# Developing a Graphical Definition System for a Spatial Parser Generator

Hiroaki Kameyama, Buntarou Shizuki, and Jiro Tanaka
Department of Computer Science, University of Tsukuba
1-1-1 Tennoudai, Tsukuba, Ibaraki, 305-8573, Japan
{ kame, shizuki, jiro }@iplab.is.tsukuba.ac.jp

**Abstract**

*Spatial parser generators automatically generate a parser of visual languages by providing grammars. The grammar is specified using text. The grammar would be easier to understand if we used a figure to input the grammar. We describe an approach in this paper that graphically defines grammar. Direct manipulation is used to define grammar in our approach, which helps the user understand the meaning. We implement the GIGA system, which enables the user to define the grammar by drawing figures. GIGA displays each element of a rule visually. The user can define the grammar easily and can understand the grammar visually. Moreover, GIGA outputs the defined visual system to a file written in CMG text form. The user can execute the system by inputting the file into Eviss.*

## 1. Introduction

A parser for conventional programming language analyses the input text based on the grammar. Parsing is the process of applying production rules to figure elements. Yacc [7] and Bison [4] generated such a parser automatically from the specification of the grammar.

The visual system, similar to a diagram editor, can handle figures. The input of the visual system includes rectangles, circles, and lines. Spatial parser generators automatically generate a parser of visual languages by providing grammar. Eviss [1] and VIC [5] are examples of such spatial parser generators.

The grammar input is defined in Eviss using text. It is difficult to understand the meaning of the grammar if it is given in textual form. It would be easier to understand if we used figures to input the grammar and edited them directly.

Therefore, we propose an approach that defines grammar using figures, and we implement the GIGA system. The GIGA system enables the user to define the components of the rule by drawing corresponding figures. GIGA infers and displays the constraints based on the positional relationships among the figures. GIGA enables more intuitive and interactive ways for the user to define the grammar.

## 2. Defining Visual Systems

### 2.1. Extended CMG

The spatial parser generator Penguins [2, 3] uses Constraint Multiset Grammars (CMG) [8] to define the grammar of visual systems. CMG consists of a set of terminal symbols (called **components**), a set of non-terminal symbols, a distinctive start symbol, and a set of production rules. The terminal and non-terminal symbols have various **attributes**. Production rules are used to rewrite a multiset of tokens (instances of terminal or non-terminal symbols) for a new symbol. The **constraint** maintains the relationships between the attributes of the tokens. Chok reported on editors, such as flow chats, state transition diagrams, and mathematical expressions [3], as examples of a visual system described with CMG. Analyzing visual languages is not sufficient for visual systems. Actual visual systems must execute statements according to the result of the analysis and redraw the figures, preserving the semantic relationships between the figure elements.

We extended the original CMG to include **action**, which is defined as "a script program executed when the production rule is applied." We can specify arbitrary actions in the Extended CMG [1], such as computing values and rewriting figures.

### 2.2. The Problems in Defining Visual Systems

A spatial parser generator produces parsers of visual languages by providing the grammar. Previous spatial parser generators treated the CMG defined using text. A considerable portion of the specification in Extended CMG includes two-dimensional information, such as the shapes and positional relations of figures. It is difficult to read and understand these rules if we describe them using text. Moreover, the following problems occur.

- **The description of constraints is complicated.** We must input many constraints to define the rule. We also must determine whether these constraints are consistent.

- **The definition of attributes is complicated.**
  The coordinates and size of a figure are used as attribute values. In many cases, a non-terminal symbol inherits the attribute of the component. However, we must define the attributes one by one.
- **It is difficult to predict the appearance of the screen after an action is performed.**
  The action redraws the figure. We cannot anticipate the appearance of the whole figure after an action is performed based on textual rules.

### 2.3. The Proposed Techniques

We took the following approaches to solve the above problems.

- **We define the components by drawing the figures directly.**
  The user inputs the figures as with general draw tools.
- **We define the constraints by arranging the components so that the constraints were satisfied.**
  The system displays the constraints on the screen as the user edits the figures. We use the following techniques for the system to infer unnecessary constraints.
  - The user associates the related figures explicitly, and the system infers the constraints from the figures that are associated.
  - The user chooses the attributes required for the definition of constraints. The system infers the constraints from the attributes of the chosen figures.
- **We define the attribute of a non-terminal symbol by selecting the attribute of the component.**
  The user selects the attributes of the input figure that were inherited.
- **We define the action by editing the figures.**
  The system copies the figures and the user then edits them. The system outputs actions from the operation history.

