Describing a Drawing Editor by Using Constraint Multiset Grammars

Kazuhisa Iizuka, Jiro Tanaka and Buntarou Shizuki
Institute of Information Sciences and Electronics, University of Tsukuba
1-1-1 Tennoudai, Tsukuba, Ibaraki, 305-8573, Japan
{iizuka, jiro, shizuki}@iplab.is.tsukuba.ac.jp

ABSTRACT

Systems that can handle visual languages such as Penguins and Eviss use a generalized editor for handling figures of the visual language. However, these editors have limited editing operations. We propose an approach to describe operations on the figures as a part of the language's grammar. It becomes easy to provide an editing operation based on the analysis of a figure, and the implementation is independent of the execution environment.

Keywords

Drawing Editor, Parser, Visual Language, Grammar.

INTRODUCTION

Visual language is a set of drawings that has a meaningful combination of fundamental figures. The drawing figures must be analyzed by the parser to recognize these figures. The visual language's grammar gives the language's specification. There are many types of grammars such as Positional Grammars [4], Relational Grammars [6], and Constraint Multiset Grammars (CMG) [8].

A parser generator, which generates the parser automatically by specifying its grammar, has been investigated. For instance, see SPARGEN [7], VLCC [5], Penguins [3], Eviss [1], and Rainbow [9].

CMG is used to provide an interactive system based on the visual language. Penguins, Eviss, and Rainbow are environments that can generate an interactive system. In these systems, figures edited on the attached editor will be analyzed immediately. However, these systems can only perform basic operations on the fundamental figures; they cannot perform the specialized operation depending on the visual language. For example, they cannot change the operations of dragging the figures based on the analysis results. In order to realize special editing operations depending on the language, an editor must be created for every language. However, it is troublesome to make programs that ask the parser for the analysis result every time. Moreover, special editing operations must be described for every execution environment so it has the problem of lacking portability.

To solve these problems, we propose an approach to describe the operations on the figures as a part of visual language's grammar, not by using an editor. As an example, we present a simple drawing editor described by the grammar in its editing operations.

DESCRIBING DRAWING EDITOR BY USING CMG

We propose an approach to describe operations on the figures as a part of the grammar and extend CMG to realize it. This extension introduces the *Action* notation to CMG so the operation on the figures can be described. We also introduce special tokens in order to notify the parser about the mouse operation or to hold an internal state. The operations on the figures are described as a grammar by using these tokens. Each operation is described as a production of CMG according to the state of the special tokens.

We described a simple drawing editor as the grammar. The user of this editor can write, move, or reform a rectangle and a line. These functions are described as CMG. This editor receives the user inputs, reflects the inputs in the special tokens of the parser, and has a canvas that receives processing of the parser; it also draws and changes the figures. This canvas does not have an editing function but has functions only as a user interface. The parser that analyzes figures based on the grammar with productions performs the actual processing.

SIMPLE DRAWING EDITOR

We now present a simple drawing editor with which the user can draw or edit a rectangle and a line. This editor has a canvas and a menu. The canvas has the following functions.

- Receives basic mouse events such as Move, Press, Release and Drag.
- Draws a specified drawing.

An identifier is assigned for each figure drawn. We call this the "Object ID." The Object ID is a natural number. The ID will increase as figures are created. The figure with the highest number is drawn upwards. The editor has an editing mode menu from which Rectangle, Line, Edit or Delete can be selected. Only one mode is chosen at a time. The editor provides the following functions by using CMG. These functions operate in each mode.

(1) **Draw Rectangle.** By dragging a canvas, a rectangle is generated by creating a diagonal line from a starting point to a terminal point.

(2) **Draw Line.** By dragging a canvas, a line is generated by connecting the starting point and a terminal point.

(3) **Edit.** Provides the following three sub-functions.

(3.1) **Handle.** A figure is selected by clicking it. Only one figure is selected. The selected figure will have two handles that correspond to the starting point and terminal point of the figure. The handle indicates the selected figure and is used for transforming the figure.

