Framework for Interpreting Handwritten Strokes using Grammars

Buntarou Shizuki[1], Kazuhisa Iizuka[1], and Jiro Tanaka[1]

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

Abstract. To support the rapid development of pen-based structured diagram editors, we propose a framework for describing such editors. The framework uses grammar to describe the context, *i.e.*, the positional relationship between handwritten strokes and other objects, which can be used to interpret ambiguous results of pattern matching, and to describe the syntax of the target diagrams. We implemented the framework by extending our visual system, which supports the rapid prototyping of structured diagram editors.

1 Introduction

Diagrams are used frequently. Examples used in UML software design include class diagrams, sequence diagrams, and statecharts. Other examples are diagrams that represent data structures, such as binary trees, stacks, and lists. Organization charts are widely used in daily life. Consequently, it would be convenient if there were a system that would allow users to sketch such diagrams on tablet computers and electronic whiteboards using a stylus. The system would then interpret the handwritten strokes to recognize the structure of the diagrams, and then make them neater or transform them into other styles.

One big problem with developing such systems is the ambiguity that is inherent in handwritten strokes. Often, a single stroke cannot be interpreted from its shape alone. However, the context of where a stroke is drawn can eliminate some choices, and sometimes eliminate the ambiguity. For example, assume that we are editing a binary tree. In this context, if a linear stroke is made within a circular stroke, we might recognize the circular stroke as a circle, whereas the linear stroke might be either the letter "l" or the number "1".

Diagram editors that can analyze *handwriting* must be able to perform the following two functions:

Recognize handwritten strokes: The editor must recognize a given handwritten stroke using both the shape of the stroke and its context.

Parse the relationships between strokes: A *spatial parser* parses the relationships between recognized strokes that have been input in a random, unordered fashion. The parser recognizes the structure of strokes and then performs follow-up actions based on its interpretation of the results.

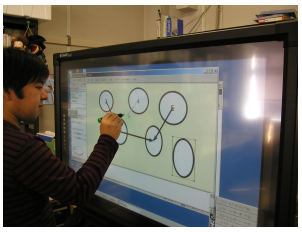

Fig. 1. A binary tree editor generated from its description

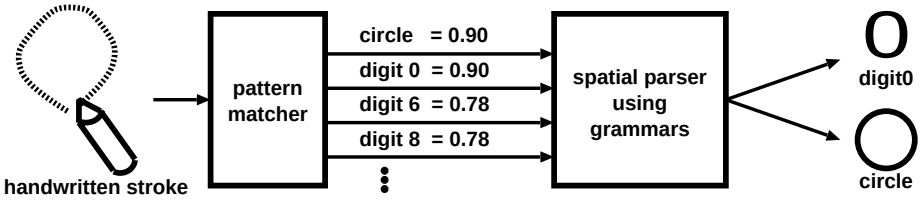

Fig. 2. The interpretation of handwritten strokes depending on the context

To help the systematic development of spatial parsers, *spatial parser generators* have been proposed, such as SPARGEN[1], VLCC[2], and Penguins[3]. A spatial parser generator generates the parser automatically from the specifications of the target diagram. A *grammar* is used to describe these specifications. However, no framework to assist in the systematic development of both of these functions has been proposed.

To provide developers with the means to develop structured diagram editors with these two functions, we propose a framework that recognizes ambiguous handwritten strokes that are drawn as parts of structured diagrams. The framework adopts an approach that uses grammar to describe: 1) the syntax of the target domain, and 2) how handwritten strokes should be interpreted. The spatial parser then selects appropriate candidates from several possible candidates from a pattern matcher using the grammatical description given by the developer. We have implemented the proposed framework. Fig. 1 shows a structured diagram editor for editing binary trees that was produced using our framework.

2 The Interpretation of Handwritten Strokes using Grammar

Fig. 2 illustrates the proposed framework for interpreting handwritten strokes using grammar. The pattern matcher performs its pattern-matching algorithm for each handwritten stroke every time a stroke is drawn. The matcher then produces an n-best list for each stroke. The spatial parser then parses the stroke with other existing strokes and objects to find plausible choices in the n-best list according to a given grammatical description. When the parser can select one

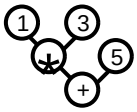

Fig. 3. An example of a diagram: a binary tree

choice, then the stroke has been recognized. When the parsing fails, the parser leaves the stroke on the canvas for future analysis.

For example, consider a diagram editor for binary trees. When the circular stroke illustrated in Fig. 2 is drawn on the canvas, the pattern matcher produces an n-best list of probabilities, which includes 0.90 for a circle, 0.90 for the digit 0, 0.78 for the digit 6, 0.78 for the digit 8, and so forth. The spatial parser will select the digit 0 if a stroke is drawn within a circle, since the stroke can be considered the label of a node. If the stroke is drawn around some text, the parser will select a circle pattern.

