A SET-BASED SHAPE GRAMMAR INTERPRETER, WITH THOUGHTS ON EMERGENCE

ANDREW I-KANG LI AND LAU MAN KUEN
The Chinese University of Hong Kong, China

Abstract. We present a set-based interpreter, implemented in Multimedia Flash, of a shape grammar for teaching the language of twelfth-century Chinese wood-frame building sections. We discuss the implementation of various aspects of the shape grammar formalism – shape representation, the part relation, and so on – the usefulness of the interpreter, and thoughts on the role of emergence in grammar interpreters.

1. Introduction

Shape grammars have a wide range of capabilities: emergence, parameterization, descriptions, labeling, weights, multiple drawings, and so on. However, interpreters that support all these capabilities have yet to be developed. Existing interpreters[1] have all been restricted, in ways that derive from the intent of the grammars they implement.

Grammars intended as creative design tools (i.e., design or synthetic grammars) generally use emergence and matching under multiple transformations, but not extensive labeling or parameterization. Interpreters that support such grammars – let us call them *synthetic interpreters* – are restricted accordingly. Because of these restrictions, at least in part, Chase (2002, 162) observes that "[t]he user interactions in such tools tend to be rather limited in scope."

Interpreters that support analytic grammars, or *analytic interpreters,* on the other hand, tend to be restricted in the opposite way: they use extensive labeling and parameterization, but not emergence or matching under multiple transformations. In this case, the restrictions seem less noticeable to the user.

Take as an example Flemming's (1987) Queen Anne interpreter which, as he emphasizes, is based on a set representation (Stiny 1982) and so does

1 Gips (1999) provides a summary.

not support emergence. He does not need this capability; indeed, he "hardly missed having a general shape grammar interpreter available" (Flemming 1987, 266).

In addition, he knows that the design space is restricted in another way: "under the selected representation, properties such as the fact that certain edges form a rectangle, are already implied and do not have to be laboriously established whenever a rectangle is called for" (Flemming 1987, 268).

Thus the restrictions of analytic interpreters are not seen as shortcomings, but those of synthetic interpreters are. Chase (2002, 162) suggests that the shortcomings "may be due in part to … a lack of understanding how grammars relate to the design process" and calls for "[f]urther research on interactions in grammar systems."

As a step in this direction, we present an analytic interpreter built for a narrow purpose: to create wood-frame building sections according to the twelfth-century Chinese building manual *Yingzao fashi* [Building standards], by Li Jie (d. 1110). This interpreter is part of a larger scheme for teaching the architectural style of this manual (Li 2003), a scheme which has led us to specific ideas about the task for which the interpreter is a tool, the user's experience in completing the task, and the interpreter as a tool for the user. These ideas have in turn led us to a specific conception of the interaction between the user and the interpreter.

We find this interpreter to be largely satisfactory for its purpose, although it also exhibits a telling shortcoming. By considering this example, we derive some lessons about the relation between users and interpreters generally.

2. Hypothesizing the language of sections

The *Yingzao fashi* is a prescriptive guide for building construction. Li's approach is in general not enumerative but generative, as noted by the architectural historian Liang Sicheng (1984), who called it "a grammar book." By contrast, what Li has to say about building sections is not generative but enumerative: a set of eighteen drawings and written descriptions (see figure 1).

For us, a grammatical understanding of the language of which the corpus is a part presupposes the following conceptual framework.[2] The corpus of sections is a set of empirical observations. The grammar is a hypothesis that makes predictions (creates designs). The predictions are tested (the designs are evaluated for stylistic correctness) and the hypothesis (the grammar) is revised accordingly.

---

2 For a more thorough discussion, see Stiny and Mitchell (1978) and Li (2003; forthcoming).

Figure 1. Drawings and written descriptions of building sections in the *Yingzao fashi.* Shown here are the three sections of six rafters' depth (Liang 1983, 319–320). Not shown are the 15 sections of four, eight, and ten rafters' depth.

To teach within this framework, we devise the following scenario. We provide a grammar; the students test and revise it as necessary. We develop the grammar with the expectation that it should generate all and more than the sections in the language. Then the students' task of evaluating and revising designs is to see and eliminate rather than to imagine and add. We expect further that students can eliminate designs by changing the constraints on schema application, not by changing the schemata themselves.

The grammar is a parametric set grammar with descriptions.[3] Each initial design[4] consists of a shape – a diagrammatic section of 6 rafters' depth – and two descriptions (one Chinese, one English). The initial shape consists in turn of a ground line, two columns (front and back), purlin placeholders, a vertical axis line, labels, and a symbol. The initial descriptions are *6-jia chuan wu, ∅, yong 2 zhu,* and *6-rafter building, ∅, with 2 columns* (see figure 2).

The grammar has four stages. In the first stage (schemata A1–A21), building components, such as columns or beams, are inserted into both the section and the descriptions. In the second stage (schemata A22–A31), any remaining labels are cleaned up; the descriptions are left unchanged. In the third stage (schemata B1–B17), building components (beams, rafters, etc.) are inserted as necessary to complete the section. However, these components are not specified in the descriptions, which therefore remain unchanged. In the fourth stage (schemata B44–B48), the descriptions are reduced to a standard form.