(3.2) **Move.** The figure can be moved by dragging.

(3.3) **Transform.** The form of a figure can be changed by dragging a handle.

(4) **Delete.** A figure can be erased by clicking it.

SYSTEM

In this section, we describe the drawing editor system and its grammar.

Parser

The CMG parser we use is based on the parser proposed by Chok and Marriott [2, 10]. We use an extended CMG parser that can invoke actions such as alternate attributes of the RHS token or delete the specified token when a production is applied.

CMG is composed of productions. If a production is applied, it will register a new token specified on the LHS. The attributes of the new token are defined by attributes of RHS tokens. The following production means that the token P can be rewritten by tokens P_1, ..., P_n provided that produced tokens satisfy all *Constraints*. Actually, when tokens P_1, ..., P_n satisfy all *Constraints*, the parser creates a new token P, and makes P_1, ..., P_n disappear from the parser. Token P's attributes are set up according to the *Attributes Assignment*. At that time, the *Action* is executed. The *Action* may be alternating attributes of the RHS tokens, deleting some token, adding a new token that differs from the LHS token, and so on.

```
P ::= P1:p1, ...,Pn:pn where (
    Constraints
) {
    Attributes Assignment & Action
}
```

Eviss and Rainbow introduced the Action to CMG. These researches use the Action to rewrite figures by parsing results.

Token

Special tokens, such as Pointer, Mode, and Control, are used for transferring the event on the canvas to the parser, maintaining the internal states, and so on. Using these special tokens, the role of the parser is expanded from parsing visual objects to analyzing interactive system states.

Generally, tokens that are used in CMG are visible objects with some shapes. However, these special tokens are invisible. The special tokens that we introduced are thus not visible objects but have attributes. The special tokens used in this editor are the Pointer, Mode, and Control tokens. The Object, Start_handle, and End_handle tokens are normal tokens.

Special Token

The Pointer and Mode tokens are used to pass the event information on the canvas to the parser. Each token has some attributes.

The Pointer token has mouse information. The event on the canvas, such as moving or pushing, is treated as an attribute change of the Pointer token. For example, if a button is pressed, the button attribute of the Pointer token will be changed from "released" to "pressed." The Pointer token has the following attributes.

- pos: Position of the mouse.
- button: Conditions of the button. (pressed, released)

The Mode token has a menu state. If the menu is selected on the canvas, the attribute of the Mode token will be rewritten.

- mode: Current editing mode.
  (rectangle, line, edit, delete)

We use the Control token to keep the internal state. The Control token has the following attributes.

- start: Starting point of the figure. It is used while creating an object.
- end: Terminal point of the figure. It is used while creating an object.
- target: Object ID of a currently drawn figure.
- state: Current state.
  (normal, draw-rectangle, draw-line, select-object, move-object, move-start-handle, move-end-handle, delete-object)
- pointer_pos: Previous mouse position. It is used for calculating the movement of the mouse.

Normal Token

Object, Start_handle, and End_handle are normal tokens. These tokens correspond to the visible objects.

The Object token corresponding to each figure drawn on the canvas is used. The Object token has the following attributes.

- id: Object ID of the figure.
- type: Type of the figure. (rectangle, line)
- pos: Position of the figure.
- start: Starting point of the figure.
- end: Terminal point of the figure.

The Start_handle and End_handle tokens correspond to the handle objects. The handle is used for reforming a figure. The tokens have the following attributes.

- id: Object ID of the handle.

- pos: Position of the handle.

System Structure

The system consists of three components: Control, Canvas, and Parser. Control initializes the following.

- Initializes Parser, and creates the Pointer, Mode, and Control tokens.
- Initializes the Canvas, and establishes the bindings of mouse and menu events to each token.

Canvas has the following functions.

- Alters the Pointer and Mode tokens if mouse or menu events occur.
- Draws, alters, or deletes the specified figure when Parser directs it.

Parser has the following functions.

