A Unified Approach for Interpreting Handwritten Strokes using Constraint Multiset Grammars

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

Abstract

A unified approach is proposed to support the rapid development of editors such as pen-based structured diagrams. This approach uses Constraint Multiset Grammars to describe the context or positional relationships among handwritten strokes and other objects. The resulting description can then be used to interpret ambiguous results of pattern matching techniques. We have implemented this approach based on Eviss, a visual system that supports the rapid prototyping of structured diagram editors.

1. Introduction

A *structured diagram editor* is a graphical editor that is tailored to a specific application domain, such as binary trees, state charts, or organization diagrams. Such editors allow the user to draw diagrams quickly within the supported domain. A well-known example is a Microsoft Organization Chart. However, such editors are usually designed only for WIMP interfaces.

The goal of our research is to provide developers with the means to rapidly develop structured diagram editors that are able to analyze *handwriting*. This requires both *methods for describing the target domain,* and *rules for recognizing handwritten strokes*. These rules must consider ambiguity, and be based not only on the shape of handwritten strokes but also on their surrounding context. To achieve our goal, we propose a unified approach using Constraint Multiset Grammars(CMGs)[5]. We have implemented our approach based on Eviss[1], our visual system, which supports the rapid prototyping of structured diagram editors. Figure 1 shows a structured diagram editor for editing computation trees; it was produced using our unified approach.

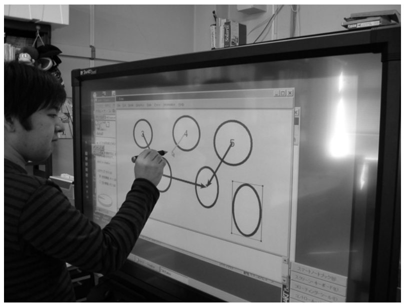
Figure 1. A structured diagram editor generated from its description

2. A unified approach using CMGs

Our unified approach uses graph grammars to describe: 1) the syntax of the target domain, and 2) how handwritten strokes should be interpreted. Raw analysis of various handwritten strokes produces several possible candidates for further investigation. Our unique approach uses grammars to describe the rules for selecting the appropriate candidate from several possible candidates.

2.1. Description of a structured diagram editor using CMGs

A *spatial parser* is required to analyze positional relationships among graphical objects that have been input in a random, unordered fashion. The parser recognizes the structure of graphical objects and then performs follow-up actions based upon its interpretation of the results. Several *spatial parser generators* have been proposed to facilitate the systematic development of spatial parsers (e.g., [4][3][6][2]). A spatial parser generator generates a parser from the specification of the target domain. A *grammar* is used to describe this specification; it specifies the syntax of the domain, such as a domain that has graphical objects that are used as primitives, and the spatial relationships that must be satisfied among the primitives of the domain. By adding supporting functions and user interfaces to the parser that is generated, a complete structured diagram editor can easily be developed.

We use a grammar that is based on CMGs[5]. A produc-

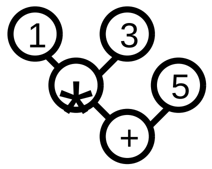

Figure 2. A visual language: computation tree

tion of the grammar has the form:

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

This indicates that the non-terminal symbol P can be rewritten to the multiset of symbols $P_i (i = 1, \cdots, n)$ when the attributes of all of the symbols satisfy the constraint C. P's attributes are assigned in $Attr$. It also executes the action $Action$ after performing the reduction. We have added $Action$ to the original CMGs for convenience.

The two productions below are the specifications for computation trees (see Figure 2), and they are the visual language that is used for providing explanations in the remainder of this paper.

```
Node::=C:Circle,T:Text where(
    close(C.mid,T.mid)
) { cp = C.mid;
    r   = C.radius;
    bound = C.bound;
} { }

Node::=N1:Node,N2:Node,N3:Node,L1:Line,L2:Line where(
    close(L1.start,N1.cp) && close(L1.end,N2.cp) &&
    close(L2.start,N3.cp) && close(L2.end,N2.cp)
) { cp = N2.cp;
    r   = N2.r;
    bound = mergeBound(N1.bound,N2.bound,N3.bound);
} { }
```

The first production indicates that a node consists of a circle and some text. The midpoint of the circle and the midpoint of the 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 the text are reduced to a node. The attribute cp of the node is defined to specify the *connection point*. This connection point is the point at which an edge could possibly be connected within a tolerable error r, which are respectively assigned as the midpoint and radius of the circle. The attribute bound is defined as the bounding box of the circle. r and bound are defined for later use. No action is specified in this production.

The second production defines the composition of the nodes. It specifies that a composite node consists of three nodes, N1, N2, and N3, and two lines, L1 and L2. L1 should start near to the connection point of N1 and end near to that of N2. Similarly, L2 should start near to the connection point of N3 and end near to that of N2. The connection point of the composite node itself is assigned as the connection point of N2. Finally, the bounding box of the composite node is assigned as the rectangle that surrounds all three of the child nodes. mergeBound(B1,B2,...) is a user-defined function that is used to calculate the bounding box.

By providing the spatial parser generator with these two productions, close(P1,P2), and mergeBound(B1,B2,...), a structured diagram editor can be developed that is specifically designed for editing computation trees.

2.2. Handwritten stroke as token

We now introduce *gesture tokens* to enable grammar descriptions that refer to handwritten strokes. A gesture token is instantiated for every handwritten stroke drawn by the user; the token holds information that is derived directly from analysis of the stroke, such as the bounding box of the stroke and the coordinates of the starting point. In addition, the token holds an analysis that a pattern recognizer produces, such as a list of candidate patterns with their appropriate properties and probabilities.

