An Interactive System for Recognizing Hand Drawn UML Diagrams

Edward Lank, Jeb S. Thorley and Sean Jy-Shyang Chen
Department of Computing Science
Queen's University

Abstract

Diagrams are widely used by software engineers to capture the structure and organization of software systems. The Unified Modeling Language (UML) is a commonly-used notation for such diagrams. We have designed and implemented a system for the on-line recognition of hand drawn UML diagrams. Input comes from an electronic whiteboard, a mouse, or a data tablet. A sophisticated segmentation algorithm groups pen strokes into symbols, coping with drawing inaccuracies that are common in hand drawn input. The system is organized around a retargetable kernel which provides a general front end for on-line recognition of any iconic notation. The kernel is extended with UML specific enhancements to segmentation, as well as UML specific glyph recognizers. A simple and intuitive graphical user interface allows the user to correct segmentation and recognition errors. Relatively messy freehand UML drawings are interpreted properly.

1 Introduction

On-line recognition of diagram notations is an active area of research, with work being done on mathematics notation [19, 5, 22, 16, 23], architecture drawings [21, 8], engineering drawings [11, 10, 13, 18, 12], and free-hand sketches of shapes and characters [4, 3, 14, 7, 15]. In parallel to this research, many software programs are available to create well-formatted, easily editable diagrams. Such systems include latex for math and typeset notations, CAD programs for engineering drawings, and various free-form drawing tools.

The unified modeling language (UML) is a diagram notation designed to model large systems, particularly large object oriented software systems. Computer Aided Software Engineering (CASE) tools to model systems with the UML include Rose (Rational), Together (Object International), Innovator (MID), Advance (IC&C), Select (Select), Forte (Forte), Object Engineering Workbench (Innovative Software), and many others. These CASE tools, while beneficial in system design, have two drawbacks. First, developers are relegated to working largely independently on separate computers. Second, developers are forced to learn to use CASE software, rather than being permitted to draw UML diagrams free hand.

Smartboards[1] present an opportunity to remedy the first problem, the requirement of independent work, by allowing users to interact collaboratively. UML diagrams can be drawn on a whiteboard by a group of software designers. As well, smartboards can be attached to a network, allowing groups at geographically separate locations the opportunity to collaborate. In this way, smartboards complement CASE tools, in providing an alternative means of creating UML diagrams to model software systems. An initial UML diagram can be created by a group of software designers on a smartboard and then imported into a CASE tool for further refinement.

The way in which a user must interact with a tool determines how it will be used. As noted by Arvo [2], in order for computers to better aid human problem solving, users must be able

[1] A Smartboard is an electronic white board that may be connected to a computer to collect user input and display output

to enter information without being forced to adhere to unnatural constraints. He notes that even light weight interaction requirements as in a gesture based system[2], force the user to first think about how to enter their information into the system, and only secondly about the actual problem. Therefore an important first step in designing a distributed, collaborative environment for UML design is the ability to recognize hand-drawn UML diagrams. A user then need only know the UML in order to use the system.

We have implemented a system which recognizes hand-drawn UML diagrams. The system is implemented in Java, and accepts input from an electronic whiteboard, a data tablet, or a mouse. The system currently recognizes class diagrams, use case diagrams, and sequence diagrams. It incorporates a third party character recognizer for hand drawn text implemented in C++ [1].

Before describing the UML recognition system, this paper provides some background on diagram recognition, on-line recognition systems, and the UML.

Our UML tool incorporates a UML recognition system and a set of features to allow user correction of recognition errors. The UML recognition occurs in stages. Initially, a software drawpad captures sketched input. When the user invokes recognition, a retargetable segmenter first performs domain-independent segmentation and character recognition. Next, UML glyph recognizers recognize those glyphs specific to UML notation. Finally, a domain specific segmenter refines the segmentation and recognition of UML glyphs based on domain knowledge. The recognition correction features allow intuitive handwritten editing of the segmentation and recognition results.

2 On-Line Diagram Recognition and the Uniform Modeling Language

This section provides background by describing existing work in diagram recognition. The section concludes with a brief introduction to the UML. Glyphs from Class diagrams, Use Case diagrams, and Sequence diagrams are described.

2.1 Computer Recognition of Diagram Notations

Computer recognition of diagram notations is an active area of research in computer science. Computer diagram recognition systems can be characterized based on whether the systems accept on-line input in an interactive fashion from the user, or perform image processing on scanned paper documents obtained off-line. Research into creating practical computer recognition systems for the on-line and off-line recognition of diagram notations such as math notation [19, 5, 22, 16, 23], music notation [20], engineering drawings [11, 10, 13, 18, 12] and the UML [9, 18] is ongoing.

Table 1 describes a multi-level view of diagram recognition proposed by Lank [17]. The multi-level view of diagram recognition applies equally well to on-line and off-line systems. No ordering is assumed on the levels in the

Level	Elements Recognized
Semantics	Meaning conveyed by the diagram is obtained
Entity	Meaningful groupings of symbols
Structure	Meaning from relative spatial placement of symbols
Syntax	Symbol identity is constrained based on inter-symbol relations
Symbol	Characters, symbols, glyphs are identified
Primitive	A line, a dot, a bounding box for a character or glyph, a set of strokes which represent a symbol
Image acquisition	Binary or grey scale image, or a set of strokes

Table 1: The multiple levels of recognition possible in diagram recognition systems.

[2] A gesture based system is a system with a set of defined movements to represent symbols and actions

model. For example, in mathematics recognition, systems have performed structural recognition without considering the underlying syntax of the equations [23]. As well, all levels need not be present in a computer diagram recognition system. For example, optical character recognition (OCR) systems are geared toward the recognition of text. OCR systems can therefore assume a structure for the text and work to extract that predefined structure rather than being required to recognize the structure of the glyphs.