- Parses tokens depending on the productions when a token is added or altered.
- If a production is applied, creates new token as production definition. If necessary, Parser directs to the canvas, changes token attributes, deletes specified tokens, or adds a new token.

PRODUCTION

The following 19 productions are defined for the Simple Drawing Editor mentioned above.

Draw Rectangle

Production (1-1) draws a temporary rectangle when pressed on the canvas. It records the Object ID and the position of the rectangle, and changes the Control token to the draw-rectangle state. Production (1-2) transforms the temporary rectangle by dragging the canvas in the draw-rectangle state. Production (1-3) registers the written rectangle into the parser as an Object token; it then changes the Control token to the normal state.

```
// Production (1-1)
RectangleDrawStart ::=
  M:mode, P:pointer, C:control where (
  M.mode == 'rectangle' &&
  P.button == 'pressed' &&
  C.state == 'normal'
) {
  C.start = P.pos;
  C.end = P.pos;
  C.target = createRectangle(C.start, C.end);
  C.state = 'draw-rectangle';
}

// Production (1-2)
RectangleDrawDrag ::=
  M:mode, P:pointer, C:control where (
  M.mode == 'rectangle' &&
  P.button == 'pressed' &&
  C.state == 'draw-rectangle'
  C.end != P.pos &&
) {
  C.end = P.pos;
  transformObject(C.target, (C.start, C.end));
}
```

```
// Production (1-3)
RectangleDrawFinish ::=
  M:mode, P:pointer, C:control where (
  M.mode == 'rectangle' &&
  P.button == 'released' &&
  C.state == 'draw-rectangle'
  )
) {
  C.state = 'normal';
  R = newToken('object');
  R.type = 'rectangle';
  R.id = C.target;
  R.start = C.start;
  R.end = C.end;
  R.pos = (R.start, R.end);
}
```

The function *createRectangle*() draws a rectangle, which makes a diagonal line from a starting point to a terminal point. It then returns an Object ID. The function *transformObject*() changes the starting point and the terminal point of the object corresponding to the specified ID. The function *newToken*() adds a new token that has the specified type and attributes to the parser, then returns a new token.

Draw Line

The Draw Line function is composed of three productions. Those productions are similar to productions applied for Draw Rectangle. Therefore, the details are omitted.

```
// Production (2-1)
LineDrawStart ::= ...

// Production (2-2)
LineDrawDrag ::= ...

// Production (2-3)
LineDrawFinish ::= ...
```

Edit - Handle

Production (3.1-1), in the normal state, changes the Control token to the move-object state when a figure is pressed. It creates handles corresponding to a starting point and a terminal point on the canvas, and registers them into the parser simultaneously. In order to react to the top object, the not_exist sentence excludes an object that comes below the target object at the mouse position. Using the Object ID (O.id and N.id), the display orders of figures are checked. After the figure is pressed, Production (3.2-1) or (3.2-2) will be applied. Production (3.1-2), in the select-object state, cancels selection when a figure is pressed. Production (3.1-1) will shift it to the move-object state. If there is a handle on the position, it is managed by a transform function (such as Production (3.3-1)). If there is no object, Production (3.1-2) changes the Control token to the normal state. Production (3.1-3), in the select-object state, changes the Control token to the normal state when the editing mode was changed.

```
// Production (3.1-1)
SelectObject ::=
  M:mode, P:pointer, C:control, O:object where (
    M.mode == 'edit' &&
    P.button == 'pressed' &&
    C.state == 'normal' &&
    isOverlapped(O.type, O.pos, P.pos) &&
    not_exist N:object where (
      isOverlapped(N.type, N.pos, P.pos) &&
      O.id < N.id
    )
  ) {
    C.state = 'move-object';
    C.pointer_pos = P.pos;
    C.target = O.id;
    S = newToken('start_handle');
    S.id = createHandle(O.start);
    S.pos = O.start;
    E.newToken('end_handle')
    E.id = createHandle(O.end)
    E.pos = O.end;
  }

  // Production (3.1-2)
  SelectObjectCancel ::=
    M:mode, P:pointer, C:control, S:start_handle,
  E:end_handle where (
    M.mode == 'edit' &&
    P.button == 'pressed' &&
    C.state == 'select-object' &&
    !isOverlapped('handle', S.pos, P.pos) &&
    !isOverlapped('handle', E.pos, P.pos)
  ) {
    C.state = 'normal';
    deleteObject(S.id);
    deleteToken(S);
    deleteObject(E.id);
    deleteToken(E);
  }

  // Production (3.1-3)
  SelectObjectCancelByMode ::=
   M:mode, , P:pointer, C:control, S:start_handle,
  E:end_handle where (
    M.mode != 'edit' &&
    P.button == 'released' &&
    C.state == 'select-object'
  ) {
    C.state = 'normal';
    deleteObject(S.id);
    deleteToken(S);
    deleteObject(E.id);
    deleteToken(E);
  }
```

