Figures and Grammars: Meta-GUI Tool for Future Human-Computer Interaction

Jiro Tanaka and Hiroaki Kameyama

Department of Computer Science,
Graduate School of Systems and Information Engineering,
University of Tsukuba, Tsukuba, Ibaraki 305-8573, Japan
email: jiro@cs.tsukuba.ac.jp, kame@iplab.cs.tsukuba.ac.jp

Abstract

In this paper, we propose Viola, which can define CMG grammar graphically and execute a visual system on the same screen. By using Viola, the user can define various visual systems interactively. It became possible to define grammars using graphical expressions and construct visual systems dynamically, while checking the grammar at the execution time.

We have actually implemented Viola in Java (j2sdk 1.4.1_01). An example of defining a grammar for network diagram and its screen shots are also shown in this paper.

1 Introduction

Currently, the desktop computers is popular. However, how the computer will change in future? How will the computer-human interaction be?

Our assumption is that the time of personal computer (PC) will over in near future and the Personal Digital Assistant (PDA) and the cellular phone will replace the present PC. We are paying attention to the computer paradigm called "ubiquitous computing."

By using PDA or a cellular phone, the reservation of a ticket, location-based services, various kinds of information gathering, etc. become possible, while moving in the real world, rather than sitting down in front of a computer. In fact, various services using a cellar phone have already been started.

Since various researches have already been carried out for the display techniques for small screen computers, we are focusing the display techniques for the large display devices. We assume that the large display devices, such as the large plasma display and the wall projection display, will be placed everywhere in the world in near future. In such environment, it seems that the new the graphical user interface technology, i.e., visual interface technology, plays an important role. Therefore, we are working for visual parsing in these years.

2 Visual parsing

Though the textual parsing has already been well-established in computer science field, the visual paring is still in the preliminary stage.

Visual languages are used in various fields, such as ER diagrams, class diagrams of UML, formula, music, and diagrams that show relationships between characters appearing in TV dramas. Visual languages have structures like textual languages.

We can assume the diagrams which represent graph structures as a visual language. A special purpose graphic editor can be considered as a system to process a visual language. We call a system which processes a visual language a *visual system*. Visual systems proposed so far have been fixed on certain specifications. It was a difficult and time consuming job to modify those systems.

Therefore, our approach is to develop a general purpose parser generator, which can generate various spatial parsers by defining the grammars of various visual languages. The generated spatial parser can parse the given visual system and execute the specified actions.

We have developed the series of visual parser generators, such as Eviss [1, 2], VIC [3] and Rainbow [4, 5] so far. In this paper, we propose yet another visual parser generator Viola, which allows the interactive and graphical input of grammars for visual systems.

2.1 The extended Constraint Multiset Grammars

We use the extended Constraint Multiset Grammars (CMG) [6, 7] in defining the grammars of visual systems in visual parser generators. CMG consist of a set of terminal symbols, 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. The production rules are used to rewrite a multiset of tokens (the instances of the terminal or non-terminal symbols) for a new symbol.

The constraints maintain the relationships between the attributes of the tokens. A production rule is defined as follows:

$$\begin{array}{rcl} T(\vec{x}) & ::= & T_1(\vec{x_1}), \cdots, T_n(\vec{x_n})\ \mathit{where} \\ & & \mathit{exists}\ T'_1(\vec{x'_1}), \cdots, T'_m(\vec{x'_m}) \\ & & \mathit{where}\ C\ \mathit{and}\ \vec{x}\ =\ F\ \mathit{and}\ A \end{array}$$

When the attributes of the tokens $T_1, \cdots, T_n$ ("normal" components) and $T'_1, \cdots, T'_m$ ("exist" components) satisfy the constraints C, the tokens $T_1, \cdots, T_n$ are rewritten to the non-terminal symbol T. Exist components are needed to recognize T and are not rewritten to T [1]. F is the function that has the attributes $\vec{x_1}, \cdots, \vec{x_n}$ and $\vec{x'_1}, \cdots, \vec{x'_m}$ of the components as arguments, and the return value of the function is given to the non-terminal symbol T as its attribute.

[1]CMG also has "not_exist" and "all" components. For details, refer to [2, 6, 7].