Our prototype system currently performs image acquisition, obtains primitives, and recognizes symbols. We are working on extending our system to perform semantic recognition of UML diagrams.

2.2 On-line Recognition Systems for Diagram Notations

Existing systems for on-line recognition of math and engineering diagrams, and recognition of line figures include [5, 22, 10, 15, 16]. The Knight Project [6] is a related project that has created a gesture based system for entering and editing UML notation on a smart whiteboard. Their web page mentions a hand drawn mode, but provides no information on this mode. We are not aware of any existing systems for the on-line recognition of hand drawn UML diagrams.

2.3 The Unified Modeling Language

The Unified Modeling Language is designed to model large software systems, particularly large object oriented software systems. Developed by Booch, Jacobson, and Rumbaugh, the UML's goal is to flexibly model the various aspects of systems, from specification through development to deployment [9].

The UML is organized around nine different diagram types: Class diagrams, Use Case diagrams, Sequence diagrams, Object diagrams, Collaboration diagrams, State diagrams, Activity diagrams, Component diagrams and Deployment diagrams. A complete description of UML, including all diagram types and variants, is outside the scope of this paper. Our on-line recognition system currently recognizes Class diagrams, Use Case diagrams, and Sequence diagrams. The user is prompted to select one of these diagram types when the system starts. For more information on the UML, the interested reader is referred to [9, 18]. We assume the reader is familiar with object oriented terminology.

Class diagrams depict the static relationships between classes and interfaces in an object oriented software system. They are the most common of the UML diagrams in system analysis and design, as an understanding of basic system relationships is essential to the construction of modular, robust, object oriented systems. Figure 1 depicts the glyphs we recognize in Class diagrams. While not an exhaustive list of all glyphs contained in class diagrams, we do support relationships, class boxes, and interfaces. Our system also supports class boxes containing attribute, method and responsibility sections.

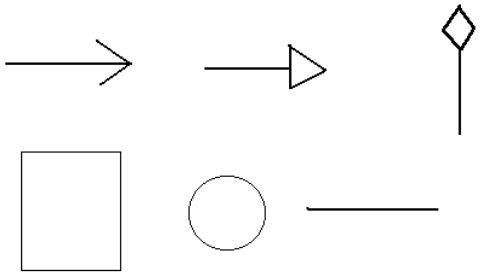

Figure 1: Class Diagram Glyphs

Use Case diagrams depict the actors external to the system, and the services the system provides to users of the system. Actors may be human users, but may also include other software systems, or external environmental factors. Use cases are those services the system provides to the actors. Figure 2 depicts the glyphs used in Use Case diagrams.

Sequence diagrams depict a task performed by the system, for example the realization of a Use Case. They depict both the actor which invokes the task and the classes involved in realizing the task. Figure 3 depicts the glyphs contained in a Sequence diagram.

3 Assumptions

3.1 The Main Challenges

There are several fundamental challenges in recognizing online diagrams. First the strokes which represent a single glyph must be identified (segmentation). Next, text must be distinguished from graphics symbols. Finally both text and graphics symbols must be recognized. Performing these operations is complicated by a number of factors. Symbols may overlap or entirely enclose one another, which makes segmentation more difficult. Since online input is often handwritten, messy handwriting must be dealt with. Messy handwriting can introduce gaps between strokes that compose a single character which, again, complicates segmentation. Handwritten input may also cause the characteristics of glyphs to be very different from user to user, so the metrics for recognition must be robust.

Designing a system for an intelligent whiteboard adds additional challenges. First, we cannot resort to keyboard input. The user must be able to input any and all information through pen strokes. Second, interaction with the whiteboard should be as similar as possible to interaction with a standard whiteboard. To aid in serendipitous input, we do not want to resort to a gesture based system or incorporate a virtual keyboard. The goal of our system is the recognition of UML diagrams drawn freehand by users. Finally, any correction features which we incorporate into our system should not impede the actual creation of diagrams, or restrict interaction. We accomplish this by allowing the user to choose when recognition should occur. In this way, a user can draw freely on the whiteboard, without the constant need to edit the recognition results.

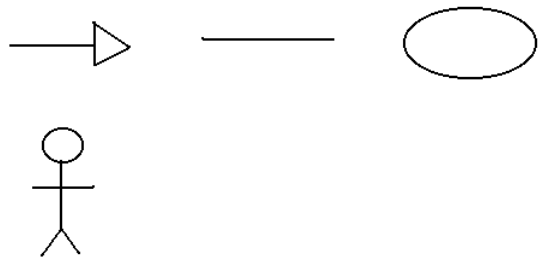

Figure 2: Use Case Diagram Glyphs

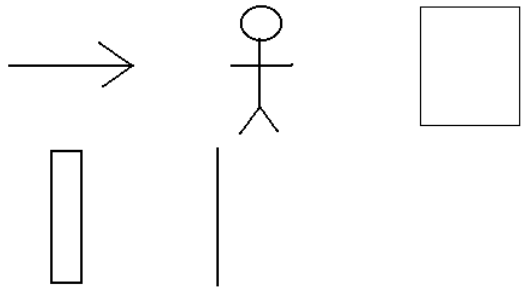

Figure 3: Sequence Diagram Glyphs

3.2 Assumptions

The goal of the intelligent whiteboard project is to design systems for on-line input of domain specific diagrams. By recognizing and encoding characteristics of on-line input in general, as well as the specifics of the diagram type being worked with, we are able to make assumptions which assist in meeting the challenges of online diagram recognition. Due to the coupling of a retargetable front end with domain specific refinement, we are able to achieve highly accurate recognition results. As this is an online system, the few remaining mistakes can be corrected by the user.