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 upon 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. Below, we give two sample productions that use gesture tokens.

Defining transformations. The production below transforms a handwritten stroke into a circle, as shown in Figure 3a, if the recognizer determines that the probability that the stroke has a circular shape is higher than 0.6.

```
CreateCircle::=G:Gesture where(
    findGesture(G,"circle",0.6)
) {} {
    createCircle(G.bound); delete(G); }
```

This production indicates that a CreateCircle consists of a gesture token G. G's shape should be circular. The user-defined findGesture(G,N,P) checks for this condition. The function tests whether a given gesture token, G, has a candidate named N whose probability is greater than P. When this constraint holds, the production creates a circle object 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 production can delete the handwritten stroke by delete(G). This function removes a token whose name is specified as its argument. As a result, the specified gesture token is deleted and the corresponding handwritten stroke disappears from the canvas. At

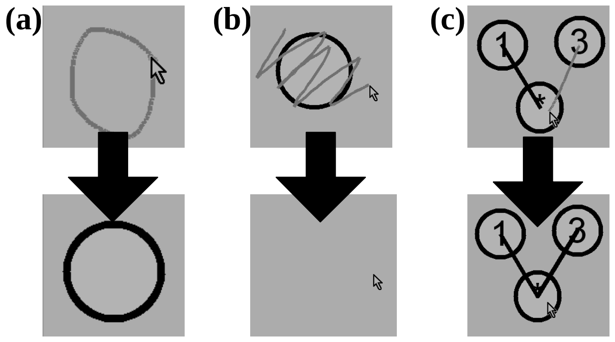

Figure 3. Examples of transformation and gesture

the same time, the non-terminal symbol CreateCircle disappears, since its criteria are no longer satisfied.

Defining gestures. Our second example defines the gesture that erases a circle object by drawing a zigzag shape over the object as shown in Figure 3b.

```
DeleteCircle::=G:Gesture,C:Circle where(
    highestGesture(G,"delete") && touch(G,C)
) {} {
    delete(G); delete(C); }
```

This production indicates that a DeleteCircle consists of a gesture token and a circle that satisfy two conditions. First, the shape of the gesture token should match the pattern whose name is registered as "delete"; highestGesture(G,N) checks this condition. This user-defined function tests whether a given gesture token G has a candidate named N and whether it has the highest probability among all of the candidates. Second, the handwritten stroke and the circle should touch; touch(O1,O2) tests whether the bounding boxes of both O1 and O2 intersect. When the conditions are met, the handwritten stroke and the circle are deleted by the action of the production. The non-terminal DeleteCircle is also erased as a result of the deletion.

3. Eliminating ambiguity by context

Recognition will fail if we examine only the shape of handwritten strokes. Suppose that the developer wants to add the capability of handwriting the edges of computation trees. For example, when the user handwrites a linear stroke between two nodes, the corresponding edge appears as shown in Figure 3c. Registering linear patterns does not work, as the shape of the pattern for the numeral 1 is also linear. However, a human can distinguish an edge from the numeral 1 by recognizing the context in which the stroke is drawn, i.e., by realizing that "a linear stroke connecting two nodes must be an edge", even though the stroke's shape is quite similar to that of the numeral 1. Gesture tokens can be used to describe such a context.

```
CreateEdge::=G:Gesture,N1:Node,N2:Node where(
    findGesture(G,"line",0.5) &&
    inCircle(G.start,N1.cp,N1.r) &&
    inCircle(G.end,N2.cp,N2.r)
) {} {
    createLine(N1.cp,N2.cp); delete(G); }
```

This production claims that a CreateEdge consists of a gesture token and two nodes. The gesture token's shape should be linear. The probability of the pattern should be greater than 0.5. The stroke should be from the connection point (see the productions in Section 2.1) of one node to the connection point of another. This condition is checked by calling the user-defined function inCircle(P,C,R). The function tests whether the given point P is within the circle with center point C and radius R. If so, a line object is created between the connection points of the two nodes. Finally, the gesture token is deleted.

Note that a developer need only add the above production to the productions in Section 2.1 to implement the capability for handwriting the edges of computation trees. Therefore, gesture tokens allow us to declaratively describe what should happen for a handwritten stroke, according to the stroke's shape and the context in which it is drawn. In addition, this technique maintains the modularity of the description.

4. Summary

We propose a unified approach to support the rapid development of pen-based structured diagram editors. Our approach uses CMGs to describe the context or positional relationships among handwritten strokes and other objects, which in turn can be used to interpret the ambiguous results of pattern matching as well as to describe the syntax of target diagrams.

References

[1] A. Baba and J. Tanaka. Eviss: A visual system having a spatial parser generator. *Proc. APCHI'98*, pp. 158–164, 1998.

[2] S. S. Chok and K. Marriott. Automatic construction of intelligent diagram editors. *Proc. ACM UIST'98*, pp. 185–194, 1998.

[3] G. Costagliola, G. Tortora, S. Orefice, and A. D. Lucia. Automatic generation of visual programming environments. *Computer*, 28(3):56–66, 1995.

[4] E. J. Golin and T. Magliery. A compiler generator for visual langauges. *Proc. IEEE VL'93*, pp. 314–321, 1993.

[5] K. Marriott. Constraint multiset grammars. *Proc. IEEE VL'94*, pp. 118–125, 1994.

[6] M. Minas and G. Viehstaedt. Diagen: A generator for diagram editors providing direct manipulation and execution of diagrams. *Proc. IEEE VL'95*, pp. 203–210. 1995.