Note that we have extended the original CMG to include action A. A is defined as "script program executed when the production rule is applied." In the extended CMG, we can specify arbitrary actions, such as computing values and rewriting figures.

2.2 List Tree example

List Tree is defined recursively by the following two production rules.

Rule 1: A non-terminal symbol "list" consists of a "circle" and a "text" in the center of it.

Rule 2: A non-terminal symbol "list" consists of a "circle," two "lines" and two "lists." The two "lists" are connected to the "circle" by the "lines."

These production rules can be written by the extended CMG as follows.

```
1:  list(point mid, integer mid_x,
2:                      string value) ::=
3:     C:circle, T:text
4:     where (
5:        C.mid == T.mid
7:     ) and {
8:        mid = C.mid
9:        mid_x = C.mid_x
10:       value = {script.string {
11:                     list @T.text@}}
12:    } and {
13:       display(value = @value@)
14:    }
15:
16: list(point mid, integer mid_x,
17:                     string value) ::=
18:    C:circle
19:    exists S1:list, S2:list,
20:                     L1:line, L2:line
21:    where (
22:       S1.mid == L1.end
23:       S2.mid == L2.end
24:       C.mid == L1.start
25:       C.mid == L2.start
26:       C.mid == T.mid
27:       S1.mid_x < S2.mid_x
28:    ) and {
29:       mid = C.mid
30:       mid_x = C.mid_x
31:       value = {script.string {
32:                concat [list @S1.value@]
33:                       [list @S2.value@]}}
34:       lef = C.lu_x
35:       right = C.rl_x
36:    } and {
37:       display(value = @value@)
38:    }
```

Lines 1 to 14 show the definition of the production rule 1. Line 1 shows the attributes of the non-terminal symbol "list." Attributes are "mid," "mid_x,"and "value." Line 3 shows that this non-terminal consists of a "circle" and a "text" string and these components are "Normal." At line 5, constraints are defined. This line shows that the attribute "mid" of "circle" C is equal to the attribute "mid" of "text" T. "mid" is an attribute that indicates

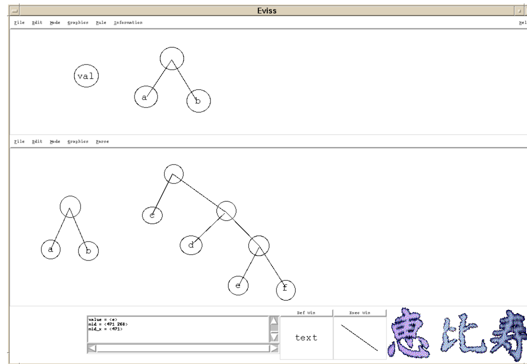


Figure 1: A snapshot of Eviss

the center's coordinates. In lines 8 to 11, the values of Attributes are defined. Line 8 shows that an attribute "mid" of "list" is equal to "mid" of "circle" C. Line 9 shows that an attribute "mid_x" of "list" is equal to "mid_x" of "circle" C. "mid_x" is an attribute indicating an abscissa. Lines 10 and 11 show the definition of an attribute "value." This definition represents that "text" string of "text" T is treated as a list. At the line 13, action is defined. This line shows output (*value*) when this production rule is applied.

Lines 16 to 38 show the definition for the production rule 2. Lines 19 and 20 show that two "lists" and two "lines" must exist somewhere in the visual sentence. Line 27 shows that "list" S1 is on the left side of "list" S2. This constraint distinguishes the left "list" from the right "list." Lines 31 to 35 show the definition of the attribute "value." Here, two "lists" S1 and S2 are connected.

3 Eviss

We have already implemented the spatial parser generator Eviss [1, 2]. Figure 1 shows the snapshot of a visual system that represents "List Tree." The upper half of the screen is called the *definition window*. A person who implements a visual system defines grammars of visual languages in the definition window. The bottom half is called the *execution window*.

In Eviss, figures are used to define rough grammars. At first, the user draws figures which he wants to define as a new non-terminal symbol from the definition window. We call these figures as "example figures." Eviss automatically extracts simple constraints and components from "example figure" and outputs to the CMG Input Window with text (Figure 2). Then the user edits the constraints and components from the CMG Input Window. The user can also specify actions in this phase.

At the *execution phase*, a user draws figure elements which should be analyzed to the execution window.