The function *isOverlapped*() returns true when a mouse is on the figure. The function *drawHandle*() draws a handle, and returns an Object ID. The function *deleteObject*() deletes the specified figure on the canvas. The function *deleteToken*() removes the specified token from the parser.

Edit - Move

Production (3.2-1) moves a figure by dragging the figure in the move-object state. Furthermore, handles are also moved with a figure. Production (3.2-2) changes the Control token to the select-object state when it is released in the move-object state.

```
  // Production (3.2-1)
  MoveObjectStart ::=
    M:mode, P:pointer, C:control, O:object,
  S:start_handle, E:end_handle where (
    M.mode == 'edit' &&
    P.button == 'pressed' &&
    C.state == 'move-object' &&
    C.pointer_pos != P.pos &&
    C.target == O.id
  ) {
    O.start = calcPosition(C.pointer_pos, P.pos,
      O.start);
    O.end = calcPosition(C.pointer_pos, P.pos, O.end);
    O.pos = (O.start, O.end);
    S.pos = calcPosition(C.pointer_pos, P.pos, S.pos);
    E.pos = calcPosition(C.pointer_pos, P.pos, E.pos);
    C.pointer_pos = P.pos;
    transformObject(C.target, O.pos);
    transformObject(S.id, S.pos);
    transformObject(E.id, E.pos);
  }

  // Production (3.2-2)
  MoveObjectFinish ::=
    M:mode, P:pointer, C:control, O:object where (
    M.mode == 'edit' &&
    P.button == 'released' &&
    C.state == 'move-object'
  ) {
    C.state = 'select-object';
  }
```

The function *calcPosition*() returns the position, which adds the difference of the position of the first two arguments to the position of the third argument. For example,

```
  calcPosition((x1, y1), (x2, y2), (x3, y3))
```

