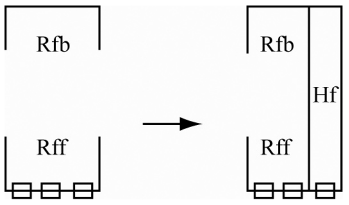
Application of shape rules is achieved by manipulating the data structure. Examples of common manipulations include finding a room with a given name, finding the north neighbor of a given room, finding the shared wall of two given rooms, subdividing and merging rooms, etc. Among these functions, some are easier to implement, others need careful design of the underlying algorithms. The following is the reasoning of the algorithm backing the function of finding the shared wall of two given rooms:

FIGURE 6. *FINDING WSTART AND WEND.*



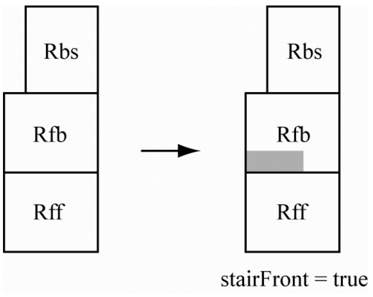
In the data structure, the shared wall of two given rooms is represented as a list of nodes connected by edges; the simplest form of a shared wall is given by two nodes connected by an edge. For two given input room nodes, *A* and *B*, in general, *A* and *B* may not be neighboring rooms at all. If, however, *A* and *B* are real neighbors, *B* can be in any one of four directions from *A*. Therefore, it is necessary for the algorithm to test all four sides of *A*; for each particular side, it is simply to test whether *B* is in the north neighbors under a given transformation *T*. If *B* is determined as a neighbor of *A* at a given side, the exact start node, *wStart*, and end node, *wEnd*, need to be further determined. The edge from the north-east node, *nodeNE*, to the north-west node, *nodeNW*, of room *A* under transformation *T* is guaranteed to be the wall of room *A*, but not necessarily the wall of room *B* (Figure 6a). As a result, *wStart* may be actually a node to the right of *nodeNE.* This node is found by traversing from *nodeNE* to *nodeNW*, testing whether *B* is its north-west neighbor or not. Similarly, *wEnd* may be actually a node to the left of *nodeNW*. This node is found by traversing from *nodeNW* to *nodeNE* and testing whether *B* is its north-east neighbor or not.

FIGURE 7. *TWO SAMPLE RULES AND THEIR META-LANGUAGE.*

	*Precondition:* *'Rff'* exists, and is three-bay (2 windows and 1 door). *Transformation*: N/A This rule adds to the front block a hall way centered about the front door.

```
if
roomExists('Rff') && room('Rff').threeBays()
then
a = room('Rff').horSplit('*','doorCentral')
b = room('Rfb').horSplit('*','doorCentral')
roomMerge(a, b).name('Hf')
```

(a) Rule 1 and its meta-language



Explicit condition:
No staircase. '*Rfb*' and '*Rff*' exist and are neighbors.
Implicit condition: No 'SfS.'
Width of front block is ≤ 18'.
Overall condition:
stairFront is false. '*Rfb*' exist. No '*SfS*.'
Width of front block is ≤ 18'.
Transformation: N/A
Add a staircase to room '*Rfb*.'

```
if
get('stairFront')=false && roomExists('Rfb') &&
roomExists('SfS')=false && get('widthOfFrontBlock')≤18
then
room('Rff').addStair(
position='lowerRightCorner', width=4, height=6)
```

(b) Rule 2 and its meta-language

All these functions collectively form an API, which grammar designers can apply to ensure computability of their designed grammars. Moreover, these functions support describing grammars in a meta-language so that shape rules can be easily translated into pieces of code. Figure 7 shows two such examples. The meta-language is in the form of *if-then*; the *if*-part determines whether the rule is applicable or not and the *then*-part is how to do the rewriting. Essentially, the meta-language is a set of function calls.