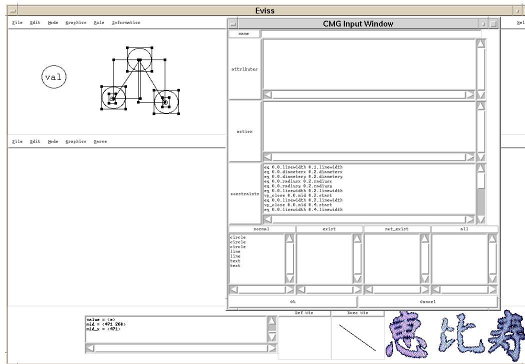


Figure 2: Defining grammars in Eviss

4 Viola

Viola is the latest version of Spatial Parser Generators developed by us. We have implemented Viola in Java (j2sdk 1.4.1_01).

There are two main differences between Eviss and Viola. The first difference is that Eviss has two windows, i.e., Definition Window and Execution Window, and only Execution Window has a spatial parser. Whereas, Viola has only one window. Because of having one window only, Viola has no border between Definition Window and Execution Window. Viola can understand the non-terminal symbols when the user defines the grammars.

The second difference is that Viola can define constraints by the direct manipulation of "example figures" without using CMG Input Window. In Eviss, user had to input CMG textually from CMG Input Window. This made difficult to define the visual system interactively. In the case of Viola, the user can define various visual systems interactively, even if the user does not know the grammar of CMG.

Viola is equipped with a drawing editor which can be used for defining grammar and for executing visual system. The drawing editor can handle rectangle, oval, arc, line, image and text. We can select the input figure element by clicking the button which is placed on the top of the drawing window.

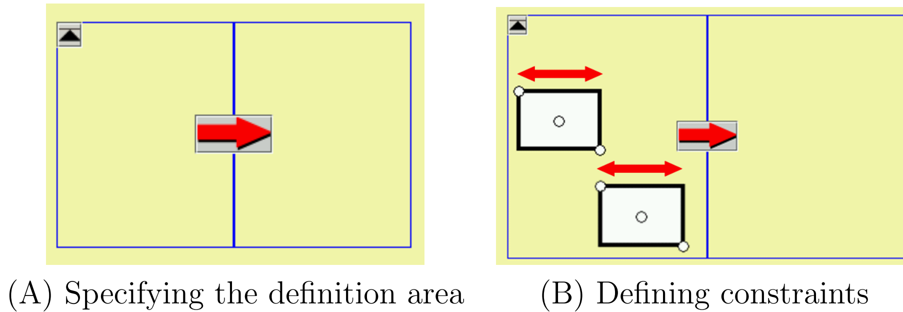
It is also possible to input the handwritten strokes by clicking the right button of the mouse. Attributes, such as the color, the width and the font, can be changed from the menu. Similar to the general purpose drawing editor, we can move, copy or delete the drawn figures.

4.1 Graphical definition of a grammar

In Viola, the user can graphically define a grammar by following steps.

Specifying the definition area. We specify the definition area first. Select the "rule" button from the tool panel and draw the frame as shown in Figure 3 – (A). We can define the grammar by editing figures inside of the frame.

Defining non-terminal symbols. Non-terminal symbols can be defined by "rule." They can be defined by drawing elementary symbols which compose the non-terminal symbol



(A) Specifying the definition area (B) Defining constraints

Figure 3: Graphical definition of a grammar

inside of the left frame of the definition area.

Defining constraints. Constraints are the conditions which elementary symbols must satisfy in order to apply the "rule." They can be defined by manipulating the elementary symbols which appears in the left frame. When the symbols are manipulated, the system infers the constrains between symbols, such as the coordinates of two points agrees, the width of two points agrees, etc., and shows the inferred constraint visually as shown in Figure 3 – (B).

Defining actions. It is possible to define various actions, such as the deletion and the movement of figures. Actions are executed when the "rule" is applied. Figures which are drawn inside of the left frame of the definition area are copied to the right side by clicking the center arrow of the definition area. We can define the action by editing the right side figure in order to express the figure after the action is performed.

4.2 Executing visual system

In order to execute the visual system, we only need to draw figures, after defining the grammar. Viola analyzes the drawn figures incrementally and impose the constraints between the figures. After the figures are parsed, the constraints are maintained, even if we move the figures. We use Chorus[8] as the constraint solver, which can handle geometric constraints.