We can understand the grammar by just looking at the screen when using these techniques.

## 3. Graphical Definition System GIGA

### 3.1. Implementing the GIGA system

We implemented the GIGA system (**G**raphical user **I**nterface for Constraint Multiset **G**rammars with **A**ction. GIGA means cartoon in Japanese), which enables the user to define a rule by drawing the figures. A snapshot of GIGA is provided in Figure 1.

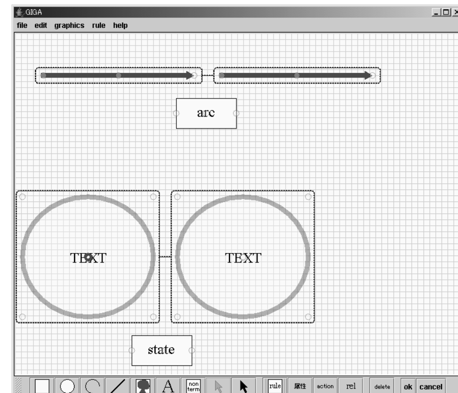


Figure 1. Snapshot of GIGA

The GIGA system is implemented in 100% pure Java language. Therefore, the user is able to use GIGA on any platform. The code size of the system is approximately 5000 lines. GIGA consists of figure-editing modules, rule-generating modules and rule-output modules. The figure-editing modules treat the input figures. Elementally, the figures are implemented as the figure class. The rule-generating modules produce rules by comparing the attribute variables of the figures. We implement each constraint as a constraint class. The rule-output modules translate the grammar into text form and output it to a file.

### 3.2. Defining Components

The user inputs the figures selected from the tool panel to define the components of the rule. It is possible to perform operations such as movement in the figure, modification of size, and deletion, as with general graphic tools. The figure is given a unique identifier automatically at this time.

It is also possible to input a non-terminal symbol as a component. The user first selects the “non-terminal” button in the tool panel. A list of non-terminal symbols already defined is displayed and the user selects the required one.

**Attribute Selection** A figure has various attributes. It is necessary to choose the required attribute when the user defines the rule. The attribute that has the coordinate value (such as the center point of a rectangle) is displayed as a circle in GIGA. The user clicks the circle of the required attribute if he or she needs the attribute. The attributes, such as

color, width, or height, are selected by modifying the value of the respective attribute.

### 3.3. Defining Constraints

The user defines the constraint by editing the figures and indicating the intended constraint. GIGA displays the following constraints and enables the user to easier understand the defined constraint.

- **The coordinates of the two attributes are equal**
  GIGA displays a red circle, which represents the attribute (Figure 2).

- **The x–coordinates of the two attributes are equal**
  GIGA displays a red guideline that connects two attributes (Figure 3). This works in the same way in the case of y–coordinates.

- **The sizes of the two attributes are equal**
  GIGA displays the constraint using an arrow (Figure 4).

- **A figure always exists on the right side of another figure**
  GIGA displays an arrow that connects two attributes.

GIGA infers the constraint interactively. Therefore, if the constraint is not satisfied between the figures, GIGA stops displaying it. Thus, it is easy to identify which type of constraint is defined.

GIGA infers the constraint between the attributes that the user associates to avoid inferring an unnecessary constraint. Association is defined by moving a figure and piling up it on another figure, and it holds even if the figures separate.

### 3.4. Defining Attributes

The user can define the attributes of a non-terminal symbol by clicking on the attribute of the component. The user can also define the other attributes by inputting the arithmetic expression of the attribute into the circle symbol using text.

### 3.5. Defining Actions

GIGA displays the two regions surrounded by the square frame shown in Figure 6 to define the action. GIGA copies the figures of components in the left region to right region. The user edits the copied figures. GIGA infers the action from the difference between the figures. The types of actions we can use in GIGA are “create,” “delete,” and “alter.” GIGA infers each action as follows.