Note that the framework uses both the shape of the stroke and its context, *i.e.*, the positional relationships involving the handwritten strokes and other objects, to recognize a given handwritten stroke. This recognition enables the selection of one choice from multiple choices, even if the choices have the same or lower probabilities than others, such as the digit 0 and the circle in Fig. 2.

Below, Section 2.1 describes the grammar that we use in this paper, with a description of a sample structured diagram editor. Section 2.2 introduces a special token that enables the developer to describe rules for recognizing a handwritten stroke depending on its context.

2.1 Describing structured diagram editors using grammar

We use a grammar that is based on CMGs[4]. In this grammar, a rule has the form:

$$P ::= P_1, \cdots, P_n \; where \; C \; with \; Attr \; and \; Action$$

This means that the non-terminal symbol P can be composed of the multiset of symbols $P_i (i = 1, \cdots, n)$ when the attributes of all of the symbols satisfy constraint C. The attributes of P are assigned in *Attr*. It also executes *Action* after performing the application. Note that *Action* is our extension of the original CMGs for convenience.

The two rules listed below are the specifications for binary trees (see Fig. 3), and they are used as examples to provide explanations in the remainder of this paper.

```
# Rule 1
Node::=C:Circle,T:Text where (
   close(C.mid,T.mid)
) {
```

Table 1. The attributes of a gesture token

name	value
`pattern`	n-best list from the pattern matcher
`start`	x-y coordinates of the starting point
`end`	x-y coordinates of the ending point
`bound`	bounding box of the handwritten stroke
`length`	length of the handwritten stroke
`time`	turnaround time for inputting the stroke

```
   cp = C.mid;
   r  = C.radius;
} { }

# Rule 2
Node::=N1:Node,N2:Node,N3:Node,L1:Line,L2:Line where (
   inCircle(L1.start,N1.cp,N1.r) && inCircle(L1.end,N2.cp,N2.r) &&
   inCircle(L2.start,N3.cp,N3.r) && inCircle(L2.end,N2.cp,N2.r)
) {
   cp = N2.cp;
   r  = N2.r;
} { }
```

The first rule indicates that a node consists of a circle and some text. The midpoints of the circle and text should be close together. `close(P1,P2)` is the user-defined function that tests whether the distance between `P1` and `P2` is within a given threshold. If so, the circle and text are reduced to a node. The attribute `cp` of the node is defined as specifying the *connection point*. This connection point is the point at which an edge might be connected within a tolerable error `r`, which is defined by the midpoint and radius of the circle. `cp` and `r` are for later use. This rule does not specify any action.

The second rule defines the composition of the nodes. It specifies a composite node consisting of three nodes, `N1`, `N2`, and `N3`, and two lines, `L1` and `L2`. `L1` must start near the connection point `N1` and end near `N2`. The condition is checked by calling the user-defined function `inCircle(P,C,R)`. The function tests whether a given point `P` lies within a circle with center point `C` and radius `R`. Similarly, `L2` should start near the connection point `N3` and end near that of `N2`. The connection point for the composite node itself is assigned as the connection point of `N2`.

By providing the spatial parser generator with these two rules, `close()`, and `inCircle()`, a structured diagram editor specialized for editing binary trees can be developed.

2.2 A handwritten stroke as token

Now, we introduce *gesture tokens* to enable grammar descriptions that refer to handwritten strokes. A gesture token is instantiated for each handwritten stroke

every time a stroke is drawn. Each token holds an n-best list that the pattern matcher produces. In addition, the token holds information that is derived directly from the stroke, such as the bounding box, the stroke, and the coordinates of the starting point. Table 1 shows the attributes that a gesture token holds. Attribute `pattern` is the n-best list and the others are derived from the stroke.

Gesture tokens can be referred to in the same ways as other kinds of tokens that correspond to graphical objects such as circles and lines. This enables the developer to specify what should happen when a handwritten stroke is drawn, based on the shape of the stroke and the positional relationship between the stroke and other objects, such as graphical objects that are already drawn on the canvas and even other handwritten strokes.