These four stages are not equally relevant to the user, whose task, we recall, is to create and evaluate designs. Only the first stage is relevant, because it is there that the user chooses the features that distinguish the design within the language, that is, the salient features. It is nondeterministic and requires the participation of the user.

---

3 For more on descriptions, see Stiny (1981). For more on the algorithm, see Li (2001).

4 We use the term *design* following Stiny's (1990, 97) definition: "an element in an *n*-ary relation among drawings, other kinds of descriptions, and correlative devices as needed." In our case the design consists of a shape (the section) and two descriptions.

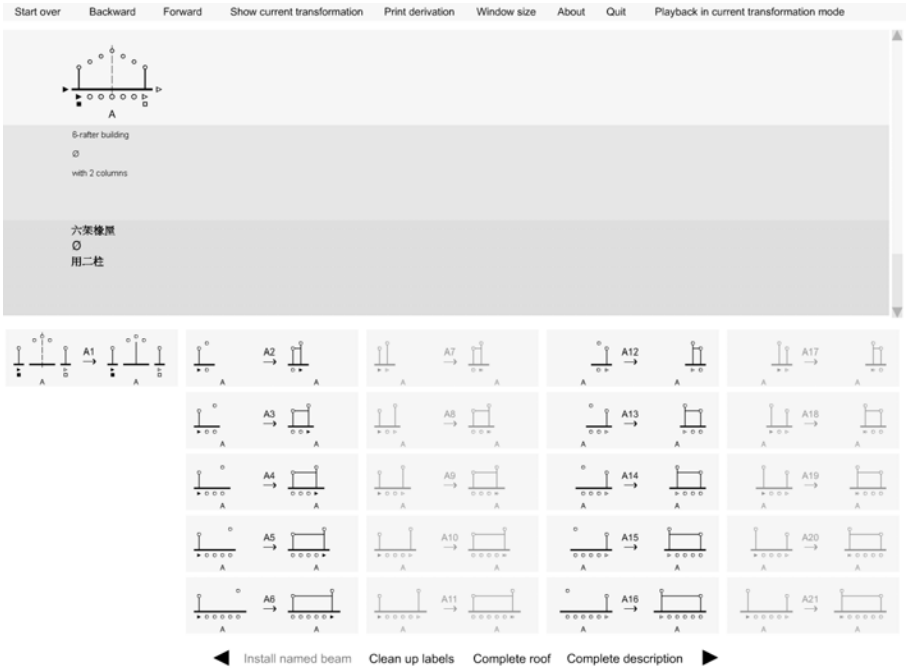

Figure 2. The opening screen of the interpreter. The initial shape and descriptions are above, in derivation view. The schemata (A1–A21) of stage 1 are below. Those schemata that can be applied to the initial shape are shown undimmed.

The other stages do not contribute any salient features. They either do grammatical housekeeping chores (stage 2), complete the section (stage 3), or complete the descriptions (stage 4). They are deterministic and of little interest to the user in the context we have described.

Thus we wish the user to pay attention to the schemata in stage 1 and to be relieved of involvement in stages 2, 3, and 4. To help make this happen is to us one of the main purposes of the interpreter.

3. The interface

3.1. REQUIREMENTS

Our situation contains the three entities of Chase's (2002) model: developer (teacher), users (students[5]), and interpreter. As already mentioned, we have developed the interpreter[6] to create a specific pedagogical experience for the students; this is as follows.

5 We use *students* and *users* interchangeably.
6 An earlier version of the interpreter was reported in Li (2002).

The student makes design decisions (applies rules) to explore the language of designs. Thus the interpreter should help the student create designs easily and assimilate the algorithm.[7] More generally, the interpreter should provide all the information that the user needs to make his decisions. In terms of the derivation of a design, it should tell the user where he has been, where he is, and where he can go. It should hide everything else.

In terms of revising the grammar, we expect that it should suffice for the user to articulate constraints verbally (e.g., *do not apply the same schema twice in succession*), and so do not provide for user modification. Given that in stage A the interpreter provides 367 schema sequences that create only 32 distinct sections, we expect that most modifications should involve filtering out inappropriate sequences, not adding schemata.

3.2. CONCEPTUALIZING THE INTERFACE

To discuss the interface, it is helpful to use some technical apparatus. Consider a design $D = \langle C, d \rangle$, where C is the shape and d is the description. Recall that the next design $\langle C_{(i+1)}, d_{(i+1)} \rangle$ is derived from the current design $\langle C_i, d_i \rangle$ in the following way. For the shapes,

$$\text{if } g(t(A)) \leq C_i, \text{ then } C_{(i+1)} = [C_i - g(t(A))] + g(t(B)),^{8} \tag{1}$$

where A and B are respectively the left and right shapes in the schema $A \rightarrow B$, t is a appropriate transformation, and g is an appropriate parametric assignment. For the descriptions,

$$d_{(i+1)} = f(d_i), \tag{2}$$

where f is a description function associated with the schema $A \rightarrow B$.

We propose that the interpreter should support the following scheme of user interaction.