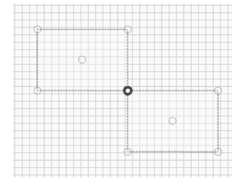

Figure 2. Displays the equal constraint (1)

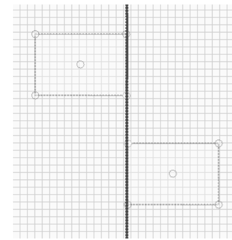

Figure 3. Displays the equal constraint (2)

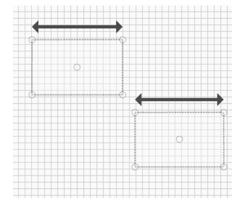

Figure 4. Displays the equal constraint (3)

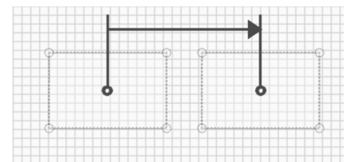

Figure 5. Displays the constraint of the right-and-left relation

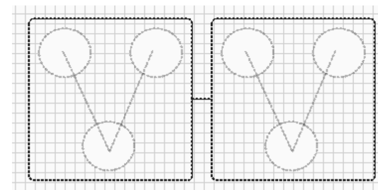

Figure 6. Definition of action

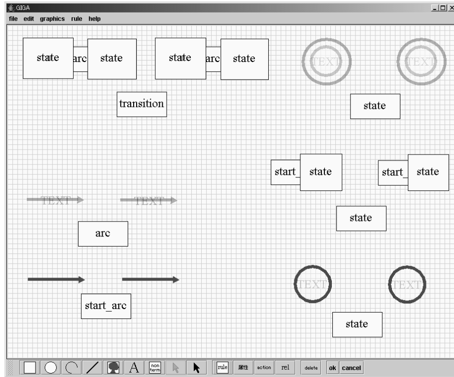


Figure 7. Definition of the State Transition Diagram Editor

- When a figure does not exist in the original region but exists in the copied region, GIGA generates the “create” action and infers the position in the new figures, utilizing the positions of the other figures.
- GIGA generates the “alter” action if the attribute of the figure in the original region is changed in the copied region.
- The user deletes the figures in the copied region to define the “delete” action.

# 4. Executing Visual Systems

## 4.1. An Example of Defining a Visual System

We will use the rule of the state transition diagram editor as an example. The state transition diagram editor is defined by the following six production rules.

1. A non-terminal symbol “arc” consists of a line and text in the center of it.
2. A non-terminal symbol “start arc” consists of a line. The line must be red.
3. A non-terminal symbol “state” consists of a text string and two circles. This rule expresses the final state.
4. A non-terminal symbol “state” consists of a circle and a text string. The circle must be red. This rule expresses the normal state.
5. A non-terminal symbol “state” consists of a circle, a text string, and a start arc. This rule expresses the start state.

```
 1: arc(point start, point end,
 2:   point mid, string text) ::=
 3:   L:line, T:text where(
 4:   L.mid == T.mid
 5: ){
 6:   start = L.start;
 7:   end = L.end;
 8:   mid = L.mid;
 9:   text = T.text;
10: }{}
11:
12: start_arc(point end) ::=
13:   L:line where(
14:   L.color == red
15: ){
16:   end = L.end;
17: }{}
18:
19: state(pont mid, integer radius,
20:   string name, string kind) ::=
21:   C1:circle, C2:circle, T:text where(
22:   C1.mid == C2.mid &&
23:   C1.radius > C2.radius &&
24:   T.mid == C1.mid
25: ){
26:   mid = C1.mid;
27:   radius = C1.radius;
28:   name = T.text;
29:   kind = "final";
30: }{}
31:
32: state(pont mid, integer radius,
33:   string name, string kind) ::=
34:   C:circle,T:text where (
35:   C.color == "red" &&
36:   T.mid == C.mid
37: ){
38:   mid = C.mid;
39:   name = T.text;
40:   kind = "normal";
41: }{}
42:
43: state(point mid, integer radius,
44:   string name, string kind) ::=
45:   A:start_arc, C:circle, T:text where(
46:   C.color == red &&
47:   A.end == C.mid &&
48:   T.mid == C.mid
49: ){
50:   mid = C.mid;
51:   name = T.text;
52:   kind = "start";
53: }{}
54:
55: transition(string start, string label
56:   string end) ::=
57:   A:arc, exist S1:state, S2:state where(
58:   A.start == S1.mid &&
59:   A.end == S2.mid
60: ){
61:   start = S1.name;
62:   label = A.text;
63:   end = S2.name;
64: }{}
```

Figure 8. The specification of state transition diagram with Extended CMG.

