

A Technique for Implementing a Computation-Friendly Shape Grammar Interpreter

Kui Yue and Ramesh Krishnamurti
Carnegie Mellon University, USA

We discuss technical issues related to implementing a general interpreter for shape grammars directed at describing building styles, including a graph-like data structure and the concept of computation-friendly grammars.

Introduction

We are investigating how to determine the interior layout of buildings given three pieces of information: its footprint; a reasonably complete set of exterior features; and a shape grammar that describes the building style and hence, the building [1]. We have developed an approach that relies on the fact that, when applied exhaustively, a shape grammar generates, as a tree, the entire layout space of a style. The approach begins with estimating a partial layout, by resolving constraints on the input features. From this estimation, further spatial and topological constraints are extracted. These constraints are then used to prune the layout tree. The layouts that remain correspond to the desired outcomes.

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 staircases, are added at a subsequent stage. There are two main ways of generating a layout: space subdivision, e.g., as in the rowhouse grammar (see sequel), and space aggregation, e.g., as in the Queen Anne grammar [2]. Combination of the two ways is possible. Consequently, here, shape rules tend to add a room, partition a room, additionally to refine a partial layout by inserting features such as doors, staircases, etc.

Thus, pruning a layout tree, effectively, is to find a tree node with layout equal to the partial layout estimation, and continue to apply the subsequent shape rules. Such a node is typically internal, although it could be, luckily, the root node. In each case, the approach essentially requires a parametric shape grammar interpreter that caters to a variety of building types; for layout determination, it would be impractical to implement individual interpreters for each grammar.

A general parametric shape grammar interpreter is an unresolved topic of research [3]. However, shape grammars that capture corpora of conventional building types – that is, composed of rectangular spaces bounded within a rectangular form – belong to a special subset. Informally, here, shape rules are parametrically specified in such a way as to make implementation tractable. Such parametric grammars do not rely on emergent or ambiguous shapes. Markers tend to drive shape rule application. Moreover, parameterization is limited to just a few kinds of variables, for example, the height or width of a space, or the ratio of partitioning a space.

The implementation of an interpreter is non-trivial. In this paper, we describe a graph-like data structure to support a general interpreter for a particular class of shape grammars. We consider counter-computational hindrances that commonly occur in shape grammars designed in the traditional manner, leading to the concept of computation-friendly shape grammars. This is illustrated by two grammars developed for nineteenth-century rowhouses located in the Federal Hill district, Baltimore [4].

Transformations of Shape Rules

Shape rules apply under a transformation [5]. Unless stated otherwise, the allowable transformations are taken to be affine, that is, preserving parallelism. However, the more commonly used transformations are similarities, which preserve angles. These are Euclidean transformations with uniform scale. The Euclidean transformations, namely, translation, rotation, reflection, and glide reflection, preserve distance. See Table 1.

For shape rule application under arbitrary non-affine transformations, shapes have to be defined parametrically [5]. For example, in Figure 1, to apply shape rule (a) to shape (b), the rule has to be considered as a parametric schema. Depending on the allowable transformations, the application of shape rule (a) to shape (c) can be considered to be either parametric or non-parametric. If the allowable transformations are similarities, the shape rule has to be parametric; if affine, then the shape rule can be applied under anamorphic (non-uniform) scale, in which case it can be con-

sidered to be non-parametric. Such distinctions are important for computational implementation. As discussed later, a computation-friendly shape grammar must specify the allowable transformations for the shape rules.

Table 1 Types of allowable transformations

	Translation	Rotation	Reflection	Glide reflection	Uniform scale	Anamorphic scale	Shear/Strain
Euclidean	Yes	Yes	Yes	Yes	No	No	No
Similarity	Yes	Yes	Yes	Yes	Yes	No	No
Affine	Yes	Yes	Yes	Yes	Yes	Yes	Yes

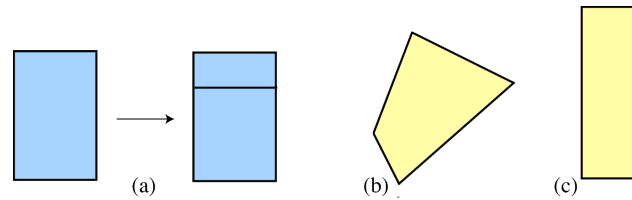


Fig. 1. A shape rule example

Data Structure for Layouts with Rectangular Spaces

The interpreter needs a data structure to represent layouts with rectangular spaces; that is, a data structure that contains both topological information of the spaces as well as the concrete geometry (for now, 2D) data of the layout including walls, doors, windows, staircases, etc. It needs to support viewing the layout as whole, viewing the layout from a particular room with its neighborhood, or simply focusing on a particular room itself. Moreover, the data structure needs to support Euclidean as well as both uniform and anamorphic scale transformations.

A graph-like data structure

A graph-like data structure has been designed to specify such rectangular spaces. A rectangular space (usually a room) is a space defined by a set of walls in a way that the space is considered to be rectangular by the human vision system. As shown in Figure 2, among other variations, such a space can be defined by four walls jointed to each other, four disjointed walls, three walls, or framed by four corners.