3.2.1 Domain Specific Assumption

The intelligent whiteboard software is designed to be retargetable. The stroke collection and retargetable segmentation are not domain specific. They may be incorporated as the front end of any on-line iconic diagram recognition system. The final stages of the recognition are domain specific.

Combining a general front end with a domain specific back end allows us to gear later stages of segmentation and recognition to a particular domain without an overall loss of generality or retargetability. By having later stages of the recognition work specifically with one diagram type, we are able to have more directed segmentation and recognition. UML is particularly well suited to this approach as its relatively small set of UML glyphs may be further

divided by diagram types (e.g. Class vs. Use Case). Each diagram type has their own recognizer. Consequently, the set of symbols that each recognizer deals with is quite small, and is composed of characters with distinct features.

3.2.2 Online Assumptions

The diagrams we recognize are produced online. This allows us to make two assumptions. The first is that we have temporal information for each stroke. The second is that we can have user correction of all segmentation and recognition results. In making these assumptions, we specialize our system for the on-line recognition of diagrams. Our techniques do not work with scanned images, as we deal with the strokes entered by users rather than the individual pixels in an image.

Knowing when each stroke occures relative to every other stroke in the diagram is key to our segmentation method. We assume glyphs are built from sequential, intersecting strokes. Only temporal neighbours to the partially constructed glyphs are considered for intersection tests and addition to the Glyph. This helps us overcome the challenge of character overlap and containment. For example, the user may continue the labeling of a class box over the box's edge, without causing the labeling characters and the class box to be combined during segmentation. In practice, this assumption has facilitated very accurate segmentation of diagram primitives.

Our second assumption, that any segmentation or recognition errors will be corrected by the users before the diagram recognition is complete, means that even though no algorithm will obtain perfect segmentation or recognition every time, we can still assume perfect output from our system.

4 Image Acquisition and Initial Primitive Extraction

4.1 Image Acquisition

Users create diagrams by drawing glyphs on the software drawpad. The drawpad is not domain specific to the UML. It collects a set of points which it groups into strokes based on mouse-pressed and mouse-released events. Each stroke is collected by the drawpad as it is drawn, and timing information is also inserted in the stroke. Timing information is important to our segmentation algorithm, due to our assumption that only sequential intersecting strokes combine to form glyphs.

White board, mouse, and data tablet is collected by capturing each stroke as a set of points. When the user is ready to invoke the recognizer on the diagram, the set of strokes is collected from the drawpad and the retargetable segmenter is invoked on the collected strokes.

4.2 Initial Primitive Segmentation

The first stage in user-invoked segmentation is to process the data by using a retargetable segmenter.

The retargetable segmenter receives a set of strokes from the drawing pad, and groups them into glyphs. Stroke information is a list containing a start time and a series of points. Strokes are examined in the order in which they were entered on the drawpad. The timing information is used to determine stroke ordering. All further segmentation is based on testing for the intersection of sequentially drawn strokes. Smithies et. al. [19] propose a segmentation algorithm based on sequential strokes, but use the probabilities of character recognition to determine segmentation.

Testing for intersection in our retargetable segmenter occurs in two stages. In the first stage, we calculate a stroke's bounding box, and test for bounding box intersections. Next, for any strokes that pass the bounding box check we divide the stroke into line segments and perform a line intersection check. Checking for bounding box overlap is computationally more efficient than checking for line segment intersections. Testing line segments for intersection is only necessary if bounding boxes overlap.

Stroke checks must test for approximate intersection of strokes, which may have arbitrary shape. Strokes are extended before testing in-

tersection; this allows for small gaps that commonly occur in hand-drawn diagrams. In the event of an approximate stroke intersection the stroke is added to the glyph. If no approximate stroke intersection occurs then the glyph is considered finished and a new glyph is started with the current stroke as its first stroke.

For purposes of the intersection test, strokes are represented as collections of line segments. This allows us to make use of Java's excellent Line2D API. A stroke consisting of N points is transformed into a sequence of N-1 line segments. Two strokes are tested for intersection by exhaustivly checking each of the segments of the first stroke against each of the segments of the second.

As stated, the segment intersections are performed using the standard Java2D API. However, the strokes themselves are modified to reduce sensitivity to noise and gaps. First we attempt to minimize pen/mouse down and up noise. This noise is introduced when a user puts their pen or mouse down to begin drawing a stroke and again when they lift it upon completion of a stroke. The direction of movement at the ends of a stroke are often completely unrelated to the overall direction of the stroke. To compensate for this, if the stroke is composed of a sufficient number of points, one point from the beginning and one point from the end of each stroke are removed.

Next the start and the end of the strokes are extented slightly. Line extension also allows for the recognition of characters that are composed of strokes which do not quite intersect, for example a T whose base does not intersect its crossbar. It also compensates for the points that were lost in the pen/mouse up and down noise removal.

To implement the extension, the dominant direction at the ends of a stroke is calculated based on the last few line segments. Each end of the stroke is extended by a line segment that is oriented in the dominant direction of the stroke end. The length of these extension lines is tunable; we find that a value of 5 to 10 pixels works well. The line intersection test is then applied to the modified strokes.

Experience shows that this segmentation method performs well. It is attractive to use because of its simplicity, speed of execution, and generality. This grouping method is in no way domain specific. It is able to a do a rough general grouping for any domain. In later processing, a domain specific segmenter is used to refine the results. In this way we are able to combine the benefits of retargetable code with domain specific performance tuning.

5 Refinement of Primitives and Symbol Recognition

After the initial recognition by the retargetable segmenter, recognized characters are sent to the UML segmenter for UML specific recognition and segmentation. Passing data from the retargetable segmenter to the UML segmenter is mediated by a filter. The filter serves the dual purpose of translating grouped strokes into the required data format as well as passing them to the appropriate recognizers. The data set from the filter is then transferred to the UML segmenter where it undergoes glyph classification, recognition, and refined segmentation. In this section, we will describe the function of the filter, the glyph classification and recognition algorithms and the refinement of segmentation.