6. A non-terminal symbol “transition” consists of an arc and two states. This rule expresses the state transition.

Figure 7 depicts the screen after the definition of the six rules of the state transition diagram editor.

GIGA can translate a defined rule into textual form and output it to a file. Figure 8 illustrates the file output by GIGA.

Lines 1 to 10 define production Rule 1. Lines 1 and 2 present the attributes of the non-terminal symbol “arc.” Line 3 indicates that the node consists of a circle and a text string. Line 4 describes the constraints. It indicates that the center of the text string is on the center of the line. Lines 6 to 9 define the values of the attributes. Line 6 states that “start” of the “arc” is equal to “start” of “L.” Line 7 states that “end” of the “arc” is equal to “end” of “L.” Line 8 states that “mid” of the “arc” is equal to “mid” of “L.” Line 9 comprises a script to substitute the attribute “text” of “T” for the attribute value of the “state.”

### 4.2. Executing the Visual System by Eviss

We use the spatial parser generator Eviss to execute the visual system that was defined using GIGA.

Eviss is a visual system that has a spatial parser generator. Eviss can generate spatial parsers for visual languages by providing the grammar. Eviss can parse the figures spatially if a certain grammar written in text is input. When a production rule is applied, Eviss redraws the figure elements so that the constraints between them always hold. Actions are also executed when the production rule is applied.

GIGA outputs the defined visual system to a file written in the text form of Extended CMG. The user can execute the system by inputting the file into Eviss. The execution of the defined state transition diagram editor is illustrated in Figure 9. The relation of each figure is maintained by the constraint solver in Eviss. If the user moves a figure recognized as “state,” then a figure recognized as “arc” is moved by Eviss so that the “end” of “arc” and the “mid” of “state” will have the same coordinates.

## 5. Discussion

We have defined various visual systems as follows using Extended CMG [1, 5].

- The visual system (Calculation Tree) for editing a calculation tree and executing it.
- The visual system (Stack) for editing a stack as a figure and executing it.
- The visual shell (VSH) for visualizing a shell with the function of a pipe.
- The interface builder (GUI) for editing GUI parts, such as a scroll bar and a button.
- The subset of visual system HI-VISUAL for specifying operation by piling up two or more icons that express the file and the command.
- The subset of visual system VISPATCH [6], which defines the rule using figures and rewrites figures by operations of the user, such as a mouse event.

We indicate the number of rules, constraints, attributes, and actions in these visual systems described by GIGA in Table 1.

A large portion of the specification of Extended CMG is two-dimensional information, such as the form and the positional relation of the figures listed in Table 1. Eighty percent of the formula of constraints is the constraint that uses the positional attribute of a figure. Fifty percent of the formula of actions is the rewriting rule. Eighty percent of the attributes of a non-terminal symbol is an attribute of a position.

This demonstrates that the graphical definition technique of GIGA is effective in many situations.

## 6. Related Works

Penguins [2, 3] is a spatial parser generator that produces an editor that supports the creation, manipulation, and parsing of diagrams from grammatical specifications. Writing textually is one way of inputting CMG in Penguins. If the type of a component is changed, the user must rewrite most of the rule.

TRIP3 [10, 9] is a system that processes the figure by the constraint. A TRIP3 user describes the relationship between the figures and the application data. TRIP3 can draw figures from the text and can create texts from the figures using the relationship. However, the user cannot define a new non-terminal symbol.

## 7. Conclusion

We proposed an approach in this paper that defines grammar using figures and we implemented the GIGA system. The GIGA system enables a user to define the components of a rule by drawing corresponding figures.

GIGA infers the constraints from the positional relationships among the figures edited by the user and displays them. GIGA infers the actions from the difference between the screen before an action is performed and the screen after an action is performed. GIGA outputs the defined visual system to a file written in the text form of Extended CMG. The user can execute the system by inputting the file into Eviss.