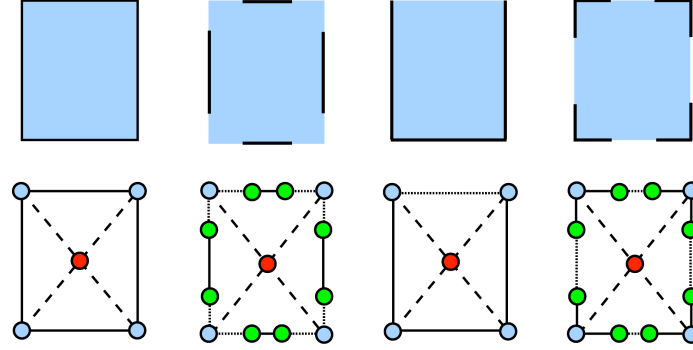


Fig. 2. Examples of rectangular spaces and graph-like data structures

There is a boundary node for each corner of the rectangular space, as well as a node for each endpoint of a wall. These 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 of room units, such as dividing a wall through node insertions, creating an opening in a wall by changing the opening's edge type to *empty*, and so on. More information about a room is recorded in the room node, e.g., a staircase within the space. Windows and doors are assigned as attributes of wall edges. Further, unlike traditional graph data structures, the angle 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 complex layouts comprising rectangular spaces.

Transformations with the graph-like structure

Under the assumption that the target layout comprises only rectangular spaces, the allowable transformations are Euclidean with uniform and anamorphic scaling. As shape rule application is marker-driven, translation is automatically handled. The graph-like data structure is capable of easily handling uniform and anamorphic scaling, by firstly matching room names, then markers on corner nodes, and lastly, by comparing possible room ratio requirements.

As a result, only rotations and reflections remain to be considered. As the spaces are rectangular, rotations are limited to multiples of 90° and reflections are about either the horizontal or vertical. Moreover, a vertical reflection can be viewed as a combination of a horizontal reflection and a

rotation. Hence, any combination of reflections and rotations is equivalent to a combination of horizontal reflections and rotations. Consequently, the following transformations are all we actually need to consider:

- $R0$: default; no rotation, with possible translation and/or scale.
- $R90$, $R180$, $R270$: a rotation through 90° , 180° , and 270° , respectively, with possible translation and/or scale.
- $RR0$, $RR90$, $RR180$, $RR270$: (first a rotation of 0° , 90° , 180° , or 270° , followed by a horizontal reflection) horizontal reflection, vertical reflection, and their combinations.

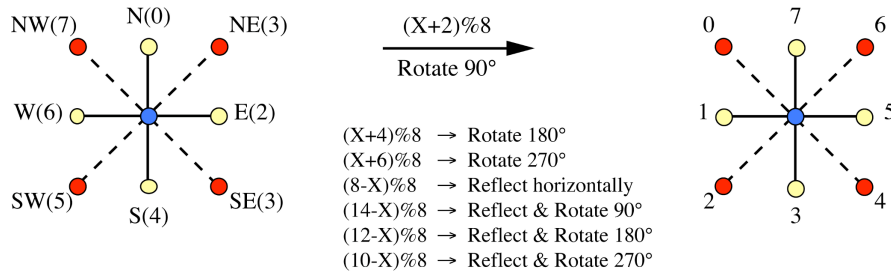


Fig. 3. Transformation of the graph-like data structure

As shown in Figure 3, transformations can be implemented on the data structure by index manipulation. Each of the eight possible neighbors of a node is assigned an index from 0 to 7; indices are then transformed simply by modulo arithmetic. For example, $\text{index}+2$ (modulo 8), rotates counter-clockwise 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 the shape rule. This gives the same result, and is much simpler to achieve. Thus, we only need to consider how to apply shape rules with the default transformation, which is automatically applicable to the configuration under different possible transformations.

Common Functions for the Graph-like Data Structure

With the graph-like data structure, a layout is represented by an eight-way doubly linked list formed by nodes and edges. Shape rule application manipulates this structure, and a set of common functions shared by the shape rules can be identified. The functions are implemented in an object-oriented fashion.

Design of classes

LNodeCorner and *LNodeRoom* classes represent a corner node and a room node, respectively. Other nodes are represented by *LNode* class. All edges are represented by *Edge* class with an attribute representing different edge types. Theoretically, knowing the handle to a node or edge is sufficient in order to traverse the entire layout. For easy manipulation, an *InteriorLayout* class is defined to represent an interior layout configuration. There are several different ways to view an *InteriorLayout* object: i) as a layout with certain status marker, ii) as a list of rooms (room nodes), and iii) as a list of nodes and edges. Different views are useful under different contexts. For example, it is convenient to use view iii) to display the underlying layout: drawing all edges as well as the associated components first, and then drawing all nodes as well as associated components. To accommodate these different views, the *InteriorLayout* maintains the following fields:

- A status marker
- Name: for display and debugging purpose
- A hashmap of a room name to a list of room nodes: for fast retrieval of one or more room nodes with a given name
- A list of room nodes for the entire layout
- A list of all nodes for the entire layout
- A list of all edges for the entire layout
- A hashmap of attributes to values for other status values particular for a special shape grammar

Examples of common functions

Examples of common manipulations include finding a room with a given name, finding the north neighbor(s) of a given room, finding the shared wall of two given rooms, etc. The sequel describes the algorithm and pseudo code for these examples.