1 The interpreter shows the current design $\langle C_i, d_i \rangle$.

2 The interpreter shows which schemata can be applied to the current design $\langle C_i, d_i \rangle$ under an appropriate transformation and parametric assignment. In other words, show those schemata $A \rightarrow B$ with associated descriptions f for which $g(t(A)) \leq C_i$ under appropriate values of t and g. In stages 1 and 2, the search is simplified to $g(A) \leq C_i$, because there is not more than one appropriate value of t.

---

7 In the previous version, users did in fact have to execute stages 2 and 3 manually. (Stage 4 had not been implemented.) It was immediately obvious that this distracted users, and we added the capability of automatically deriving the deterministic sequences.

8 As José Pinto Duarte first pointed out, this interpreter actually implements $g(t(A))$ and $g(t(B))$, not Stiny's formulation of $t(g(A))$ and $t(g(B))$. There seems to be no difference in outcome.

3 The interpreter shows the outcome $C_{(i+1)}$ of the schema application, where $C_{(i+1)} = [C_i - g(t(A))] + g(t(B))$.

4 The user chooses a schema.

5 The interpreter applies the schema and updates the current design.

In addition, the interpreter should record the derivation $\langle\langle C_0, d_0\rangle, \langle C_1, d_1\rangle, \ldots, \langle C_n, d_n\rangle\rangle$ and allow the user to undo schema applications back to the initial design $\langle C_0, d_0\rangle$.

3.3. DESCRIPTION

The screen has two halves (see figure 2). The upper half is devoted to the current design. The user can toggle between two views: the current design and the next design in a large size (current transformation view; see figure 3), or the whole derivation in a small size in a scrollable window (derivation view; see figure 2). Both the Chinese and the English descriptions are shown. The derivation can be printed.

The lower half of the screen contains the schemata. These are divided over four "pages" corresponding to the stages mentioned above. Thus the first page, entitled *install named beam,* contains schemata A1–A21; the second, *clean up labels,* A22–A31; the third, *complete roof,* B1–B17; and the fourth, *complete description,* B44–B48. Only the shape schemata are shown; the description functions are shown in pop-up windows (see figure 3).

Schemata that cannot be applied to the current shape are dimmed. Those that can be applied are undimmed. The schemata in stages 1 and 2 can be applied under not more than one transformation and one assignment. Thus one state – undimmed – covers all possible applications.

In stage 3, by contrast, some schemata can be applied under two reflections and one assignment (see figures 4 and 5). Applicable schemata are shown under one of the two reflections; the user can toggle between them by clicking the *mirror* button. As a further help to the user, the schemata are also shown under reflection. In stage 4, there are no shape schemata, only description functions, so the question of transformations and assignments is moot.

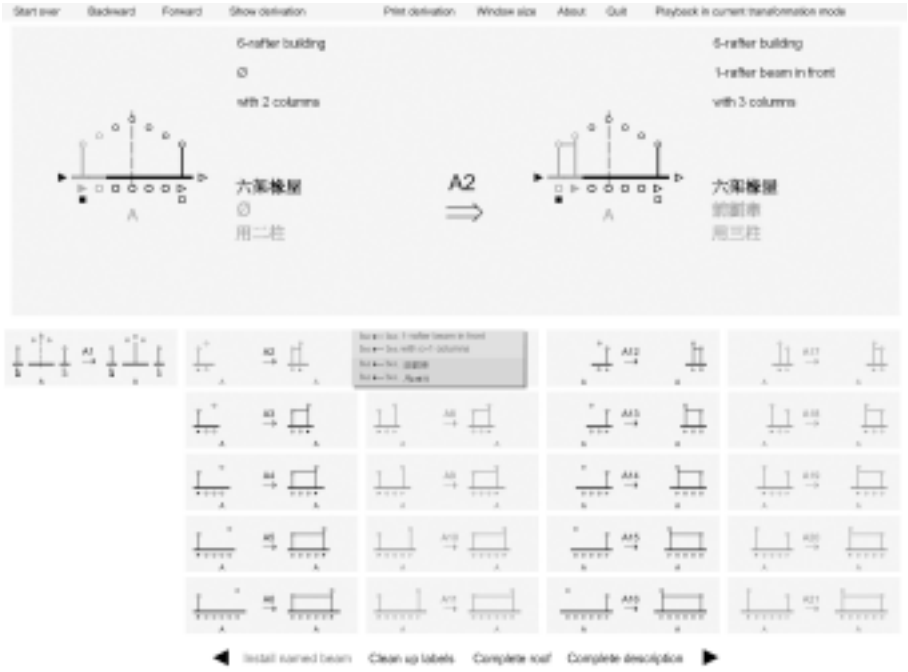

Figure 3. Previewing the next design. The user has rolled over schema A2. The current and next designs are shown in current transformation view. The description functions are shown in a pop-up window. The elements to be changed in the shape and the descriptions are highlighted.