A user can define grammar easily using the GIGA system and can understand the grammar visually.

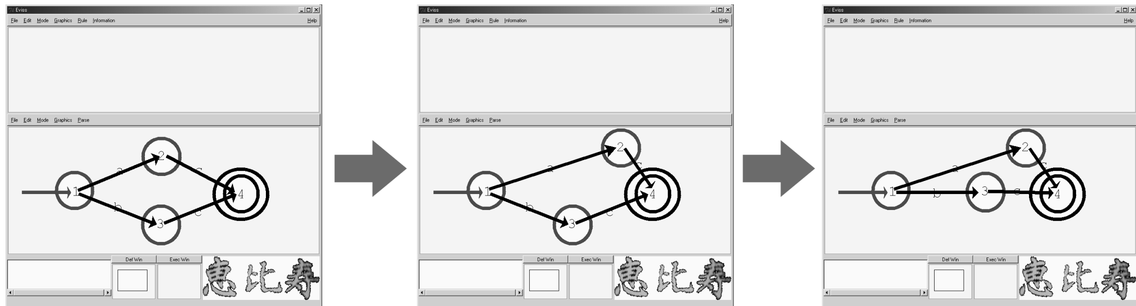

Figure 9. Execution of State Transition Diagram Editor

Table 1. The number of rules that GIGA can define

| | Num. of rules | Number of constrains | Number of attributes | Number of actions |
|---|---|---|---|---|
| Calculation Tree | 2 | 7/7 (100%) | 6/8 (75%) | 5/5 (100%) |
| Stack | 4 | 5/5 (100%) | 12/18 (67%) | 2/6 (34%) |
| VSH | 11 | 35/55 (64%) | 10/29 (34%) | 0/1 (0%) |
| GUI | 14 | 72/85 (85%) | 58/100 (58%) | 1/1 (100%) |
| HI-VISUAL | 15 | 23/43 (65%) | 32/50 (64%) | 4/14 (29%) |
| VISPATCH | 24 | 112/134 (84%) | 65/100 (65%) | 3/5 (60%) |

## References


[1] Akihiro Baba and Jiro Tanaka. Eviss: A visual system having a spatial parser generator. In *Proceedings of Asia Pacific Computer Human Interaction*, pages 158–164, July 1998.

[2] Sitt Sen Chok and Kim Marriott. Automatic construction of user interfaces from constraint multiset grammars. In *Proceedings of IEEE Symposium on Visual Language*, pages 242–249, 1995.

[3] Sitt Sen Chok and Kim Marriott. Automatic construction of intelligent diagram editors. In *Proceedings of ACM Symposium on User Interface Software and Technology*, pages 185–194, 1998.

[4] Robert Corbett and Richard Stallman. Bison: Gnu parser generator. Texinfo documentation, Free Software Foundation, Cambridge, Mass, 1991.

[5] Kenichiro Fujiyama, Kazuhisa Iizuka, and Jiro Tanaka. Vic:cmg input system using example figures. In *Proceedings of International Symposium on Future Software Technology (ISFST' 99), Nanjing, China*, pages 67–72, October 1999.

[6] Yasunori Harada, Kenji Miyamoto, and Rikio Onai. Vispatch:graphical rule-based language controlled by user event. In *Proceedings of the 1997 IEEE Symposium on Visual Languages*, 1997.

[7] Steven C. Johnson. Yacc: Yet another compiler compiler. In *UNIX Programmer's Manual*, volume 2B, pages 353–387. Holt, Rinehart, and Winston, 1979.

[8] Kim Marriott. Constraint multiset grammars. In *Proceedings of the IEEE Symposium on Visual Languages*, pages 118–125, 1994.

[9] Shin Takahashi, Satoshi Matsuoka, Tomihisa Kamada, and Akinori Yonezawa. A general framework for bi-directional translation between abstract and pictorial data. *ACM Transactions on Information Systems*, 10(4):408–437, October 1992.

[10] Shin Takahashi, Satoshi Matsuoka, Akinori Yonezawa, and Tomihisa Kamada. A general framework for bidirectional translation between abstract and pictorial data. In *Proceedings of the ACM Symposium on User Interface Software and Technology*, volume 4, pages 165–174, November 1991.