Finding room(s) with a given name

In the data structure, a room node represents a room. An *InteriorLayout* object maintains a hashmap of room names to lists of room nodes. Thus, finding room(s) with a given name is simply to query the hashmap with the room name as input.

```
findRoomNodes(Name)
```

Query the name-to-rooms hashmap with parameter *Name*.

Finding the north neighbor(s) of a given room

Finding the north neighbor(s) of a given room is a special case of finding neighbor(s) of a given room. It turns out all that finding neighbor functions in the other three directions can be implemented as finding the north neighbor(s) under a certain transformation. For example, the east neighbor(s) of a given room is the same as the north neighbor(s) of the given room under a $R90$ transformation.

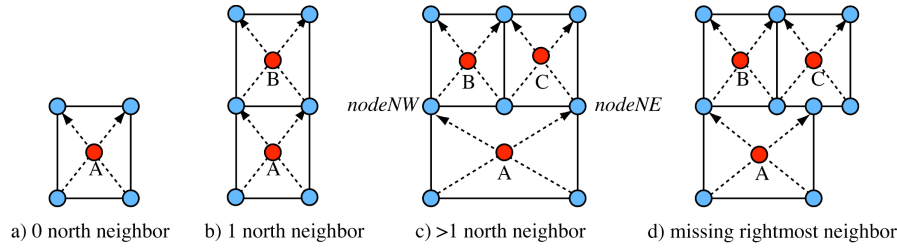


Fig. 4. Different cases of north neighbor(s) of a room

A room may have zero, one, or more north neighbors (Figure 4), which can be represented by a list of room nodes. Intuitively, to find the north neighbor(s) of A , we could start by finding A 's north-east corner node, $nodeNE$, and north-west corner node, $nodeNW$. Then, we traverse through each corner node from $nodeNE$ (inclusive) to $nodeNW$ (exclusive) along the westerly direction to find its north-west neighbors. All north-west neighbors found are the desired room nodes. For example, in Figure 4c, the north neighbors found are B , and C . However, as shown in Figure 4d, this intuitive algorithm will miss the rightmost neighbor room when the $nodeNE$ is on the south edge of that neighbor room, and is not the end node. Therefore, we need to modify the intuitive algorithm to have the correct start node and end node to loop through.

It can be proven that the $nodeNW$ is always the correct end node as a north neighbor B has to overlap with room A , which means room B must have a south-east corner node, $nodeSE$, at the right side of $nodeNW$ (Figure 5a), or is $nodeNW$ (Figure 4c). Otherwise, B is not a north neighbor of A .

The starting node can be either $nodeNE$, or a node to the right of $nodeNE$ (Figure 5b). If $nodeNE$ is not the start node, then it has neither north-west nor south-west neighbors, since having either neighbor means that $nodeNE$ is the correct start point (Figure 5c), which is a contradiction. However, the reverse is not true; as shown in Figure 5d, $nodeNE$ has neither a north-west nor south-west neighbor, but $nodeNE$ is still the correct start node. That is, the only condition for a node, $nodeSE$, to the right of $nodeNE$, to be the correct start node, it must have a north-west neighbor.

Therefore, under the condition that *nodeNE* has no north-west and south-west neighbor, the algorithm searches for the first node, which is to the right of the *nodeNE* with a north-west neighbor. If such a node is found, it is the real start node. If a *null* neighbor is found, *nodeNE* is still the correct start node. The pseudo code is given below.

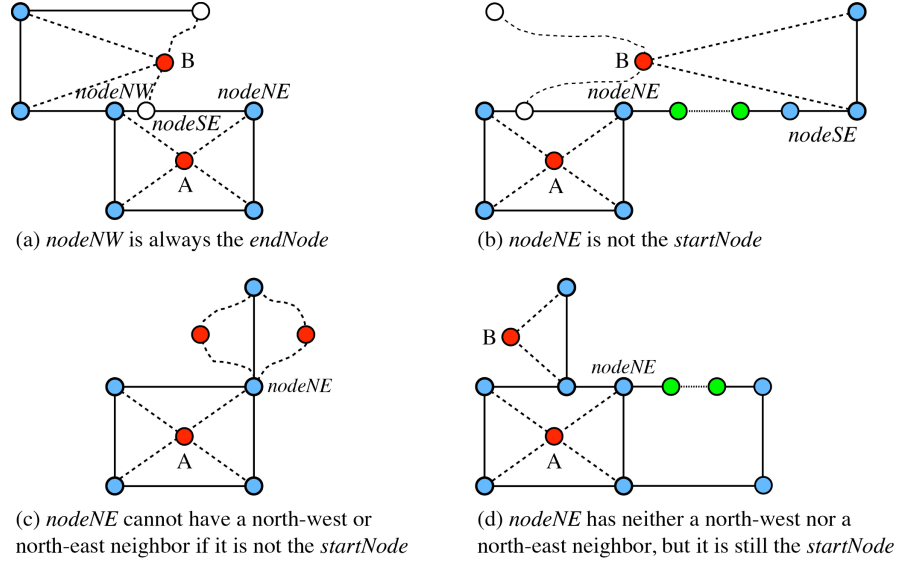


Fig. 5. The start and end node for finding neighbor room(s)

```

findNorthNeighbors(A, T)
  (all operations related to directions are under transformation T)
  endNode  $\leftarrow$  north-west neighbor of A
  nodeNE  $\leftarrow$  north-east neighbor of A
  startNode  $\leftarrow$  nodeNE
  if nodeNE has neither north-west nor north-east neighbor
    search for a right neighbor, node, of nodeNE, with a north-west neighbor
    if found, startNode  $\leftarrow$  node
    go through each node in between startNode (inclusive) and endNode (exclusive), and get all north-west neighbors, neighbors
  return neighbors

findEastNeighbors(A) // Other neighbors are likewise defined
return findNorthNeighbors(A, R90)