When the user rolls the mouse over an applicable schema $A \rightarrow B$, the next design is previewed in the upper half of the screen, and the associated description function appears in a pop-up window (see figure 3). Several objects are highlighted in red:

- The schema's left and right shapes A and B (in stage 3, the left and right shapes under transformation $t(A)$ and $t(B)$, where t is toggled);
- The corresponding subshapes $g(t(A))$ and $g(t(B))$ in the current and next shapes C_i and $C_{(i+1)}$;
- Each description function f; and
- Those parts of the current and next descriptions d_i and $d_{(i+1)}$ that are manipulated by the function f.

Thus the user can easily survey the possible next designs. He can then apply one of the applicable schemata or back up to the previous design and resurvey the possibilities.

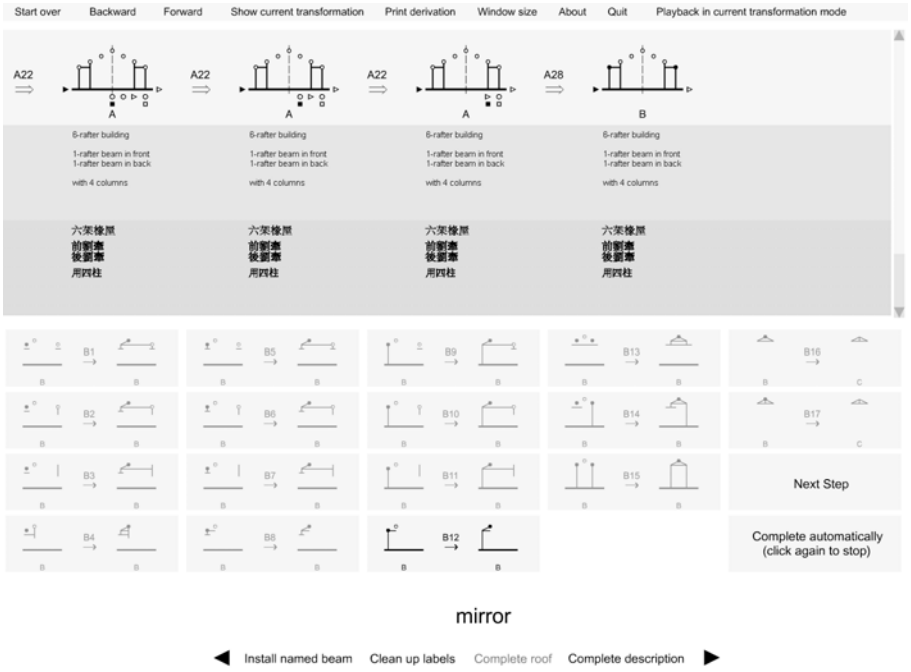

Figure 4. The schemata of stage 3 shown under the first of two reflections.

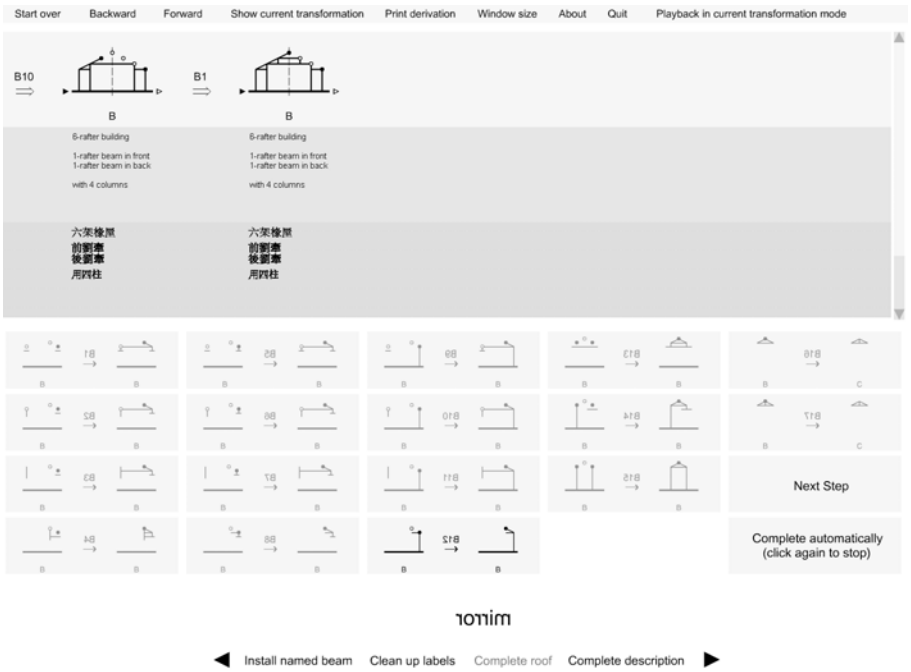

Figure 5. The schemata of stage 3 shown under the second of two reflections.

When the user clicks on an applicable schema, he applies it under the available transformation and assignment. The next design becomes the current design, and the new set of applicable schemata are shown.

The user can generate a design by applying each schema manually from beginning to end but, as we have mentioned above, such close engagement in the deterministic stages is distracting. Thus in stages 2, 3, and 4, the user can click *complete automatically,* and the interpreter executes that stage of the derivation.

4. The implementation

The interpreter is implemented in Macromedia Flash, because Flash is relatively easy to use and allows extensive control over the interface. In addition, the projectors it creates are easy to distribute because they do not need player software.