5. SHAPE AND GRAPH GRAMMARS

Graphs provide a natural way of describing complex situations on an intuitive level. Graph grammars (Brouno 1990; Rozenberg 1997) are rule-based modification of graphs through graph rule application. Graph grammars have been developed as an extension to graphs of formal string grammars. Among string grammars, context-free grammars have proven extremely useful in practical applications and powerful enough to generate a wide spectrum of interesting formal languages. Analogously, most research focuses on 'context-free' graph grammars, which typically means local modifications of graphs without 'global' constraints. Rule application on graphs is, typically, label-driven. The two basic choices for rewriting a graph are: *node replacement* and *hyperedge replacement*.

Shape grammars are rule-based rewriting system of shapes. In many ways, it can be viewed as an extension to shapes of formal string grammars. The shared root implies the close connection between graph and shape grammars.

As an example, Drews and Kreowski (1999) used collage grammars to generate pictures, e.g., Sierpinski gasket. This suggests that there is an intersection between both graph and shape grammars.

Consequentially, shape grammars can take advantage of graph grammar research results, especially for those 'context-free' shape grammars; that is, the shape rewriting happens locally. For example, as shown in Figure 8, the ice-ray grammar (Stiny 1977), which is essentially a process of polygon subdivision, can be implemented as a graph grammar. Each point correspond a vertex and each polygon is decorated with a hyperedge (the vertices drawn in squares together with dashed tentacles). Figure 9 shows a sample rule applied in Figure 8: the right-hand hyperedges are labeled either *S* as candidates for further rule application or *T* for no more rule application; this is based on certain criteria, for example, on the area of the underlying polygon.

Graph and shape grammars deal with differing fields of application. For example, graph grammars are found in computer science related applications, while shape grammars apply mainly to (architectural or mechanical) design. Approaches by which graph grammars are investigated could be instructive to shape grammar research. For instance, context-freeness could be an important criterion to classify shape grammars.

On the other hand, shapes differ significantly from graphs and so do their grammars. Shape grammars do not deal solely with pure pictures; they are usually imbued with semantics, and represent designs in reality. In this respect, dimension is typically important. Graph grammars, however, are inherently dimensionless. Semantics make most shape grammars context-sensitive; this limits whatever advantages provided by the nice theorems in graph grammars.