```

Finding the shared wall of two given rooms

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. The pseudo code is given below.

```

findWallShared( $A, B$ )
  transformations  $\leftarrow \{R0, R90, R180, R270\}$ 
  for each transformation in transformations
    results  $\leftarrow$  findNorthWallShared( $A, B$ , transformation)
    if results is not null
      return {results, transformation}
  return null

findNorthWallShared( $A, B$ , transformation)
  if  $B$  not in neighbors  $\leftarrow$  findNorthNeighbors( $A$ , transformation)
    return null
  nodeNE  $\leftarrow$  north-east neighbor of  $A$ 
  nodeNW  $\leftarrow$  north-west neighbor of  $A$ 
  wStart  $\leftarrow$  null
  wEnd  $\leftarrow$  null
  for each node, node, from nodeNE to nodeNW
    if north-west neighbor of node is  $A$ 
      wStart  $\leftarrow$  node, and break
  if wStart is null, then wStart  $\leftarrow$  nodeNE (Figure 6b)
  for each node, node, from nodeNW to nodeNE
    if north-east neighbor of node is  $B$ 
      wEnd  $\leftarrow$  node, and break

```

```

if  $wEnd$  is null
   $wEnd \leftarrow nodeNW$  (Figure 6b)
return  $\{wStart, wEnd\}$ 

```

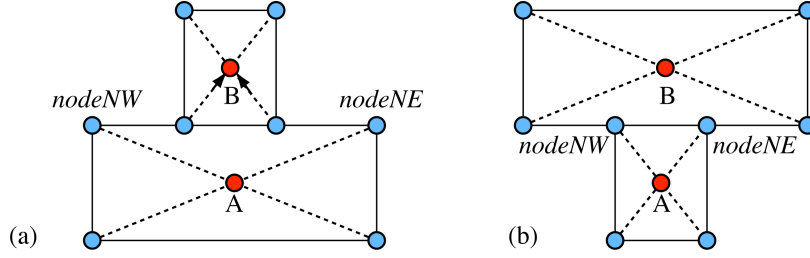


Fig. 6. Finding $wStart$ and $wEnd$

Computation-Friendly Shape Grammar

It is claimed that, in design, ambiguity serves a positive and deliberate function [6]. In principle, shape grammars can be devised to take advantage of ambiguity in creating novel designs [5]. However, ambiguity, in general, is inherently counter-computable, and the level of ambiguity has to be controlled for any computational implementation to be tractable.

Traditionally, a shape grammar is designed to simply and succinctly describe the 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 into programming code, shape rules have to be specified in a *computation-friendly* way: that is, shape rules need to be quantitatively specified; moreover, there is enough precision in the specification to disallow generation of ill-dimensioned configurations. For a general shape grammar interpreter, this requires the underlying shape rules to follow a certain computation-friendly framework. In the following, the concept of computation-friendly is further elaborated upon through comparing two shape grammars for the same rowhouse corpus.

A traditional rowhouse grammar

The Baltimore rowhouse grammar, developed by Casey Hickerson, consists of 52 shape rules that generate first floor configurations with features of stairs, fireplaces, windows, exterior doors and interior doors. Rules are organized into phases, progressing from the major configurations that constrain the design process to minor configurations that follow logically from other configurations, namely: I) Block generation: rules 1~4; II) Space generation: rules 5~7; III) Stair generation: rules 8~17; IV) Fireplace generation: rules 18~22; V) Space modification: rules 23~24; VI) Front door and window generation: rules 25~29; VII) Middle and back door and window generation: rules 30~39; and VIII) Interior door generation: rules 40~52.

Rules are marked as required (*req*) or optional (*opt*). Required rules must be applied if applicable while optional rules may be applied at the interpreter's discretion. The decision whether to apply an optional rule directly impacts the overall design. In effect, the final design is determined by the set of optional rules that were applied. Whenever a rule is applied, it must be applied exhaustively; that is, the rule must be applied to every subshape that matches the rule's left-hand-shape. Finally, rules must be applied in sequence: after Rule x has been applied exhaustively, only Rules $x+1$ and greater may be applied.

Like other shape grammars, labels are used in two ways: to control where shape rules may apply, and to ensure that mutually exclusive rules cannot be applied to the same design. Spaces and stairs are labeled with two or three characters that indicate the general location of the space or stair within the house. For instance, *Rfb* indicates a Room in the front block of the house that is oriented toward the back, a dining room. Wall labels are always of the form $x(y)$ where x is a label for a space that the wall bounds (or P in the case of certain perimeter walls) and y is a one letter code indicating the side of the space the wall defines. For example, the front wall of the room labeled *Rfb* is labeled *Rfb(f)*. Within some rules, variables are used to match more than one label: the character $*$ matches any string of characters while the string $\{x/y\}$ matches the strings x or y . *Boolean* global labels are used to ensure that mutually exclusive rules are not applied with default value *false*. Due to space limitation, only the rules from phases I, II, III (all but the last) and V are shown here (Figure 7). A sample derivation is given in Figure 8.