will return

```
  (x3+x2-x1, y3+y2-y1).
```

Edit - Transform

Production (3.3-1) changes the Control token to the move-start-handle state when it is pressed on a start-handle in the select-object state. (Refer to Production (3.1-2).) Since the end-handle is written above the start-handle, it investigates whether the end-handle has overlapped or not. Production (3.3-2) transforms the figure in the move-start-handle state. Furthermore, the handle is also moved by figure transformation. Production (3.3-3) changes the Control token to the select-object state when it is released in the move-start-handle state.

```
  // Production (3.3-1)
  MoveStartHandleStart ::=
    M:mode, P:pointer, C:control, S:start_handle,
  E:end_handle where (
    M.mode == 'edit' &&
    P.button == 'pressed' &&
    C.state == 'select-object' &&
    isOverlapped('handle', S.pos, P.pos) &&
    !isOverlapped('handle', E.pos, P.pos)
  ) {
    C.state = 'move-start-handle';
    C.pointer_pos = P.pos;
  }

  // Production (3.3-2)
  MoveStartHandleDrag ::=
    M:mode, P:pointer, C:control, O:object,
  S:start_handle where (
    M.mode == 'edit' &&
    P.button == 'pressed' &&
    C.state == 'move-start-handle' &&
    C.pointer_pos != P.pos &&
    C.target == O.id
  ) {
    O.start = calcPosition(C.pointer_pos, P.pos,
      O.start);
    O.pos = (O.start, O.end);
    S.pos = calcPosition(C.pointer_pos, P.pos, S.pos);
    C.pointer_pos = P.pos;
    transformObject(C.target, O.pos);
    transformObject(S.id, S.pos);
  }

  // Production (3.3-3)
  MoveStartHandleFinish ::=
    M:mode, P:pointer, C:control where (
    M.mode == 'edit' &&
    P.button == 'released' &&
    C.state == 'move-start-handle'
  ) {
    C.state = 'select-object';
  }

  // Production (3.3-4)
  MoveEndHandle ::=
    M:mode, P:pointer, C:control, E:end_handle
  where (
    M.mode == 'edit' &&
    P.button == 'pressed' &&
    C.state == 'select-object' &&
    isOverlapped('handle', E.pos, P.pos)
  ) {
    C.state = 'move-end-handle';
    C.pointer_pos = P.pos;
  }

  // Production (3.3-5)
  MoveEndHandleStart ::= ...

  // Production (3.3-6)
  MoveEndHandleFinish ::= ...
```

Production (3.3-4) to Production (3.3-6) are mappings of the end-handle. They are described in the same way as Productions (3.3-1) to (3.3-3).

Delete

Production (4-1) deletes the figure it is pressed against and changes the Control token to the delete-object state. The not_exist sentence is used in order to react to the top object when figures overlap. Production (4-2) changes the Control token to the normal state when it is released.

```
  // Production (4-1)
  DeleteObjectStart ::=
    M:mode, P:pointer, C:control, O:object where (
    M.mode == 'delete' &&
    P.button == 'pressed' &&
    C.state == 'normal' &&
    isOverlapped(O.type, O.pos, P.pos) &&
    not_exist N:object where (
      isOverlapped(N.type, N.pos, P.pos) &&
      O.id < N.id
    )
  ) {
    C.state = 'delete-object';
    deleteObject(O.id);
    deleteToken(O);
  }

  // Production (4-2)
  DeleteObjectFinished ::=
    M:mode, P:pointer, C:control where (
    M.mode _== 'delete' &&
    P.button == 'released' &&
    C.state == 'delete-object'
  ) {
    C.state = 'normal';
  }
```

IMPLEMENTATION

We have implemented the drawing editor described above. Figure 1 and 2 show snapshots of this editor. This implementation consists of the CMG parser, grammar definition of CMG and support programs. The supporting programs contain initializing scripts (create canvas and editing mode buttons) and user-define functions (*createRectangle*(), *isOverlapped*(), and so on). The initializing scripts bind

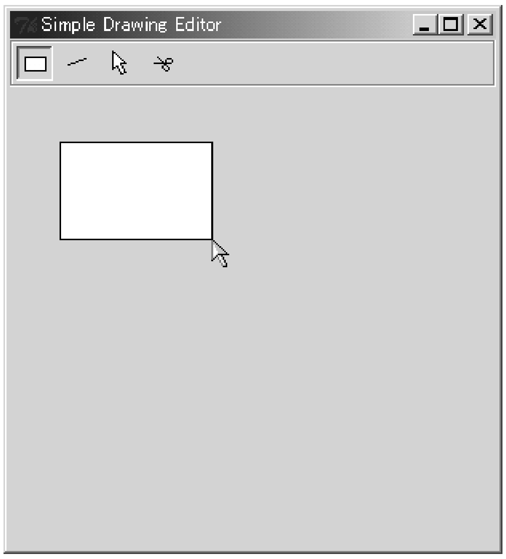

Figure 1. Drawing a rectangle.