We will discuss here, not the mechanics of the supporting capabilities – recording the derivation, undoing schema applications, etc. – but the implementation of the shape grammar formalism – shape representation, the part relation, and so on. As has already been mentioned, emergence is not required. Thus the interpreter uses a set representation.

In the discussion below, we follow the scheme of interaction given above. However, the last step, applying a schema and its description function, involves only housekeeping and no shape grammar representation; we omit it.

4.1. SHOWING THE CURRENT DESIGN

A design is represented as a table of values in the following way. It consists of a shape and two descriptions, and is a finite set of discrete elements. Each element has a finite number of attributes, and each attribute has a finite number of values. To display a particular design is simply to display its elements with the appropriate value of each attribute.

The shape is made up of elements that are either parts of the section (e.g., the ground line or a column) or control devices (labels and symbols). Each element has a fixed position and the following attributes:

- Intensity: hidden, dimmed, shown, or highlighted;
- Form: circle, square, single triangle, or double triangle (for labels); *A, B,* or *C* (symbols only); and
- Fill: hollow or solid (labels only).

The English description consists of text elements (letters, numerals, and other symbols). The Chinese description consists of images of Chinese characters, rather than characters as text elements, which Flash does not support.

Below is the code for the `Shape` function, which instantiates a design. Some parts of the section are represented as arrays, where the index indicates the part's position between the front and the back of the building. For example, there are 7 possible column locations `i`, $0 \leq i \leq 6$ from front to back. A given column is represented as `column_var[i]`, where `i` is its location. As we will see, this representation enables the parameterization.

```
function Shape() {                    // creates both shape and
                                      // descriptions
  this.num_rafter = 6;                // the depth of the building
                                      // in rafters
                                      // other values are 4, 8,
                                      // and 10
  this.purlin_front = 0;              // locates the front purlin
  this.purlin_mid = this.num_rafter / 2;
                                      // locates the ridge purlin
  this.purlin_back = this.num_rafter;
                                      // locates the back purlin
  this.purlin_var = new Array();
  this.column_var = new Array();
  this.beam_var = new Array();
  this.roof_var = new Array();
  this.control_var = new Array();
  this.controlclean_var = new Array();
  this.baseline_var = new Array();
  for (i = 0; i <= 6; i++) {          // 6 = this.num_rafter
    this.purlin_var[i] = "circ_hol_show";
    this.column_var[i] = "hide";
    this.control_var[i] = "circ_hol_show";
    this.controlclean_var[i] = "hide";
    this.baseline[i] = "show";
  }
  for (i = 0; i <= 5; i++) {          // 5 = this.num_rafter - 1
    this.beam_var[i] = "hide";
    this.roof_var[i] = "hide";
  }
  this.vertaxis_var = "show";
  this.controlfront_var = "show";
  this.controlback_var = "show";
  this.stage_var = "A_show";
  this.column_var[this.purlin_front] = "show";
  this.column_var[this.purlin_back] = "show";
  this.control_var[this.purlin_front] = "tri_sol_show";
  this.control_var[this.purlin_back] = "tri_hol_show";
  this.controlclean_var[this.purlin_front] = "sq_sol_show";
  this.controlclean_var[this.purlin_back] = "sq_hol_show";
  this.c = 2;
  this.be1 = "6-rafter building";
  this.be2 = "∅";
  this.be3 = "with " + String (this.c) + " columns";
  this.bc1 = "six_jia_chuan_wu";
  this.bc2 = "nil";
  this.bc3 = "yong_" + String (this.c) + "_zhu";
}
shape_current = new Shape();
shape_new = new Shape();
```

4.2. CHECKING THE APPLICABILITY OF SCHEMA A2

Recall that the schemata that can be applied to the current design are shown undimmed. How the interpreter determines the applicability of each schema is best seen by looking at two sample schemata: one with a single transformation, a single assignment, and descriptions; and the other with two transformations, a single assignment, and no descriptions.

The first sample schema A2 inserts a one-rafter-long beam and a column into the current shape and the descriptions. It can be applied to a current shape under a maximum of one transformation and one assignment. Thus the interpreter does not need to determine the transformation; it has only to test the possible assignments. The code is shown below.

```
if (check (_root.shape_current) <> -1) {
  rule_left = new _root.Shape();
  rule_right = new _root.Shape();
  init_rule (rule_left, rule_right, check (_root.shape_current));
  gotoAndPlay ("show");
}
stop();
```

There are two functions here: `check` and `init_rule`. The first, `check`, takes the current shape `shape` as its argument and returns the parametric assignment `i` under which schema A2 can be applied to `shape`; if there is no such assignment, it returns –1. It does this by examining each position `i` in the front half of the building, $0 \leq$ `i` $\leq$ `purlin_mid`. For each value of `i`, it checks whether the value of each attribute of each element in the left shape of the schema matches the value of the corresponding attribute in the current shape, that is, whether $g(A) \leq C_i$. In other words, the part relation $\leq$ is implemented as matching pairs of values; the transformation t is moot; and the assignment g is the index `i` of an array.

```
function check (shape) {
  for (i = shape.purlin_front; i < shape.purlin_mid; i++) {
    if (shape.column_var[i] == "show" and
      shape.column_var[i + 1] == "hide" and
      shape.purlin_var[i] == "circ_hol_show" and
      shape.purlin_var[i + 1] == "circ_hol_show" and
      shape.control_var[i + 1] == "circ_hol_show" and
      shape.control_var[i] == "tri_sol_show") {
      return i;
    }
  }
  return -1;
}
```