4.3 Defining gestures of hand-writing strokes

We often use hand-writing strokes, which are called "gestures," on PDA or Tablet PC. Delete or modify actions are expressed by "gestures." By treating a hand-writing strokes similar to figures, Viola can define "gestures."

For example, when a notched handwriting stroke is drawn on a circle, it means the gesture of deleting a circle. To define this rule, we need to input a circle and the notched hand-writing strokes into the left frame of the definition area first. The right frame of the definition area are kept blank as shown in Figure 4. SATIN[9] is used for recognition of a handwriting stroke.

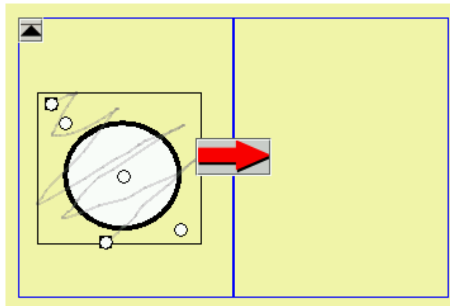
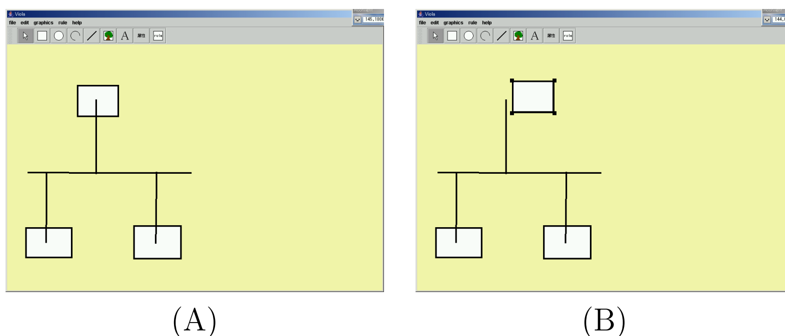


Figure 4: Defining a “gesture” for deleting a circle



(A) (B)

Figure 5: Network diagrams

5 An example of defining a grammar

In order to explain the procedure of defining a grammar using Viola and how to realize a visual system, we show the example of defining a network diagram.

In Viola, a figure can be freely drawn like the ordinary drawing tool. Although it is possible to draw a network diagram as shown in Figure 5 – (A), the drawn figure is not recognized as a network diagram because no rules for a network diagram are defined. If a square is moved as shown in Figure 5 – (B), a network diagram will collapse. In order to make the drawn figure recognize as a network diagram, it is necessary to define suitable rules. This can be realized by defining two rules as shown in Figure 6.

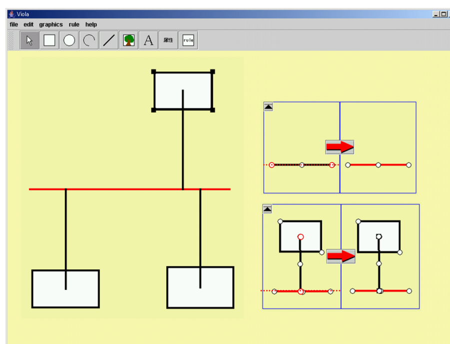
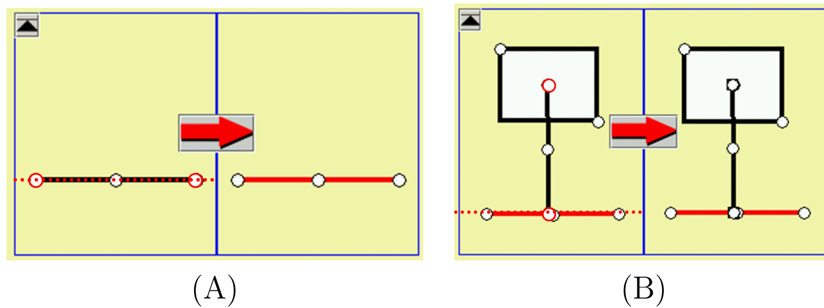


Figure 6: Defining a grammar



(A) (B)

Figure 7: Defining a network diagram

As a first rule, we define the line used as the trunk of a network diagram. We specify the definition area and draw a straight line to the left-hand side frame of the definition area. At that time the system automatically infers the constraint that the Y coordinates of the starting point and a terminal point of the line are equal and it is illustrated by the red dotted line. Then we change the color of the line at the right frame of the definition area into red and define it as an action (Figure 7 –(A)).