2.3 Rules using gesture tokens

To explain how gesture tokens can be used in grammatical descriptions and how handwritten strokes can be processed, this section presents a simple rule that is described using a gesture token. This rule transforms a handwritten stroke into a circle, as illustrated in Fig. 4, if the pattern matcher determines that the probability that the stroke has a circular shape exceeds 0.5.

```
_CreateCircle::=G:Gesture where (
   findGesture(G,"circle",0.5)
) {} {
   createCircle(G.bound);
   delete(G);
}
```

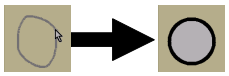

Fig. 4. The transformation of a circular stroke into a graphical object

This rule indicates that `_CreateCircle` consists of a gesture token `G`. `G` should be circular. The user-defined `findGesture(G,N,P)` checks this condition. The function tests whether a given gesture token, `G`, has a candidate named `N` whose probability exceeds `P`. When this constraint holds, the rule creates a circle that is inscribed within the bounding box of the stroke using `createCircle(B)`, where `B` is the bounding box of the circle being created.

Note that the rule deletes the handwritten stroke using `delete(G)`. As a result, the corresponding handwritten stroke disappears from the canvas. Simultaneously, the non-terminal symbol `_CreateCircle` disappears, since its criteria are no longer satisfied. This scheme systematically deletes unnecessary handwritten strokes (`G` in this rule) and pseudo-tokens (`_CreateCircle` in this rule) that are introduced for the description of rules.

3 Context-dependent interpretation

When the user draws a handwritten stroke on a structured diagram editor, the context of the stroke can be classified into the following three categories:

(1) Syntax for the target diagram that the structured diagram editor supports

(2a) Existing tokens around the handwritten stroke that are already recognized as parts of the target diagram
(2b) Existing gesture tokens around the handwritten stroke that are not yet recognized as parts of the target diagram

Since category (1) is the precondition of a structured diagram editor, all the rules assume the context of category (1). A rule that assumes only the context of category (1) can be described by placing only a gesture token as the rule's multiset of symbols P_i. An example of a rule in this category is the rule described in Section 2.3, which creates a circle that forms a binary tree. This rule assumes only the syntax of binary trees. A rule that assumes the context of categories (1) and (2a) should use tokens other than gesture tokens. A rule that assumes the context of categories (1) and (2b) should use two or more gesture tokens. The next two subsections describe examples of these two kinds of rules.

3.1 Interpretation depending on existing tokens

A rule for recognizing handwritten strokes depending on the existing tokens of the target diagrams uses tokens of the syntax other than gesture tokens. As an example of such rules, we show the rule for recognizing a linear handwritten stroke drawn between two nodes like Fig. 5 as an edge between the two nodes on a binary tree editor.

```
_CreateEdge::=G:Gesture,
      N1:Node,N2:Node where (
   findGesture(G,"line",0.3) &&
   inCircle(G.start,N1.cp,N1.r) &&
   inCircle(G.end,N2.cp,N2.r)
) {} {
   createLine(N1.cp,N2.cp);
   delete(G);
}
```

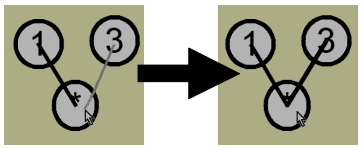

Fig. 5. The interpretation of a linear stroke between two nodes

This rule holds that `_CreateEdge` consists of a gesture token and two nodes. The shape of the gesture token should be linear. The probability of the pattern should exceed 0.3. The stroke should run from the connection point (see Section 2.1) of one node to the connection point of another. This condition is checked by calling the user-defined function `inCircle()`. If these conditions hold, a line object is created between the connection points of the two nodes. Finally, the gesture token is deleted. The line object that this rule creates is then processed using the second rule described in Section 2.1. As a result, a composite node consisting of the three nodes appears on the canvas.

Note that this rule requires very low probabilities, *e.g.*, 0.3 is the threshold for recognizing a linear handwritten stroke between two nodes as an edge. This means that ambiguous strokes sketched roughly by the user can be recognized. This powerful recognition is achieved by referring to the context.

3.2 Interpretation depending on existing gesture tokens

A rule for recognizing handwritten strokes depending on other handwritten strokes uses two or more gesture tokens. As an example of such a rule, we show a rule for recognizing two handwritten strokes, a circular stroke and a stroke inside the circular stroke, as a circle and a text label that form a node of a binary tree, as depicted in Fig. 6.

```
_CreateNode::=G1:Gesture,
      G2:Gesture where (
   findGesture(G1,"circle",0.5) &&
   insideOf(G2.bound,G1.bound)
) {} {
   C = createCircle(G1.bound);
   createText(getString(G2),C.mid);
   delete(G1);
   delete(G2);
}
```

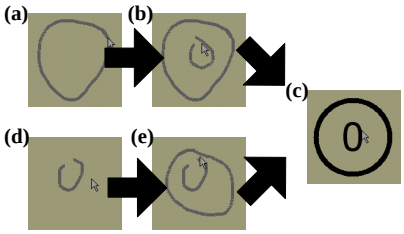

Fig. 6. The interpretation of a circular stroke and a stroke written inside the circular stroke

This rule claims that if two gesture tokens `G1` and `G2` exist, and `G2` is inside `G1`, an inscribed circle in the bounding box of `G1` is created. Moreover, new text is placed at the center of the circle. Function `getString(G)` is used to obtain the text string. It returns a string that corresponds to the textual pattern with the highest score out of the textual patterns in the n-best list of `G`.