FIGURE 8. *IMPLEMENTING THE ICE-RAY GRAMMAR AS A GRAPH GRAMMAR.*

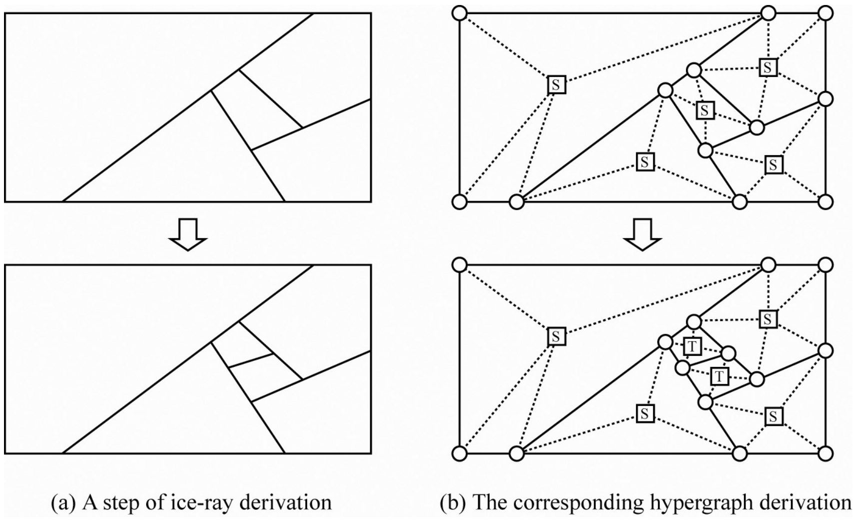

S
T
(a) A step of ice-ray derivation
(b) The corresponding hypergraph derivation

FIGURE 9. *THE SHAPE AND GRAPH RULES APPLIED.*

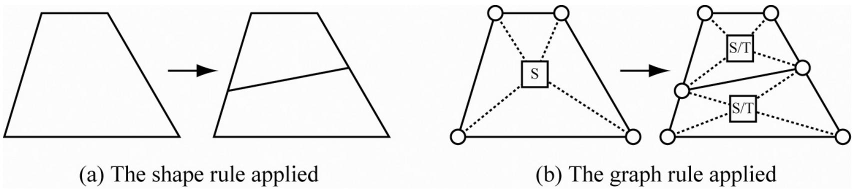
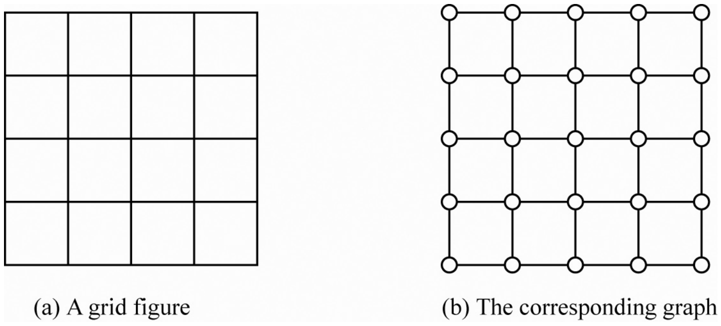


FIGURE 10. *SUBSHAPE RECOGNITION IN A GRID FIGURE.*



Graph grammars are essentially label-driven; however, this does not offer much help in solving the fundamental problem of subshape recognition in shape grammars. As a classical example (Figure 10), there are many, actually an uncountable, number of square subshapes. Converting the grid figure to a graph does not change the basic characteristic of the problem.

6. REMARKS

It is clear that appropriate classification of shape grammars is critical to the success of the framework proposed. In this respect, approaches in graph grammar research appear to be instructive—similar approaches might prove useful in investigating shape grammars for such classification. Equally important is to enrich the framework with additional sub-frameworks so that its generality can be further investigated.

ACKNOWLEDGEMENTS

This research was supported in part by a grant from US Army Corps of Engineers, Engineer Research and Development Center – Champaign. Any opinions, findings, conclusions or recommendations presented in this paper are those of the authors and do not necessarily reflect the views of CERL.

REFERENCES

Brouno, C., 1990, Graph Rewriting: An Algebraic and Logic Approach, *Handbook of Theoretical Computer Science,* vol. B: *Formal Models and Semantics,* MIT Press.

Chau, H.H., Chen, X., McKay, A. and Pennington, A., 2004, Evaluation of a 3D Shape Grammar Implementation, *in* J.S. Gero (ed.), *Design Computing and Cognition '04.* Boston.

Cormen, T.H., Leiserson, C.E., Rivest, R.L. and Stein, C., 2004, *Introduction to Algorithms, Second Edition*, The MIT Press.

Drewes, F. and Kreowski, H.J., 1999, Picture Generation by Collage Grammars, *Handbook of Graph Grammars and Computing by Graph Transformation,* vol. 2, *Applications, Languages, and Tools,* World Scientific Publishing Co.

Flemming, U., 1987, More than the Sum of Parts: the Grammar of Queen Anne Houses, *Environment and Planning B: Planning and Design,* 14, 323-350.

Koning, H. and Eizenberg, J., 1981, The Language of the Prairie: Frank Lloyd Wright's Prairie houses, *Environment and Planning B: Planning and Design,* 8, 295-323.

Krishnamurti, R., 1981, The construction of shapes, *Environment and Planning B: Planning and Design,* 8, 5-40.

McCormack, J.P. and Cagan, J., 2002, Supporting Designer's Hierarchies through Parametric Shape Recognition, *Environment and Planning B: Planning and Design,* 29, 913-931.

Rozenberg, G. (ed.), 1997, *Handbook of Graph Grammars and Computing by Graph Transformation*, vol. I., Foundations, World Scientific Publishing Co.

Stiny, G., 1977, Ice-ray: a note on Chinese Lattice Designs. *Environment and Planning B: Planning and Design,* 4, 89-98.

Stiny, G., 1980, Introduction to Shape and Shape Grammars. *Environment and Planning B: Planning and Design,* 7, 343-351.

Stiny, G., 2006, *Shape: Talking about Seeing and Doing*, MIT Press, Cambridge.