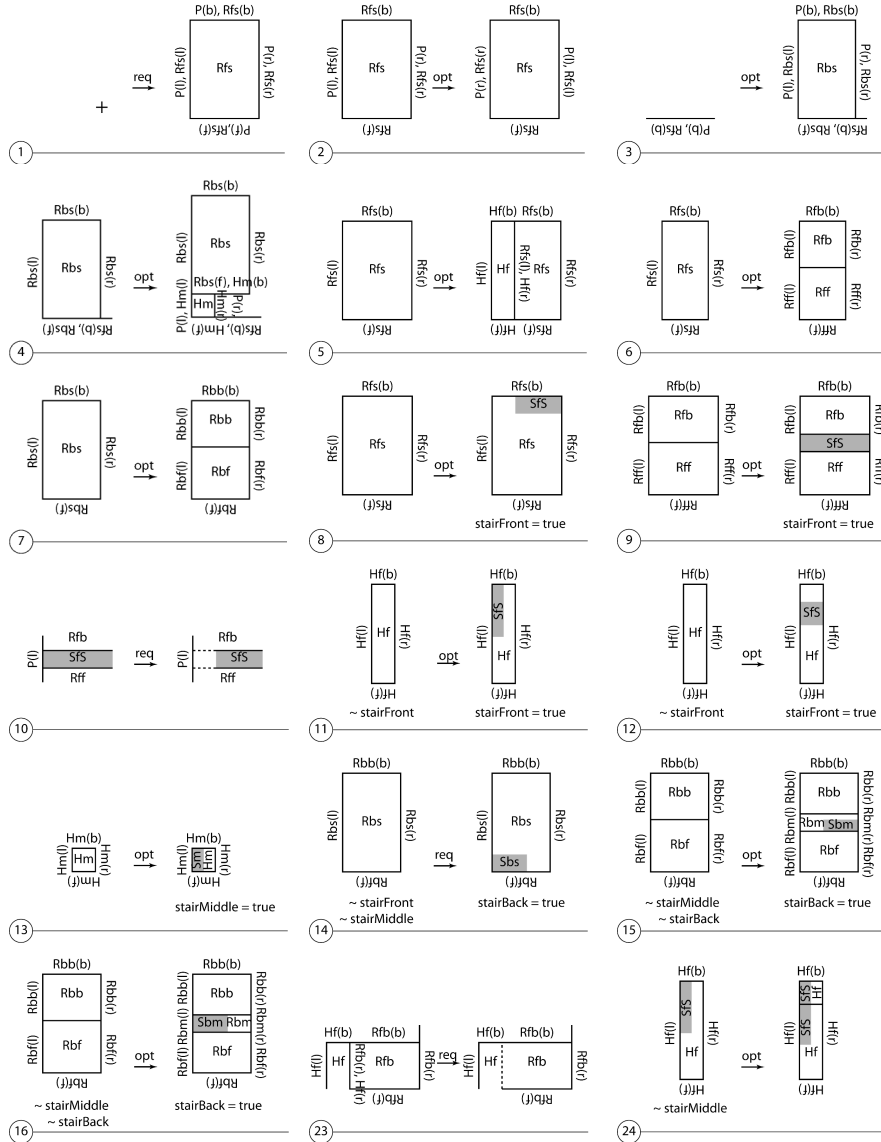


Fig. 7. Rules from four phases of the traditionally defined rowhouse grammar

Phase I: Block Generation

The four rules (1~4): i) generate the front block; ii) mirror the front block; iii) generate the back block; and iv) generate the middle block.

Phase II: Space Generation

The four rules (5~7) generate: i) a hallway in the front block; ii) two spaces within the front block; and iii) two spaces within the back block.

Phase III: Stair Generation

There are 10 rules (8~17): i) generate stair at the back wall of a single-space front block; ii) generate stair between the two spaces of a double-space front block; iii) modify the stair generated by Rule 9 if it runs the entire house width; iv) generate partial width stair in the front hallway; v) generate full-width stair in the front hallway; vi) generate stair in the middle block; vii) generate stair at the front of a single-space back block; viii) generate partial-width stair between the two spaces of a double-space back block; ix) generate full-width stair between the two spaces of a double-space back block; and x) generate accessory stair on the back wall of the back room of a back block.

Phase V: Space Modification

There are two rules, 23 and 24: i) modify the back room of a front block if the front hallway does not adjoin the middle or back block; and ii) generate a service stair behind a partial-width stair in the front hallway.