Note that this rule enables the user to draw the two strokes in any order. For example, when the user draws the circular stroke first (Fig. 6a), and then the inner stroke (Fig. 6b), the rule transforms the two strokes in the manner depicted in Fig. 6c. The same result is obtained when the user draws the inner stroke (Fig. 6d) and then the circular stroke (Fig. 6e). Therefore, the proposed framework can provide the user with natural recognition.

4 Another Example

This section shows another example of a structured diagram editor that is defined in this framework. Fig. 7 illustrates a network diagram editor that is defined using nine rules. The user can handwrite a network node, a network segment, and a connector that connects the network node with the network segment.

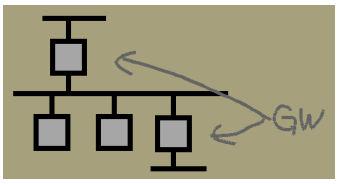

Fig. 7. Another example: a network diagram editor

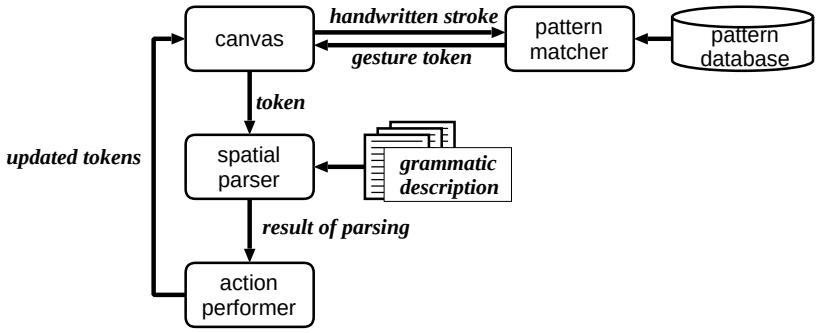

Fig. 8. Schematic diagram of the system

Recognized nodes, segments, and connectors are automatically connected and tidied into squares, horizontal lines, and vertical lines, respectively. Since the constraint solver maintains all connections and alignments, all objects connected to a dragged object are automatically moved for maintaining the structure. Therefore, the user can drag objects while maintaining the syntax of the diagram.

Note that the user can handwrite such an annotation using free strokes such as the "GW" with two arrows shown in Fig. 7, since the framework leaves all handwritten strokes that are not recognized by the given rules.

5 Implementation

Fig. 8 is a system structure that implements the proposed framework. The figure illustrates the components and the flow of data between the components. Below is a description of how each of the components performs:

Canvas feeds the raw data of a handwritten stroke into the pattern matcher, *i.e.*, the coordinates of the sampled points, and receives a gesture token. Moreover, when the state of the canvas changes, *i.e.*, new tokens are created, existing tokens are deleted, or the attributes of the existing tokens are changed, the canvas requests that the spatial parser parse them.

Pattern matcher tries to match a stroke with the patterns registered in the pattern database. The matcher returns the n-best list of the match and the raw data of the stroke as a gesture token.

Spatial parser searches for applicable rules in the grammatical description. If the parser finds one, it applies the rule and asks the action performer to perform the rule's action, if any. If the parser cannot find any applicable rule, the gesture token remains on the canvas for future use. This mechanism enables the recognition of handwritten strokes that depend on other handwritten strokes, as described in Section 3.2.

Action performer executes the action of the applied rule and updates the state of the canvas, if necessary.

We implemented the proposed framework using this system structure. Currently, we have implemented the pattern matcher using the recognizer distributed as a part of SATIN[5]. We used Tcl/Tk, C, and Java for the implementation.

6 Related Work

Several frameworks have been proposed for processing a handwritten stroke depending on its context.

Electronic Cocktail Napkin[6] supports the recognition of the configuration of handwritten strokes in a drawing. The recognizer parses handwritten strokes depending on the context, with user-defined production rules. Therefore, the target application domain is similar to ours. However, the system only allows several built-in gestures. In our framework, the developer can define new gestures by describing rules.