The other function `init_rule` sets values to be used if the schema is applied. It takes as arguments the shapes `left` and `right` and the assignment `t` returned by `check`. It sets the value of each attribute of each element in the left shape of the schema to `na`; it sets all other elements to

none. This set of na values is in effect the "inverse" of the left shape of the schema. As we will see below, it will be added, not subtracted, if the next shape is calculated. In other words, init_rule prepares $-g(A)$ (as it were) and $g(B)$ to be used if the next shape $C_{(i+1)} = [C_i + (-g(A))] + g(B)$[9] needs to be calculated.

```
function init_rule (left, right, t) {
  for (i = left.purlin_front; i <= left.purlin_back; i++) {
    left.purlin_var[i] = "none";
    left.column_var[i] = "none";
    if (i < left.purlin_back) {
      left.beam_var[i] = "none";
    }
    left.roof_var[i] = "none";
    left.controL_var[i] = "none";
    left.controlclean_var[i] = "none";
    left.baseline_var[i] = "none";
//
    right.purlin_var[i] = "none";
    right.column_var[i] = "none";
    if (i < right.purlin_back) {
      right.beam_var[i] = "none";
    }
    right.roof_var[i] = "none";
    right.controL_var[i] = "none";
    right.controlclean_var[i] = "none";
    right.baseline_var[i] = "none";
  }
  left.vertaxis_var = "none";
  left.controlfront_var = "none";
  left.controlback_var = "none";
  left.stage_var = "none";
//
  right.vertaxis_var = "none";
  right.controlfront_var = "none";
  right.controlback_var = "none";
  right.stage_var = "none";
//
  left.column_var[t] = "na";
  left.purlin_var[t] = "na";
  left.purlin_var[t + 1] = "na";
  left.control_var[t] = "na";
  left.control_var[t + 1] = "na";
  left.stage_var = "na";
//
  right.purlin_var[t] = "circ_hol_show";
  right.purlin_var[t + 1] = "circ_hol_show";
  if ((t + 1) == right.purlin_mid) {
    right.vertaxis_var = "hide";
  }
  right.column_var[t] = "show";
  right.column_var[t + 1] = "show";
  right.beam_var[t] = "show1";
  right.control_var[t] = "circ_hol_show";
```

9 This approach offers the slight advantage of requiring only an addition function, rather than both addition and subtraction functions. However, it is inconsistent with the formal definition of rule application and so is not ideally clear.

```
  right.control_var[t + 1] = "tri_sol_show";
  right.stage_var = "A_show";
}
```

4.3. CHECKING THE APPLICABILITY OF SCHEMA B12

The second sample schema B12 differs from A2 in having two reflections under which it may be applied to the current shape. It does two checks, one on each reflection, as seen in the code below.

```
if (/:mirror == 0) {
  gotoAndStop ("check");
  if (check (_root.shape_current) <> -1) {
    gotoAndPlay ("show");
  }
} else {
  gotoAndStop ("check_reflect");
  if (check_reflect (_root.shape_current) <> -1) {
    gotoAndPlay ("show_reflect");
  }
}
stop();
```

`Check` and `check_reflect` both work like `check` for A2 above. They return the assignment `i` under which the schema B12 can be applied to a shape (or, if there is no such assignment, –1). The difference is that `check` searches from the front to the middle of the building, while `check_reflect` searches from the back towards the middle of the building. The left shape A is in effect reflected.

```
function check (shape) {
  for (i = shape.purlin_front; i <= (shape.purlin_mid-1); i++) {
                                        // from the front to the
                                        // middle
    if (shape.purlin_var[i] == "circ_sol_show" and
      shape.purlin_var[i + 1] <> "hide" and
                                        // the "next" position is
                                        // towards the back
      shape.column_var[i] == "show" and
      shape.beam_var[i].indexOf ("show") <> -1 and
      shape.stage_var == "B_show") {
        return i;
    }
  }
  return -1;
}
//
function check_reflect (shape) {
  for (i = shape.purlin_back; i >= (shape.purlin_mid + 1); i--) {
                                        // from the back to the
                                        // middle
    if (shape.purlin_var[i] == "circ_sol_show" and
      shape.purlin_var[i - 1] <> "hide" and
                                        // the "next" position is
                                        // towards the front
      shape.column_var[i] == "show" and
      shape.beam_var[i - 1].indexOf ("show") <> -1 and
```

```
      shape.stage_var == "B_show") {
        return i;
    }
  }
  return -1;
}
```