5.1 Filter

The filter translates the retargetable segmentation results into a data format which is compatible with the rest of the system. The filter is the first domain specific stage in the recognition chain. The filter can perform two functions. If the retargetable drawpad and segmenter were to serve as the front end to a third party system, our data format would need to be translated to that of the third party system. As well, recognizers have glyph sets that are, at least partially, domain specific. The filter can therefore prepare the grouped strokes for higher-level recognition, or it can pass them to the appropriate recognizer before returning them.

In the UML implementation the filter does both. Among other changes an initial character recognition is required to convert the grouped strokes to the appropriate format for our UML segmenter. The filter treats all glyphs as text

characters and recognizes them using the third party character recognizer. The entire data set is translated to the datatypes used throughout the rest of the system. The dataset is passed on to the the UML segmenter.

5.2 UML Glyph Identification

5.2.1 Histogram

The UML segmenter refines recognition and segmentation results using domain specific information. The first stage of the UML Segmenter constructs a histogram of the glyph sizes. The size used for this calculation is the maximum of a glyph's bounding box's width or height. The mean of the smallest half of these sizes, refered to as the character-set mean, and the standard deviation for the diagram are calculated from the histogram.

5.2.2 Glyph Classification

The next step in the UML segmenter is to classify the glyphs as UML glyphs or characters based on the histogram information. A glyph's size difference from the character-set mean, measured in standard deviations, is calculated. This value is used to classify the glyph.

The size distribution of the glyphs in a standard UML diagram is bimodal, with the character-set mean falling within the first mode. This is because UML glyphs are typically much larger than the characters used to label them. Multiple characters are usually associated with each UML glyph, which causes the first mode to encompass a much larger set of symbols than the second mode. This means that the character-set mean will fall within the first mode. Because of this distribution, if a glyph is one or more standard deviations larger than the character-set mean it can be classed as UML with a great deal of certainty. In fact, we have yet to observe a character misidentified as a UML glyph. Infrequently, small UML glyphs will be classified as characters.

5.2.3 UML Glyph Recognition

Once the glyphs have been classed as UML or characters, a refinement of the recognition is applied to the UML glyphs. Any glyph that has been classed as a UML symbol is re-recognized using a diagram specific recognizer. The diagram type is supplied by the user when they open the Editor.

Of the three diagram types with which we are working, Class diagrams present the most challenges to recognition. As may be seen from figure 1, figure 2 and figure 3, Class diagrams have a larger set of symbols than do Use case and Sequence diagrams. Recognition of symbols within this diagram type is further complicated by the similarities between the symbols used to denote relations between classes (i.e. generalization, containment and dependencies). Because of its relative complexity, we will describe our Class diagram symbol recognizer in detail. Our symbol recognizers for Use Case and Sequence diagrams contain many of the same features, but are somewhat simplified due to the more restricted glyph set associated with these diagrams.

The number of strokes contained within a glyph is a useful guide in determining which tests to apply, and in what order to apply them. For example, it is very unlikely that an interface (a circle) would be drawn with two or more strokes. For this reason testing a two stroke glyph to see if it is an interface is a low priority. For glyphs containing three or more strokes the interface test is not applied at all. Likewise, it is far more likely that a one stroke glyph will be an interface than a containment relationship (diamond arrow), so the order in which the tests are applied reflects this assumption.

A set of tests which employ distance metrics is used to recognize non-arrow symbols in this recognizer. The non-arrow symbols are quite distinct, so a distance metric works well. First, the total stroke length of the glyph is compared to the perimeter of its bounding box. If they are within a tunable distance (10% for example), of one another this suggests that the glyph is rectangular and it is recognized as a Class box. Similarly the total stroke length is compared to π multiplied by the maximum of the bounding box's width and height to detect a circle. The final distance metric is for a line, and uses a diagonal line across the bounding box as its measure.

The total stroke length to bounding box perimiter ratio of arrow types is quite differ-

ent to that of other characters. This allows us to classify a symbol as an arrow using a distance metric. However, due to the similarity between arrows, a distance metric is not an effective means for futher classification. More involved calculations must be used to distinguish between the arrow types.

After a glyph is determined to be an arrow, the next task is to determine which strokes represent its head. This is best accomplished by determining which stroke represents its body, and letting all other strokes represent its head. A body stroke test can be implemented as a simple find maximum operation using the maximum of stroke height and width as the comparison attribute. Once the head strokes have been identified, a series of tests are used to implement a decision tree to recognize the arrow.

The first head type to either identify the arrow as, or eliminate as a possibility is an open arrow. The open arrow check is performed thus:

1. Construct a bounding box for the strokes that represent the arrow head.
2. Draw two lines, an X-parallel and a Y-parallel, through the box. The X-parallel should be drawn at 1/2 the height of the heads bounding box. The Y-parallel should be drawn at 1/2 the box's width. See figure 4 for illustrations of the parallels.
3. Count the number of times the X-parallel line intersects a head stroke, and the number of times the Y-parallel line intersects a head stroke.

As can be seen from the figure 4, in all but the case of the open arrow head both the X-parallel and Y-parallel lines will intersect a head stroke at least twice. If either line has fewer than two intersections, the arrow can be labeled as an open arrow. If two intersections do occur, an open arrow may be eliminated as a possibility for the primary label of the arrow.

The next check uses the pythagorean theorem, and either selects or eliminates the diamond arrow type. The pythagorean check is implemented thus:

1. Divide the bounding box of the head strokes into four quadrants
2. Go through the points contained in each of the head's strokes, assigning each point to the quadrant that contains it.
3. Construct a stroke for each quadrant composed of the points which were assigned to it in the previous stage.