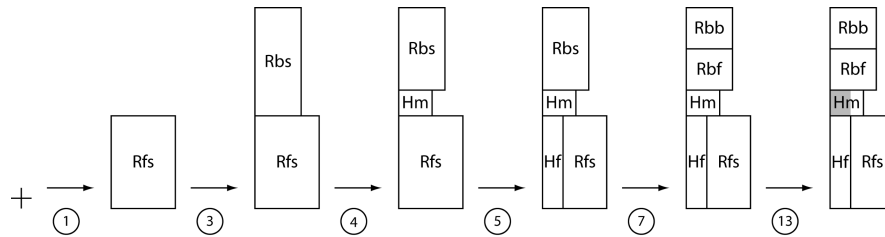


Fig. 8. Derivation of 236 East Montgomery Street

A new version of the rowhouse grammar

In many aspects, the above grammar is not computation-friendly. In particular, the conditions that apply to shape rules are not specified. In order to implement the rowhouse grammar, a new computation-friendly version of the grammar has to be developed. To focus on how to make a traditionally designed shape grammar computation-friendly, we consider only a subset of the corpus, namely, working-class rowhouses, excluding large, luxurious rowhouses, which are considered in the original grammar. Unlike their luxurious counterparts, working-class rowhouses usually have a unique set of staircases on the first floor. Table 2 is a summary of all of the cases under consideration, with the corresponding desired generated layouts obtained by the new version of the grammar. Note that the mechanism of generating fireplace is essentially identical to generating interior doors or staircases. Therefore, we omit the shape rules for generating fireplaces. Moreover, for layout determination, as the feature inputs include windows and exterior doors, rules relating to generating windows and exterior doors are not considered here. Table 3 shows the new shape rules; again, for reasons of space limitation, rules relating to interior doors have been omitted.

Table 2 Baltimore rowhouses under consideration and desired generated layouts

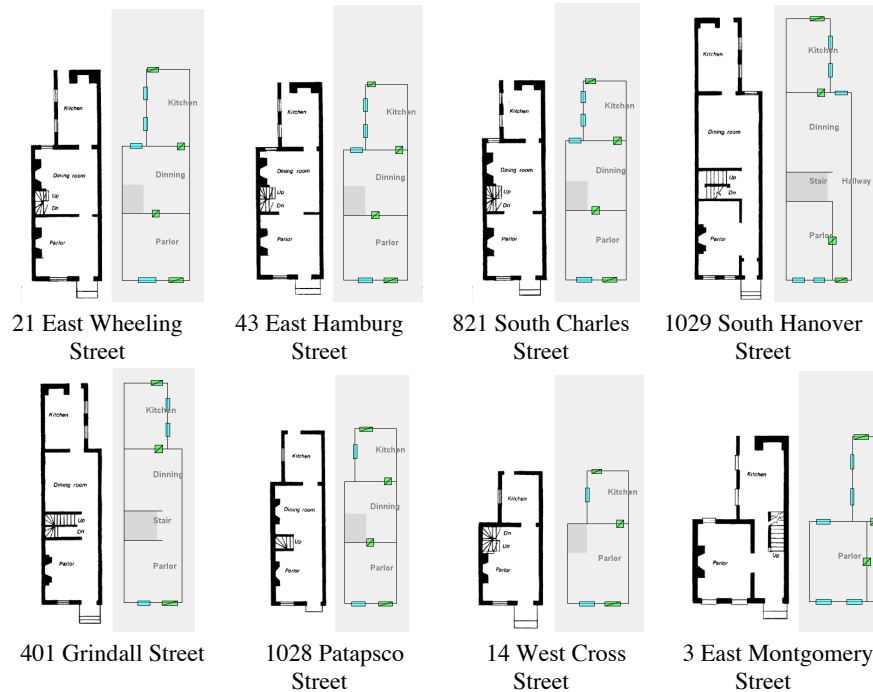
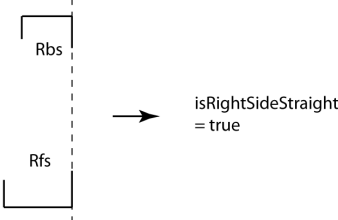
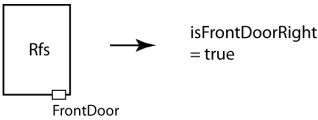
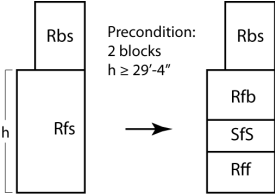
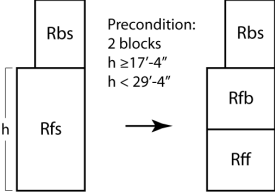
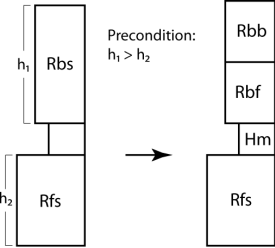
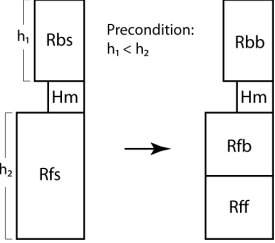


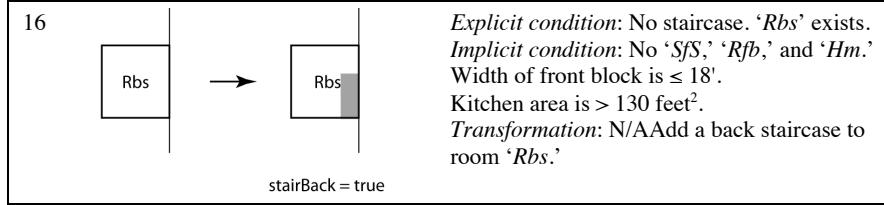


Table 3 New computation-friendly rowhouse shape rules

0		On pre-processing the building feature input, the initial shape is a list of rectangular blocks, which is a decomposition of the footprint, as well as 2D bounds for windows and doors, which have been assigned to the rectangular blocks. Lines are either X- or Y-axis aligned. The line at the bottom corresponds to the front of the building. The column on the left shows two typical examples
1		<i>Precondition:</i> 2 rectangular blocks <i>Transformation:</i> N/A This rule assigns names to the front and back blocks.
2		<i>Precondition:</i> 2 rectangular blocks <i>Transformation:</i> N/A This rule assigns names to the front and back blocks.