4.4. PREVIEWING THE NEXT DESIGN

When the user rolls over an applicable (undimmed) schema, the next design is created and displayed. The rollover code for schema A2 is shown below.

```
on (release, rollOver) {
  // init begin
  _root.current_rule = "A2";
  if (/:display_mode == "normal") {
    _root.current_stage = "normal/initshape";
    _root.stage_preview = "normal/preview";
    _root.rule_object = "normal/arrow";
  } else if (/:display_mode == "overview") {
    _root.current_stage = "overview/derivation/stage" + String
(_root.history.stage_current);
    _root.stage_preview = "overview/derivation/stage" + String
(_root.history.stage_current + 1);
    _root.rule_object = "overview/derivation/arrow" + String
(_root.history.stage_current + 1);
  }
  _root.show_rule (_root.current_rule, _root.rule_object);
  // init end
  target = check (_root.shape_current);
  show_match (_root.shape_current, target);
  _root.shape_new = apply (_root.shape_current);
  _root.preview_change (_root.shape_new, _root.stage_preview);
  show_change (_root.shape_new, target);
  /:change = 1;
  gotoAndStop ("highlight");
}
```

The important functions here are `check`, `show_match`, `apply`, `preview_change`, and `show_change`. `Check` has already been seen above.

`Show_match` highlights $g(t(A))$ as a part of C_i. It accepts as its arguments the current shape `shape_source` and the assignment `target`. It makes a copy `shape` of the current shape and, for each attribute of each element in $g(t(A))$, overwrites the value as `highlight`.

```
function show_match (shape_source, target) {
  shape = new Object();
  _root.Object_duplicate (shape_source, shape);
  //
  shape.purlin_var[target] = "circ_hol_highlight";
  shape.purlin_var[target + 1] = "circ_hol_highlight";
  shape.column_var[target] = "highlight";
  shape.control_var[target] = "tri_sol_highlight";
  shape.control_var[target + 1] = "circ_hol_highlight";
  shape.baseline_var[target] = "highlight";
  shape.baseline_var[target + 1] = "highlight";
```

```
  shape.stage_var = "A_highlight";
  shape.be1 = "";
  shape.bc1 = "";
  //
  _root.show_matching (shape, _root.current_stage);
  delete shape;
}
```

In addition, `show_match` calls `show_matching`, which activates the display. `Show_matching` takes as its arguments the next shape `shape` and the variable `current_stage`, which specifies the view (current transformation or derivation). It examines the value of each attribute of each element; if the value is `highlight`, then the element is highlighted.

After calling `show_match`, the rollover code calls `apply`, which takes as its argument the current shape `shape_source` and returns the next shape `shape`. It does this by creating a copy of the current shape and transforming it through addition and subtraction. The code for `apply` is shown below.

```
function apply (shape_source) {
  shape = new _root.Shape();
  _root.Object_duplicate (shape_source, shape);
  //
  shape = Add (Subtract (shape, rule_left), rule_right);
  shape.c++;
  if (shape.be2 <> "∅") {
    shape.be2 += chr(13) + "1-rafter beam in front";
    shape.bc2 += chr(13) + "qian_zhaqian";
  } else {
    shape.be2 = "1-rafter beam in front";
    shape.bc2 = "qian_zhaqian";
  }
  shape.be3 = "with " + String (shape.c) + " columns";
  shape.bc3 = "yong_" + String (shape.c) + "_zhu";
  return shape;
}
```

The important functions in `apply` are `Add` and `Subtract`, which we will return to shortly. The next descriptions are created by simply inserting new text strings into the current descriptions.

`Preview_change` takes as its arguments the next shape `shape` and the variable `shape_name`, which specifies the view (current transformation or derivation), and displays the next shape in the appropriate view.

`Show_change` is similar to `show_match`: it highlights $g(t(B))$ as a part of $C_{(i+1)}$.

4.5. ADDITION AND SUBTRACTION

As mentioned above, `rule_left` is created by `init_rule` and intuitively is the "inverse" of $g(A)$, that is, $-g(A)$. `Subtract` and `Add` are identical. `Add` takes as its arguments two shapes `shape_A` and `shape_B` and returns their sum `shape`. It examines each attribute of each element of `shape`, and

overwrites that value onto the corresponding value in `shape_B`, unless that value is `none`. The code for `Add` is shown here.

```
function Add (shape_A, shape_B) {
  shape = new _root.Shape();
  _root.Object_duplicate (shape_A, shape);
  //
  for (i = shape.purlin_front; i <= shape.purlin_back; i++) {
    if (shape_B.purlin_var[i] <> "none") {
      shape.purlin_var[i] = shape_B.purlin_var[i];
    }
    if (shape_B.column_var[i] <> "none") {
      shape.column_var[i] = shape_B.column_var[i];
    }
    if ((shape_B.beam_var[i] <> "none") and (i < shape_A.purlin_back))
{
      shape.beam_var[i] = shape_B.beam_var[i];
    }
    if (shape_B.roof_var[i] <> "none") {
      shape.roof_var[i] = shape_B.roof_var[i];
    }
    if (shape_B.control_var[i] <> "none") {
      shape.control_var[i] = shape_B.control_var[i];
    }
    if (shape_B.controlclean_var[i] <> "none") {
      shape.controlclean_var[i] = shape_B.controlclean_var[i];
    }
    if (shape_B.baseline_var[i] <> "none") {
      shape.baseline_var[i] = shape_B.baseline_var[i];
    }
  }
  if (shape_B.vertaxis_var <> "none") {
    shape.vertaxis_var = shape_B.vertaxis_var;
  }
  if (shape_B.controlfront_var <> "none") {
    shape.controlfront_var = shape_B.controlfront_var;
  }
  if (shape_B.controlback_var <> "none") {
    shape.controlback_var = shape_B.controlback_var;
  }
  if (shape_B.stage_var <> "none") {
    shape.stage_var = shape_B.stage_var;
  }
  return shape;
}
```