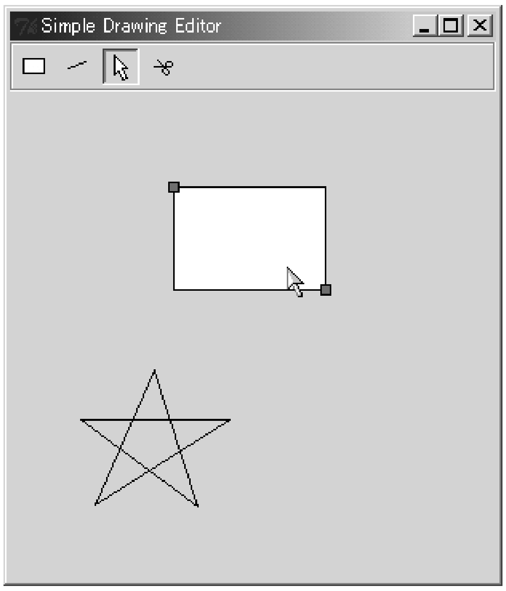

Figure 2. Moving an object.

events to special tokens. All descriptions of editing operations are defined by the grammar. These supporting programs are written in Tcl/Tk and consist of about 300 lines.

ADVANTAGES

There are various advantages in our approach. The important thing is the integration of describing operations and parsing figures. The operation and parsing figures can be described on the same framework. Therefore, it becomes easier to describe the operation depending on the figure's meaning provided by parsing. For example, depending on the conditions of the overlap of a handle and the pointer, pushing the button forks to Production (3.1-2) or Production (3.3-1).

It is possible to isolate implementations of operations from the editor. This means that the editor can be ported with no modifications to every target environment, implementation language (e.g. C, Java, and Tcl) and GUI toolkit (e.g. GTK, Java AWT, and Tk).

CONCLUSION

We proposed an approach to describe operations on the figures as a grammar. As an example of this, we gave a simple drawing editor employing a grammar. The editor is implemented based on the extended CMG and the special tokens. The detailed operations are described as productions. The proposed technique makes it easy to offer the editing operation based on the analysis result of a figure, and the implementation is independent of the execution environment.

REFERENCES

1. A. Baba and J. Tanaka, "A Visual System Having a Spatial Parser Generator", IPSJ Journal, Vol.39, No.5, pp.1385-1394, 1998. (In Japanese)
2. S. S. Chok and K. Marriott, "Automatic Construction of User Interfaces from Constraint Multiset Grammars", Proceedings of IEEE Symposium on Visual Languages, pp. 242-249, 1995.
3. S. S. Chok and K. Marriott, "Automatic Construction of Intelligent Diagram Editors", Proceedings of the ACM Symposium on User Interface Software and Technology, pp.185-194, 1998.
4. G. Costagliola, A. D. Lucia, S. Orefice, and G. Tortora, "Positional Grammars: a Formalism for LR-like Parsing of Visual Languages", Visual Languages Theory, Springer, 1998.
5. G. Costagliola, G. Tortora, S. Orefice, and A. D. Lucia, "Automatic Generation of Visual Programming Environments", IEEE Computer, Vol.28, No.3, pp.56-66, March 1995.
6. F. Ferrucci, G. Tortora, M. Tucci, and G. Vitiello, "A Predictive Parser for Visual Languages Specified by Relation Grammars", Proceedings of IEEE Symposium on Visual Languages, pp.245-252, 1994.
7. E. J. Golin and T. Maglierry, "A Compiler Generator for Visual Languages", Proceedings of IEEE Symposium on Visual Languages, pp.314-321, 1993.
8. R. Helm, K. Marriott, and M. Odersky, "Building Visual Language Parsers", Conference proceedings on Human Factors in Computing Systems (CHI'91), pp.105-112, 1991.
9. S. Joung and J. Tanaka, "Generating a Visual System with Soft Layout Constraints", Proceedings of the International Conference on Information (Information'2000), pp.138-145, 2000.
10. K. Marriott, "Constraint Multiset Grammars", Proceedings of IEEE Symposium on Visual Languages, pp.118-125, 1994.