As may be observed in figure 5, a perfect diamond will form a right angle triangle in each of the quadrants (using the quadrant walls as "a" and "b", and the diamond stroke as "c".) Using the pythagorean theorem we can check the stokes that we have created for each of the quadrants to see how closely they conform to this hypothesis. Due to messy hand writting, it is likely that a diamond will vary by up to 40% from the expected length of a perfect diamond stroke in one or more of its quadrants (see second image in figure 5). However, since we are testing in four quadrants, we may allow for considerable variance, and still have reasonable confidance that a closed arrow will fail the test in at least one of the quadrants, as in the third image in figure 5. (Hand drawn closed arrows often leave one of the quadrants almost completely empty.) If an arrowhead passes the pythagarean test in each of the four quadrants it is labeled as a diamond. If it does not then, by process of elimination, we may assume that it must be a closed arrow.

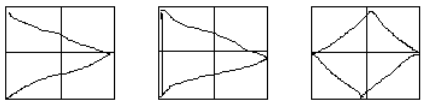

Figure 4: Intersection Test Applied to Open, Closed and Diamond arrow heads, respectively

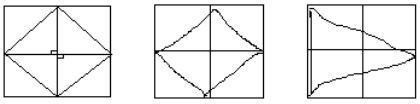

Figure 5: Pythagorean test applied to an ideal diamond, a handdrawn diamond and a closed arrow. Clearly the closed arrow fails the test in at least one of its quadrants

5.3 Refining Segmentation

After the recognition of symbols is refined, a domain specific refinement of segmentation is performed. The set of glyphs representing the diagram are divided into UML symbols and characters. As characters are not specific to the domain of UML there is no domain specific knowledge to apply to their resegmentation. Domain specific knowledge is applied to the resegmentation of the set of symbols labeled as UML glyphs.

5.3.1 Character Segmentation Refinement

There is little resegmentation applied to the symbols labeled as characters. An intersection test similar to that applied in the retargetable segmenter is applied. If the bounding boxes of two neighboring character glyphs overlap, all of the strokes of the first glyph are checked for intersection with the strokes of the second glyph. In the event of an intersection the two glyphs are combined and re-recognized. This method will correct mis-segmentation that may occur if a character is composed of three or more strokes drawn such that stroke i does not intersect stroke (i+1), but stroke (i+2) intersects both i and (i+1). See figure 6 for an example of an H drawn in this way.

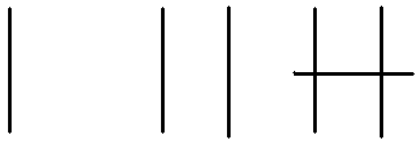

Figure 6: An H drawn in such a way as to need refined character segmentation

5.3.2 UML Segmentation Refinement

UML glyphs are often missegmented by the retargetable segmenter. For example, lines representing associations between UML glyphs are often drawn sequentially with at least one UML glyph, and intersect both glyphs. Correcting this segmentation is an important part of glyph recognition.

As the figures of glyphs in Section 2 demonstrate, the various UML diagrams have different glyph sets. Due to different glyphs being present in different diagram types, the refinement of UML glyphs is diagram specific.

Class diagrams

In Class diagrams, glyphs can be composed of one or several strokes. Class boxes are rectangular, and interfaces are circular. Relationships between glyphs (associations, generalizations or dependencies) are lines or arrows. In observing users drawing glyphs, class boxes were drawn as one, two or three stroke glyphs, and interfaces were uniquely drawn with one stroke. We therefore examine each stroke in multi-stroke glyphs to determine whether it defines a rectangle or circle. We then examine two and three stroke groups in multi-stroke glyphs to determine whether the strokes define a rectangle.

In a one stroke rectangle or circle, the start and end points should be relatively close. We assume that the start and end points should be no further away than one third of the longest of the bounding box's width and height. This allows a user to extend the end point of a glyph stroke beyond its start point. Some bypass is natural in free hand drawing, but the points should be relatively close, unlike lines and arrows, where the start and end points should be at opposite ends of the glyphs. We then use a distance metric to determine the variation between bounding box length and the stroke length. If the stroke is approximately the same size (90% to 115%) as the bounding box's perimeter, we conclude that the stroke defines a rectangle, and split it into a new glyph. Any preceding and succeeding strokes are split into additional glyphs, and the original multi-stroke glyph is removed from the diagram.

If the glyph fails the rectangle check, we check for a circle, noting that the bounding box of a circle should be roughly square, and that the circumference of the circle should be approximately three to three and a half times the width or height of the bounding box. If the stroke fulfils this characteristic, it is split and recognized as an interface.

We then examine remaining multi-stroke glyphs for two and three stroke rectangles. In a two stroke rectangle, the start points of both strokes and the end points of both strokes

should be near one another. In a three stroke rectangle, start and end points of the various strokes also coincide. We test for closeness using the same metric as was used for one stroke glyphs. Arrows should fail the point closeness tests. Given these characteristics, we once again examine the perimeter to determine whether the stroke grouping might define a rectangle, and if it does we reclassify the set of strokes as a class box. Once again, preceding and succeeding strokes define separate glyphs, and are passed to the recognizer.

Care must be taken when examining strokes. Diamonds, in particular, can often be misrecognized as circles. In practice, their length is similar to the length of a circle contained in the same bounding box. To avoid this, we use the histogram data collected to separate UML glyphs from characters. We use the histogram to determine if the stroke group's bounding box is large enough to be a UML glyph, and only if it is do we apply circle and rectangle test. In practice, this seems to work well. We have yet to have the test fail, though we haven't explicitly tried to "fool" the system by drawing inclusion diamonds much larger than is standard in UML diagrams. If the system fails and segments these diamonds as interfaces, user correction can easily be done using the correction features for segmentation described in section 6.2.1.