5. Conclusion

We have seen the implementation of a parametric set grammar that supports neither emergence nor matching under multiple transformations. The infelicities of this implementation are evident and many: it is complicated, inefficient, and inconsistent with formal definitions.

On the other hand, we know from classroom experience that it does what we designed it to do, namely to show the current design, check the applicability of a schema, preview the next design, and apply a schema. These four capabilities have produced an interface that has proven to be

virtually self-explanatory. In fact, rather than using shape grammar to explain the interpreter, we now use the interpreter to explain shape grammar.

That a set-based implementation, even a suboptimal one such as this, can accomplish so much suggests that we have not addressed the really difficult issues, chief among which is, according to Chase (2002, 162), "handling the unexpected nature of emergent features." Indeed, it seems clear that the key to easy implementation is simply to avoid emergence altogether.

The question is begged: can emergence be avoided? We believe that it depends on the expected interaction between user and implementation. If the grammar is not expected to be static – if, for example, the user is allowed to revise it – then emergence is indispensable. An infallible set representation is impossible, because it is impossible to foresee all possible revisions.

This is exactly what happened in our classroom experience. Recall that we had expected that students would modify the grammar by increasing the constraints on schema application; we did not expect students to alter other aspects of the grammar. In the event, some students modified the schemata (on paper) in ways that we had not anticipated – for instance, inserting columns in unexpected positions – and that therefore could not be supported by our set representation.

Our implementation fell short, given our premise that the grammar is a hypothesis to be tested and revised. In addition, we had made no provision for students to modify the interpreter by altering existing rules or defining new ones. This is an interface challenge that we thought we had been spared.

But if, on the other hand, the user simply "operates" (and does not modify) the implementation – that is, he uses the grammar as is – then an appropriate set representation will suffice. Thus Flemming (1987) does not miss a general interpreter for two reasons: his set representation is immune from challenge, and his interface does not need to support rule definition by users.

It seems that we can forego emergence with only a narrow category of grammars: those that are fixed or changeable within known limits. For all others, including ours, which we had initially considered analytic, we must be able to implement emergence. This is one research goal.

At the same time, we believe that, between the limitations of a fixed representation and the complexity of emergence, there may be room for creative expedience. It would be worthwhile to investigate further the relation between the user–interpreter interface and the technical characteristics of the implementation.

Acknowledgements

We would like to thank the Chinese University of Hong Kong (CUHK) for a Direct Grant for Research; the Department of Architecture, CUHK, for

special support; Wang Yang for preparing the illustrations; and our students for cheerfully taking the grammar in directions we had not foreseen.

References

Chase, SC: 2002, A model for user interaction in grammar-based design systems, *Automation in construction* **11**: 161–172.

Flemming, U: 1987, The role of shape grammars in the analysis and creation of designs, *in* YE Kalay (ed), *Computability of design,* John Wiley, New York, pp. 245–272.

Gips, J: 1999, Computer implementation of shape grammars, paper read at NSF/MIT Workshop on Shape Computation, at Cambridge, Mass.

Li, AI: 2001, A shape grammar for teaching the architectural style of the *Yingzao fashi,* Ph.D. dissertation, Department of Architecture, Massachusetts Institute of Technology, Cambridge, Mass.

Li, AI: 2002, A prototype interactive simulated shape grammar, *in* K Koszewski and S Wrona (eds), *Design e-ducation: connecting the real and the virtual, Proceedings of the 20th Conference on Education in Computer Aided Architectural Design in Europe,* eCAADe, Warsaw, pp. 314–317.

Li, AI: 2003, The *Yingzao fashi* in the information age, paper read at The Beaux-Arts, Paul-Philippe Cret, and 20th-century architecture in China, at University of Pennsylvania.

Li, AI: forthcoming, Styles, grammars, authors, and users, paper read at Design computing and cognition '04, at Cambridge, Mass.

Liang Sicheng: 1984, Zhongguo jianzhu zhi liangbu "wenfa keben" [The two "grammar books" of Chinese architecture], *Liang Sicheng wenji* [The collected works of Liang Sicheng], Zhongguo jianzhu gongye, Beijing, pp. 357–363.

Stiny, G: 1981, A note on the description of designs, *Environment and planning B: planning & design* **8**: 257–267.

Stiny, G: 1982, Spatial relations and grammars, *Environment and planning B* **9**: 113–114.

Stiny, G: 1990, What is a design?, *Environment and planning B: planning & design* **17**: 97–103.

Stiny, G and Mitchell, WJ: 1978, The Palladian grammar, *Environment and planning B: planning & design* **5**: 5–18.