3		<p><i>Transformation: N/A</i></p> <p>Blocks of row houses are either left- or right-aligned with a straight line. This information is captured by the attribute, <i>isRightSideStraight</i>. This rule sets the Boolean attribute, <i>isRightSideStraight</i>.</p>
4		<p><i>Transformation: N/A</i></p> <p>This rule sets the Boolean attribute, <i>isFrontDoorRight</i>.</p>
5	 <p>Precondition: 2 blocks $h \geq 29'-4"$</p>	<p><i>Precondition: 2 rectangular blocks with the height (h) of the front block $\geq 29'-4"$.</i></p> <p><i>Transformation: N/A</i></p> <p>This rule divides the front block into two public rooms and a staircase room.</p>
6	 <p>Precondition: 2 blocks $h \geq 17'-4"$ $h < 29'-4"$</p>	<p><i>Precondition: 2 rectangular blocks with the height (h) of the front block between $17'-4"$ and $29'-4"$.</i></p> <p><i>Transformation: N/A</i></p> <p>This rule divides the front block into two equal rooms.</p>
7	 <p>Precondition: $h_1 > h_2$</p>	<p><i>Precondition: 3 rectangular blocks, and the height of the back block (h_1) is greater than the front block (h_2).</i></p> <p><i>Transformation: N/A</i></p> <p>This rule divides the back block into two rooms.</p>
8	 <p>Precondition: $h_1 < h_2$</p>	<p><i>Precondition: 3 rectangular blocks, and the height of the back block (h_1) is smaller than the front block (h_2).</i></p> <p><i>Transformation: N/A</i></p> <p>This rule divides the front block into two rooms.</p>

9		<p><i>Precondition:</i> 'Rfs' exists, and is three-bay (2 windows and 1 door). <i>Transformation:</i> N/A This rule adds to the front block a hall way centered the front door.</p>
10		<p><i>Precondition:</i> 'Rff' exists, and is three-bay (2 windows and 1 door). <i>Transformation:</i> N/A This rule adds to the front block a hall way centered about the front door.</p>
11		<p><i>Explicit condition:</i> No staircase. 'Rfs' exists. <i>Implicit condition:</i> No 'SfS,' 'Rfb' and 'Hm.' Width of front block is $\leq 18'$. Kitchen area is ≤ 130 feet². <i>Transformation:</i> N/A This rule adds a front staircase with dimension 4' \times 6'.</p>
12		<p><i>Precondition:</i> No staircase. 'SfS' exists. <i>Transformation:</i> N/A Add a staircase to room 'SfS.'</p>
13		<p><i>Explicit condition:</i> No staircase. 'Rfb' and 'Rff' exist and are neighbors. <i>Implicit condition:</i> No 'SfS.' Width of front block is $\leq 18'$. <i>Overall condition:</i> stairFront is false. 'Rfb' exist. No 'SfS.' Width of front block is $\leq 18'$. <i>Transformation:</i> N/A Add a staircase to room 'Rfb.'</p>
14		<p><i>Explicit condition:</i> No staircase. 'Hf' exists. <i>Implicit condition:</i> No 'SfS.' Width of front block is $> 18'$. <i>Transformation:</i> N/A Add a front staircase to the hallway at the side next to exterior.</p>
15		<p><i>Explicit condition:</i> No staircase. stairFront is false. 'Hm' exists. <i>Implicit condition:</i> No 'SfS.' Width of front block is $\leq 18'$. No 'Rfb.' <i>Transformation:</i> N/A Add a middle staircase to room 'Hm.'</p>



The new shape grammar comprises five phases: block (mass) generation (rule 1~4), space generation (rule 5~10), stair generation (rule 11~16), space modification (rule 17~20), and interior door generation (rule 21~25); rules 17-25 are not shown due to space limitations.

A significant difference between the new and original shape grammars is that every shape rule of the new shape grammar quantitatively specifies the conditions that apply. For example, this condition can be the number of spaces in terms of blocks (rules 1 and 2), a value in a specific range (rules 5 and 6), and a relationship of two or more values (rules 7 and 8). Some conditions are straightforward. Others require not only reasoning based on common design knowledge, but also certain threshold values, statistically determined. The following illustrates the complexity, using as exemplar, the rules for generating staircases.

Firstly, rules (rule 11~16) are not necessarily exclusive to one another. For example, both rules 11 and 16 can apply to the layouts where no exclusive condition has been specified as to when to apply each rule. As stated previously, we currently only consider working-class row houses, each with a unique staircase on its first floor. Therefore, for each layout, only one of the shape rules for generating staircases applies.

Secondly, if there is a staircase room *SfS*, then rule 12 has to apply. As a result, an implicit condition for Rule 11, 13, 14, 15, and 16 is that the current layout has no staircase room *SfS*.