Use Case diagrams

In Use case diagrams, use case ellipses often overlap associations and generalizations. To separate these glyphs we test for ellipses, and, if an ellipse exists, we split the glyph into a use case and glyphs containing preceding and succeeding strokes. Ellipses are drawn, generally, as one stroke.

To test for ellipses, we once again test start and end points to determine whether they are close. If they are, we check for a bounding box for the stroke with its width greater than its height. If these characteristics are satisfied, we look for a perimeter greater than twice the bounding box's width.

The histogram data is again used to test only those strokes which might be large enough to be a UML glyph. The histogram test eliminates from consideration the heads of actors, which, in some circumstances, might fulfil the ellipse test, and the closed arrows on generalization relationships.

Sequence diagrams

Sequence diagrams have many rectangles interconnected with lines, and separating these is important. To accomplish the refinement of segmentation we once again test for rectangles. Any rectangles are eliminated and reclassified as either Class boxes or activity boxes for classes. The rectangle tests are described in the segmentation refinement for class boxes.

The histogram data is important in the refinement of Sequence diagram segmentation. When excluded, we noted that the actor was often broken up into multiple class boxes. The histogram ensures that the smaller strokes used to create actors are not split during the refinement of segmentation.

6 User Interaction

6.1 Successive Refinement during Recognition

It is important to note that although the recognition is performed in stages, no part of the glyph information is considered static at any stage unless the user has corrected it. Once the retargetable segmenter has done a simple grouping of the strokes, they are translated by the filter into the appropriate format for the UML segmenter. Any of the segmentation that the retargetable segmenter has performed may be changed by the UML segmenter and recognizer. In turn any of this revised information may be changed by the user. The system does not refine user corrections.

6.2 Editing Features

The UML Recognizer supports two editing modes. Segmentation and glyph recognition are both subject to errors. As well, our assumptions (for example, assuming that only sequential strokes form characters) may sometimes be incorrect. Allowing the user to correct segmentation and recognition errors solves this problem. As well, allowing user correction of segmentation and symbol recognition allows us to assume perfect output from our system.

To perform editing, we change the mode of our UML reconizer from drawing mode to one of two editing modes: segmentation editing, or glyph editing. The user changes modes with the radio buttons visible in the lower right hand corner of figure 7.

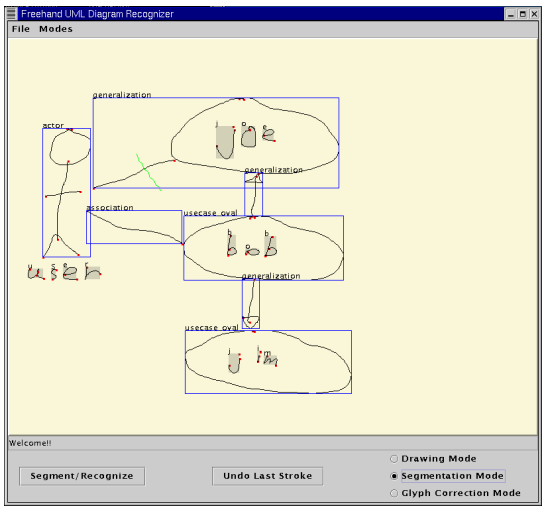

Figure 7: Screen Shot 1

Segmentation editing allows correction of segmentation errors. Glyph editing allows correction of errors in our glyph (UML or character) recognizers. Glyph editing also allows a user to delete entire glyphs.

6.2.1 Segmentation correction

Two types of segmentation errors occur. A glyph may be over-segmented, meaning that a glyph has been misrecognized as two distinct glyphs. A glyph may also be under-segmented, meaning that two or more glyphs have been misrecognized as one glyph. Over-segmented glyphs are corrected with a "join" operation, and under-segmented glyphs are corrected with a "split" operation.

Smithies et al. [19] present a segmentation correcting feature in their math recognition system which is both flexible and simple. They allow a user to "scribble" over a stroke or set of strokes and join them together into glyphs. Strokes which are intersected by the scribble (edit stroke) are grouped.

We make explicit the join and split operation rather than performing implicit splits via joins. In our system, when a user switches to segmentation edit mode, their pen colour is changed to green, and their stroke is an edit stroke which is not added to the diagram. See figure 8 for a visible edit stroke at the left side of the image.

Their edit stroke is considered complete on pen up. After a user draws an edit stroke, the system prompts them to choose whether to join glyphs, split glyphs, or to cancel (if the edit stroke was incorrect). The user selects an option and, as appopriate, glyphs are split or joined. See figures 8 and 9 for an example of a split operation.

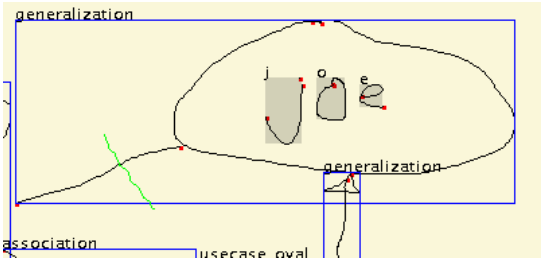

Figure 8: Glyph prior to user segmentation (note edit line on left side)

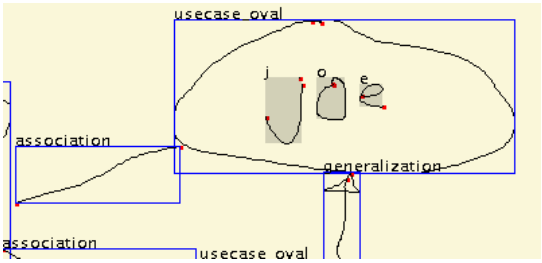

Figure 9: Result of user segmentation

To split or join a glyph, the system must first determine which glyphs are affected. To do this, the editing line is collected in the same manner as strokes are collected when drawing a UML diagram. Glyphs containing strokes that intersect the edit stroke are flagged and collected for editing.