Artkit[7] is a toolkit that supports the implementation of interactions using handwritten input. The toolkit supports handwritten input using *sensitive regions*. A sensitive region has a set of acceptable shapes of handwritten input and algorithms for processing the drawn input. In Translucent Patches[8], the notion of patches corresponds to the context. Each patch defines how handwritten strokes should be processed and displayed. Strokes drawn within the region that a patch covers are processed using the patch. Flatland[9] is an electronic whiteboard system. A context is defined as a *behavior* that defines how strokes are processed and displayed. A handwritten stroke drawn by the user is added to a *segment*, which can have a behavior. This means that the recognition of new strokes is determined by the behavior. Plugging another behavior into the segment can change the recognition of a stroke. The system provides several behaviors. Unlike these systems, our proposed framework has the capacity to define context as a grammatical description.

DiaGen[10] and Electronic Cocktail Napkin provide *syntax-directed editing*, which enables the user to reshape a diagram, while maintaining the syntax of the diagram. This framework can also support syntax-directed editing. We use a constraint solver for this end. Rules can request the spatial parser to maintain the conditions while objects are dragged, once objects are parsed to form a non-terminal symbol. Currently, we use SkyBlue[11] as the constraint solver.

Our implementation can also realize the beautification described in [12] using the capabilities of the constraint solver in *Action*. For example, we can align two leaf nodes to make a composite node on a binary tree editor. This beautification requires only the addition of a command in the *Action* of the second rule in Section 2.1, which constrains the y coordinates of `N1` and `N2` to be the same.

7 Discussion

About recognition The proposed framework does not exclude existing algorithms for recognition and interfaces for resolving ambiguity in handwritten strokes. By incorporating such algorithms and interfaces, it is possible to achieve higher precision in recognition. One example is the incorporation of the n-best list interface for higher precision in character recognition. Then, the developer can define a rule that activates the n-best list interface on nodes of binary trees. The interface will show the possible recognition of the text, asking the user to indicate

an intended choice. When the user selects the choice from the list, recognition of the text is fixed.

About description The framework enables a developer to define powerful rules for recognizing handwritten strokes by utilizing a parsing algorithm that is derived from CMGs. Moreover, the developer can describe both rules for recognition and rules defining the syntax of a diagram in one specification language. However, our current implementation forces the developer to embed some "magic numbers" directly in the grammatical description, such as 0.5 in the rule described in Section 2.3, to define the threshold for recognition. Since such parameters are dependent on the pattern database, the developer may have to re-tune the parameters in the rules when new patterns are registered in the pattern database. We are now investigating how to separate such numeric parameters from the grammatical description to allow developers to tune parameters more easily.

8 Summary

We proposed a framework to support the rapid development of pen-based structured diagram editors that support the recognition of a handwritten stroke depending on its context. The framework uses CMGs to describe the context or positional relationships of handwritten strokes and other objects, which in turn can be used to interpret the ambiguous results of pattern matching and to describe the syntax of target diagrams.

References

1. Golin, E.J., Magliery, T.: A compiler generator for visual languages. IEEE VL'93 (1993), 314–321
2. Costagliola, G., Tortora, G., Orefice, S., Lucia, A.D.: Automatic generation of visual programming environments. Computer **28** (1995) 56–66
3. Chok, S.S., Marriott, K.: Automatic construction of intelligent diagram editors. ACM UIST'98 (1998) 185–194
4. Marriott, K.: Constraint multiset grammars. IEEE VL'94 (1994) 118–125
5. Hong, J.I., Landay, J.A.: SATIN: a toolkit for informal ink-based applications. ACM UIST'00 (2000) 63–72
6. Gross, M.D., Do, E.Y.L.: Ambiguous intentions: a paper-like interface for creative design. ACM UIST'96 (1996) 183–192
7. Henry, T.R., Hudson, S.E., Newell, G.L.: Integrating gesture and snapping into a user interface toolkit. ACM UIST'90 (1990) 112–122
8. Kramer, A.: Translucent patches. JVLC **7** (1996) 57–77
9. Igarashi, T., Edwards, W.K., LaMarca, A., Mynatt, E.D.: An architecture for pen-based interaction on electronic whiteboards. AVI 2000 (2000) 68–75
10. Minas, M., Viehstaedt, G.: DiaGen: A generator for diagram editors providing direct manipulation and execution of diagrams. IEEE VL'95 (1995) 203–210
11. Sannella, M.: SkyBlue: a multi-way local propagation constraint solver for user interface construction. ACM UIST'94 (1994) 137–146
12. Chok, S.S., Marriott, K., Paton, T.: Constraint-based diagram beautification. IEEE VL'99 (1999) 12–19