Rule 14 adds a staircase to a hallway. Obviously, the hallway needs to be wide enough to hold the staircase, hence the width of the front block. From the samples (Figure 9), 18 feet is a good threshold value to distinguish whether or not rule 14 can apply. To ensure the exclusive application of rule 14, an implicit condition for rules 11, 13, 15 and 16 is that the width of the front block is smaller or equal to 18 feet.

If in the left side of rules 11, 13, 15, and 16, there is an *Rfb* room, then rule 13 should be applied to add a staircase there. So, an implicit condition for rule 11, 15 and 16 is that there is no *Rfb* room. If in the left side of rules 11, 15, and 16, there is a middle block *Hm*, then rule 15 should be applied to add a staircase in the middle block. Thus, an implicit condition for rules 11 and 16 is that there is no *Hm* room.

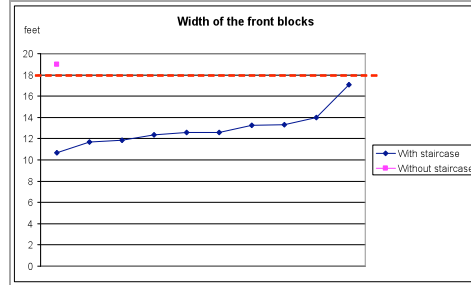


Fig. 9. Quantifying the shape rules generating staircases

It remains to distinguish between rules 11 and 16. The implicit conditions added by rules 12, 13, 14, and 15 can be summarized as: if there are only a *Rfs* room (the front block) and a *Rbs* room (the back block) in the current layout, then rules 11 and 16 can be applied. Rule 16 adds a staircase to an *Rbs* room, which is actually a kitchen. Therefore, the kitchen space has to be large enough to hold a staircase as well as function as a kitchen. From the samples, the average area of kitchens without a staircase is 127.7 feet², and the minimum is 92.8 feet². The area of a staircase is about 26~30 feet². The kitchen area of the case (by using rule 11) is 94.4 feet², and the kitchen area of the case (by using rule 16) is 165.5 feet². The average of these two cases is about 130 feet², which is close to the average of kitchens without staircases. So, 130 feet² is used as the threshold value. As a result, an added condition for rule 16 is that the area of kitchen is greater than 130 feet². An additional condition for rule 11 is that the area of the kitchen is smaller or equal to 130 feet². Table 4 gives a summary of implicit conditions to make rules for generating staircases exclusive.

Table 4 Implicit conditions to make staircase rules exclusive

	Rule 12	Rule 14	Rule 13	Rule 15	Rule 11	Rule 16
Rule 12	With 'SfS'	No 'SfS'	No 'SfS'	No 'SfS'	No 'SfS'	No 'SfS'
Rule 14		Front block width > 18'	Front block width ≤ 18'	Front block width ≤ 18'	Front block width ≤ 18'	Front block width ≤ 18'
Rule 13			With 'Rfb'	No 'Rfb'	No 'Rfb'	No 'Rfb'
Rule 15				With 'Hm'	No 'Hm'	No 'Hm'
Rule 16					Kitchen ≤ 130 ft ²	Kitchen > 130 ft ²

Discussion

The graph-like data structure has the capacity to support the desired interpreter, while, at the same time, necessarily requiring the shape rules to be computation-friendly. To ensure that shape rules from a variety of sources are interpretable, a framework for specifying computation-friendly shape grammars needs to be further developed.

Rule application is a process of traversing an underlying layout tree. This requires an 'undo' mechanism that enables backtracking to a previous partial configuration. Brute force cloning of configurations prior to applying a rule is both computationally expensive and complex, particularly as elements of a layout may need to reference one another. One possibility is to use dancing links [7] to realize this undo functionality; by incorporating features such as windows, doors, and even staircases as nodes linked to the graph units, the entire graph structure can be represented using an eight-way doubly linked list. Another possibility is to design an 'undo' counterpart for each shape rule, which gets applied during backtracking.

The research reported in this paper was funded in part by the US Army Corps of Engineers/CERL, whose support is gratefully acknowledged. The work of Casey Hickerson in developing the first Baltimore rowhouse grammar is also acknowledged.

References

1. Yue K, Hickerson C, Krishnamurti R (2008) Determining the interior layout of buildings describable by shape grammars. *CAADRIA'08*, Chiang Mai, Thailand
2. Flemming U (1987) More than the sum of parts: the grammar of Queen Anne houses. *Environment and Planning B* 14: 323-350
3. Chau HH, Chen X, McKay A, Pennington A (2004) Evaluation of a 3D shape grammar implementation. in JS Gero (ed), *Design Computing and Cognition'04*. Kluwer Academic Publishers, Dordrecht, pp. 357-37
4. Hayward ME (1981) Urban vernacular architecture in nineteenth-century Baltimore. *Winterthur Portfolio* 16(1): 33-63
5. Stiny G (2006) *Shape: Talking about seeing and doing*. MIT Press, Cambridge
6. Fish J (1996) *How sketches work: A cognitive theory for improved system design*. PhD Thesis, Loughborough University
7. Knuth DE (2000) Dancing links. in J Davies, B Roscoe and J Woodcock (eds), *Millennial Perspectives in Computer Science*, Palgrave Macmillan, Basingstoke, pp. 187-214