There are two restrictions on editing segmentation: when joining glyphs the user must select at least two glyphs to join, and when splitting glyphs the user can only scribble over one glyph with his or her edit stroke. The first restriction is obviously a requirement. You need

at least two glyphs to join glyphs. We feel the second restriction is reasonable for two reasons. First, after each edit stroke, the user is prompted for which operation is desired, a split or a join. The system is therefore only dealing with one edit stroke at a time. Second, all editing of segmentation errors can be performed by some combination of splitting one glyph and joining multiple glyphs. A moment with a pen and paper can convince you of this fact.

6.2.2 Editing glyph recognition

The best character recognizers and UML glyph recognizers will occasionally make recognition errors. Some facility for user correction of recognition is needed. As well, when drawing freehand diagrams, users will sometimes wish to erase glyphs in the diagram. Both these operations are combined in glyph editing mode. Again, changing this mode requires only a single click from the user.

To edit glyph identity in glyph editing mode, the active mouse listener listens in the drawpad for a mouse clicked event. When the mouse is clicked, the diagram is searched for glyphs whose bounding boxes contain the current mouse position. If more than one glyph's bounding box contains the point, we assume that the user wishes to edit the glyph with the smaller bounding box. This assumption would be dangerous in notations where extensive overlapping of glyphs occurs. However, in UML notation, characters rarely completely overlap, and UML glyphs are much larger than the characters contained in them. This assumption has worked well in practice.

When the glyph is selected, the system allows the user to correct recognition or delete the glyph. The top five recognition choices for the glyph are displayed. The user can select from the list of five recognition choices. See figure 10 and figure 11 for an example of user correction of glyph identity. If the label which the user wishes to select is not present in the top five symbols from the recognizer, an exhaustive set of glyphs can be shown using the "Hand Label Glyph" button.

Recall that our smart segmentation algorithm used a histogram to separate the glyphs into UML glyphs and character glyphs. The

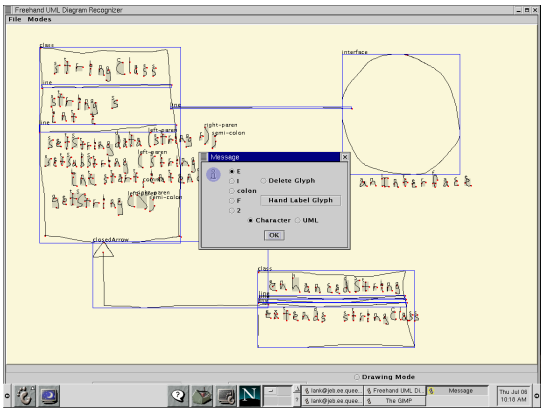

Figure 10: User correction of glyph recognition.

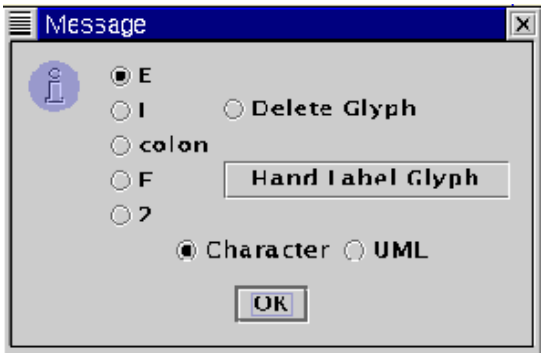

Figure 11: Close-up of Recognition Correction Dialogue

system also allows the user to correct the system's recognition of glyph type using the radio buttons at the bottom of the edit window in figure 11.

7 Conclusion

We have designed a system which recognizes hand drawn UML diagrams. Characteristics of our system, which will be demonstrated at CASCON 2000, include:

- A drawing pad which collects domain independent stroke information from the user.
- A retargetable segmentation algorithm which performs an initial primitive extraction.
- Recognizers for UML Glyphs.

- Domain specific enhancement of segmentation.
- Simple and intuitive facilities for user correction of segmentation and recognition errors

The implementation of a retargetable front end with domain specific enhancements has proven beneficial. It has enabled us to achieve highly accurate recognition results, yet much of our system remains general enough to integrate easily into recognition systems for other diagram notations.

When integrated with an intelligent whiteboard, our system has a number of advantages. The fact that our system accepts freehand drawings without forcing a user to learn a set of artificial actions to invoke behaviour in a system is beneficial. Users can interact with the intelligent whiteboard in the same manner they interact with any whiteboard. This aids problem solving by allowing users to think of the problem rather than how the system is used [2].

We plan to make the transition from a non-networked implementation to a distributed application which will allow group collaboration at geographically separate locations. Our system would then allow these groups the same benefits realized by whiteboard collaboration for localized groups. We also plan to extend our system to allow input and output of an XML interpretation of a UML diagram. This would allow our system to serve as a front end for any CASE tool which can import an XML interpretation of a UML diagram. It would also allow groups to edit on a smartboard UML diagrams produced by a CASE tool.

In the diagram recognition domain, we plan to extend our system in two ways. First, we intend to add the recognition of additional notations to our system. Work is planned in math and hand-drawn molecular diagrams initially. Second, we plan to perform some higher level recognition of the relationships between recognized glyphs. Addressing issues involving the recognition of the logical relationship between symbols and the meaning conveyed by symbol groupings would enhance the output of our system by providing recognition of the meaning conveyed by the diagram as a whole, rather than stopping after symbol identification.