As a second rule, we define the line which connects a trunk to network apparatus. The definition range of a rule is drawn newly and a straight line is drawn to a left-hand side area. Then, the drawn straight line is recognized as a trunk under a previous rule.

One rectangle showing network apparatus is inputted, and a straight line is inputted so that the rectangle and trunk will be connected. The starting point of the line is agreed with the center point of the rectangle, and the Y coordinate of the terminal point of the line is agreed with the Y coordinate of a trunk (Figure 7 –(B)).

By defining the above two rules, the diagram of Figure 5 – (A) is recognized as a network diagram. The layout of a network diagram does not collapse, even if we move the rectangle which expresses network apparatus, since the relation between figures is held by Viola.

6 Meta-GUI as a tool for intelligence

When we think the intellectual activities of human beings, it often carried out "drawing an idea on a figure." For example, in KJ method, a memorandum is arranged on two dimensional space. When we are going to support such human being's intellectual activities using a computer, the display screen of PC is truly too small. Only by using a big screen display, the infrastructure which can support such human being's intellectual activities will be established. Although the computer had only the capability to process a text at the beginning, by using GUI (Graphical User Interface), it became the tool more sociable to human.

We imagine that the paradigm which should also be called "meta-GUI" will come shortly. Although there are various possibilities about whether meta-GUI is considered to be what kind of thing, here we propose is a "visual system," which is not a passive user interface. We propose that 2-dimensional graphics have structure, and they need to recognize the meaning of structure, and should operate wisely (actively). Here, we expect that the research on spatial parser generators play the important role.

7 Concluding remarks

Textual notations have been used to define a grammar in previously proposed spatial parser generators. This made difficult to define the visual system interactively.

In this paper, we proposed Viola, which can define CMG grammar graphically and execute a visual system on the same screen. By using Viola, the user can define various visual systems interactively, even if the user does not know the grammar of CMG. It became possible to define grammars simultaneously using graphical expression, while inspecting the rightness of grammar and dynamically construct a visual system.

8 Acknowledgments

The authors would like to express thanks to the previous and current members of IPLAB, University of Tsukuba, for their help and programming support.

References

[1] Akihiro Baba and Jiro Tanaka: A Visual System Having a Spatial Parser Generator, Transactions of IPSJ, Vol. 39, No. 5, pp. 1385–1394, 1998, (in Japanese).

[2] Akihiro Baba and Jiro Tanaka: Eviss: a Visual System Having a Spatial Parser Generator, Proceedings of Asia Pacific Computer Human Interaction, pp. 158–164, 1998.

[3] Kenichirou Fujiyama, Kazuhisa Iizuka and Jiro Tanaka: VIC: CMG Input System Using Example Figures, Proceedings of the International Symposium on Future Software Technology, pp.67–72, 1999.

[4] SackTae Joung and Jiro Tanaka: Rainbow: Implementing Layout Constraints in Visual System Generator, Transactions of IPSJ, Vol.41, No.5, pp. 1246-1256, 2000, *in Japanese*.

[5] SackTae Joung and Jiro Tanaka: Generating a Visual System with Soft Layout Constraints, Proceedings of the International Conference on Information – Information'2000 – (to appear).

[6] Kim Marriott: Constraint Multiset Grammars, Proceedings of the IEEE Symposium on Visual Languages, pp. 118–125, 1994.

[7] Sitt Sen Chok and Kim Marriott: Automatic Construction of User Interfaces from Constraint Multiset Grammars, Proceedings of the IEEE Workshop on Visual Languages, pp. 242–249, 1995.

[8] Hiroshi Hosobe: A Modular Geometric Constraint Solver for User Interface Applications, Proceedings of the 14th Annual ACM Symposium on User Interface Software and Technology (UIST'01), pp. 91–100, 2001.

[9] Jason I. Hong and James A. Landay: SATIN: a toolkit for informal ink-based applications, Proceedings of the ACM Symposium on User Interface Software and Technology, pp. 63–72, 2000.