Refinements are also planned to the correction facilities of the system. Glyph editing, in particular, could be improved. For example, class boxes often need to be resized to allow the addition of methods or attributes. We intend to implement more editing features, such as glyph resizing and moving, which users may wish to have in the UML domain.

Human Computer Interface design, software engineering, and diagram recognition present several difficult challenges. Our work has addressed many of these challenges by providing a robust, usable recognizer for the on-line drawing of UML diagrams. Many of the issues addressed by our system have implications for the recognition of a large subset of diagram notations and are not restricted solely to the domain of UML recognition.

Acknowledgements

The authors wish to thank Dr. Dorothea Blostein for her feedback and support. We would like to thank Richard Zanibbi for enlightening suggestions and many interesting conversations, and Hanaa Barakat for line drawing routines from a scribble applet which were easy to extend to incorporate stroke collection. Special thanks also to Smart Technologies for their donation of a SmartBoard in support of this research. This work was funded by the Natural Science and Engineering Research Council of Canada (NSERC) and the Center of Excellence for Communication and Information Technology of Ontario (CITO).

About the Authors

Edward Lank is a Ph.D. student and Jeb Thorley and Sean Chen are undergraduate students in the Diagram Recognition Lab, Department of Computing and Information Science, at Queen's University, Canada. Their current research interests include diagram recognition, human computer interaction, and software retargetability.

References

[1] J. Arvo. Caltech interface tools (cit). www.cs.caltech.edu/~ arvo/software.html.

[2] J. Arvo. Computer Aided Serendipity: The Role of Autonomous Assistants in Problem Solving. In *Proceedings f Graphics Interface '99*, pages 183–192, Kingston, Ontario, Canada, 1999.

[3] J. Arvo and K. Novins. Fluid sketches: Continuous recognition and morphing of simple hand-drawn shapes. Submitted to 13th Annual ACM Symposium on User Interface Software and Technology.

[4] J. Arvo and K. Novins. Smart Text: A Synthesis of Recongnition and Morphing. In *AAAI Spring Symposium on Smart Graphics*, pages 140–147, Stanford, California, USA, 2000.

[5] L. Chen and P. Yin. A System for On-line Recognition of Handwritten Mathematical Expressions. *Computer Processing of Chinese and Oriental Languages*, pages 19–39, June 1992.

[6] Michael Thomsen Christian Heide Damm, Klaus Marius Hansen and Michael Tyrsted. The knight project. www.daimi.au.dk/~ knight/.

[7] W. Citrin and M. D. Gross. Distributed Architectures for Pen-Based Input and Diagram Recognition. In *ACM Conference on Advanced Visual Interfaces '96*, 1996.

[8] E Do and M. D. Gross. Thinking with diagrams in architectural design. www.mrc-cbu.cam.ac.uk/projects/twd/discussion-papers/architecture.html.

[9] J. Rumbaugh G. Booch and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1998.

[10] D. Elliman G. Hutton, M. Cripps and C. Higgins. A Strategy for On-line Interpretation of Sketched Engineering Drawings. In *Fourth International Conference on Document Analysis and Recognition*, pages 771–775, Ulm, Germany, 1997.

[11] G. Goldman and S Zdepski. Grids in Design and CAD. In *Proceedings ACADIA 91 - Reality and Virtual Reality*, pages 33–43, Los Angles, California, USA, 1991.

[12] G. Goldman and S Zdepski. Grids in Design and CAD. In *Proceedings ACADIA '91 - Reality and Virtual Reality*, pages 33–43, Los Angles, California, USA, 1991.

[13] M. D. Gross. Why can't cad be more like lego? *Automation in Construction Journal*, 1996.

[14] J. A. Jorge and M. J. Fonseca. A Simple Approach to Recognise Geometric Shapes Interactively. In *3rd IAPR International Workshop on Graphics Recognition, GREC'99*, Jaipur, India, 1999.

[15] H. Kojima and T. Toida. On-line Handdrawn Line-figure Recognition and its Application. In *9th Intl. Conf. on Pattern Recognition*, pages 1138–1142, Rome, Italy, 1988.

[16] A. Kosmala and G. Rigoll. Recognition of On-Line Handwritten Formulas. In *6th International Workshop on Frontiers in Handwriting Recognition*, pages 219–228, Taejon, Korea, 1998.

[17] Edward Lank. Describing diagram recognition systems. available at www.cs.queensu.ca/~ lank/dr.process.ps.gz.

[18] R. Pooley and P. Stevens. *Using UML - Software Engineering with Objects and Components*. Addison-Wesley, 1998.

[19] K. Novins S. Smithies and J. Arvo. A Handwriting-Based Equation Editor. In *Proceedings f Graphics Interface '99*, pages 84–91, Kingston, Ontario, Canada, 1999.

[20] M. V. Stückelberg and D. Doermann. On Musical Score Recognition using Probabilistic Reasoning. In *5th International Conference on Document Analysis and Recognition*, pages 115–118, Bangalore, India, 1999.

[21] E. Valveny and E. Martí. Application of Deformable Template Matching to Symbol Recognition in Hand-written Architectural Drawings. In *5th International Conference on Document Analysis and Recognition*, pages 483–486, Bangalore, India, 1999.

[22] J. Coronado Y. Dimitriadis and C. de la Maza. A New Interactive Mathematical Editor, Using On-line Handwritten Symbol Recognition, and Error Detection-Correction with an Attribute Grammar. In *First International Conference on Document Analysis and Recognition*, pages 242–250, Saint Malo, France, 1991.

[23] R. Zannibbi. Recognition of mathematics notation via computer using baseline structure. Master's thesis, Queen's University, 2000.