Available online at www.sciencedirect.com
SCIENCE DIRECT
ELSEVIER
Computers & Graphics 29 (2005) 518–532
COMPUTERS & GRAPHICS
www.elsevier.com/locate/cag

LADDER, a sketching language for user interface developers

Tracy Hammond[a,*], Randall Davis[b]

[a] *MIT Computer Science and Artificial Intelligence Laboratory, 32 Vassar St., 32-239, Cambridge, MA 02141, USA*
[b] *MIT Computer Science and Artificial Intelligence Laboratory, 32 Vassar St., 32-237, Cambridge, MA 02141, USA*

Abstract

Sketch recognition systems are currently being developed for many domains, but can be time consuming to build if they are to handle the intricacies of each domain. In order to aid sketch-based user interface developers, we have developed tools to simplify the development of a new sketch recognition interface. We created LADDER, a language to describe how sketched diagrams in a domain are drawn, displayed, and edited. We then automatically transform LADDER structural descriptions into domain specific shape recognizers, editing recognizers, and shape exhibitors for use in conjunction with a domain independent sketch recognition system, creating a sketch recognition system for that domain. We have tested our framework by writing several domain descriptions and automatically generating a domain specific sketch recognition system from each description.
© 2005 Elsevier Ltd. All rights reserved.

Keywords: Object recognition; Object modeling; Shape; Hierarchical scene analysis; Knowledge representation; Representation languages; Representations; Vision and scene understanding; User-centered design

1. Introduction

As pen-based input devices have become more common, sketch recognition systems are being developed for many hand-drawn diagrammatic domains such as mechanical engineering [1–3], UML class diagrams [4–7], webpage design [8], GUI design [9,10], virtual reality [11], stick figures [12], course of action diagrams [13], and many others. These sketch interfaces (1) allow for more natural interaction than a traditional mouse and palette tool [14] by allowing users to hand sketch the diagram, (2) can automatically connect to a CAD system preventing the designer from having to enter the same information twice, (3) can offer real-time design advice from CAD systems, (4) allow more powerful editing since the shape is recognized as a whole, (5) provide diagram beautification to remove mess and clutter, and (6) use display as a trigger to inform the sketcher that the shapes have been correctly recognized. However, sketch recognition systems can be quite time consuming to build if they are to handle the intricacies of each domain. Also we would prefer that the builder of a sketch recognition system be an expert in the domain rather than an expert in sketch recognition at a signal level. Rather than build each recognition system separately, our group has been working on a multi-domain recognition system that can be customized for each domain.

Using our framework, in order to build a sketch recognition system for a new domain, a developer need only write a domain description which describes what the domain shapes look like, and how they should be displayed and edited after they are recognized. Thus, the writer of the domain description does not need to know how to program a system to perform sketch recognition. This domain description is then automatically translated

*Corresponding author.
E-mail addresses: hammond@csail.mit.edu (T. Hammond), davis@csail.mit.edu (R. Davis).

0097-8493/$ - see front matter © 2005 Elsevier Ltd. All rights reserved.
doi:10.1016/j.cag.2005.05.005

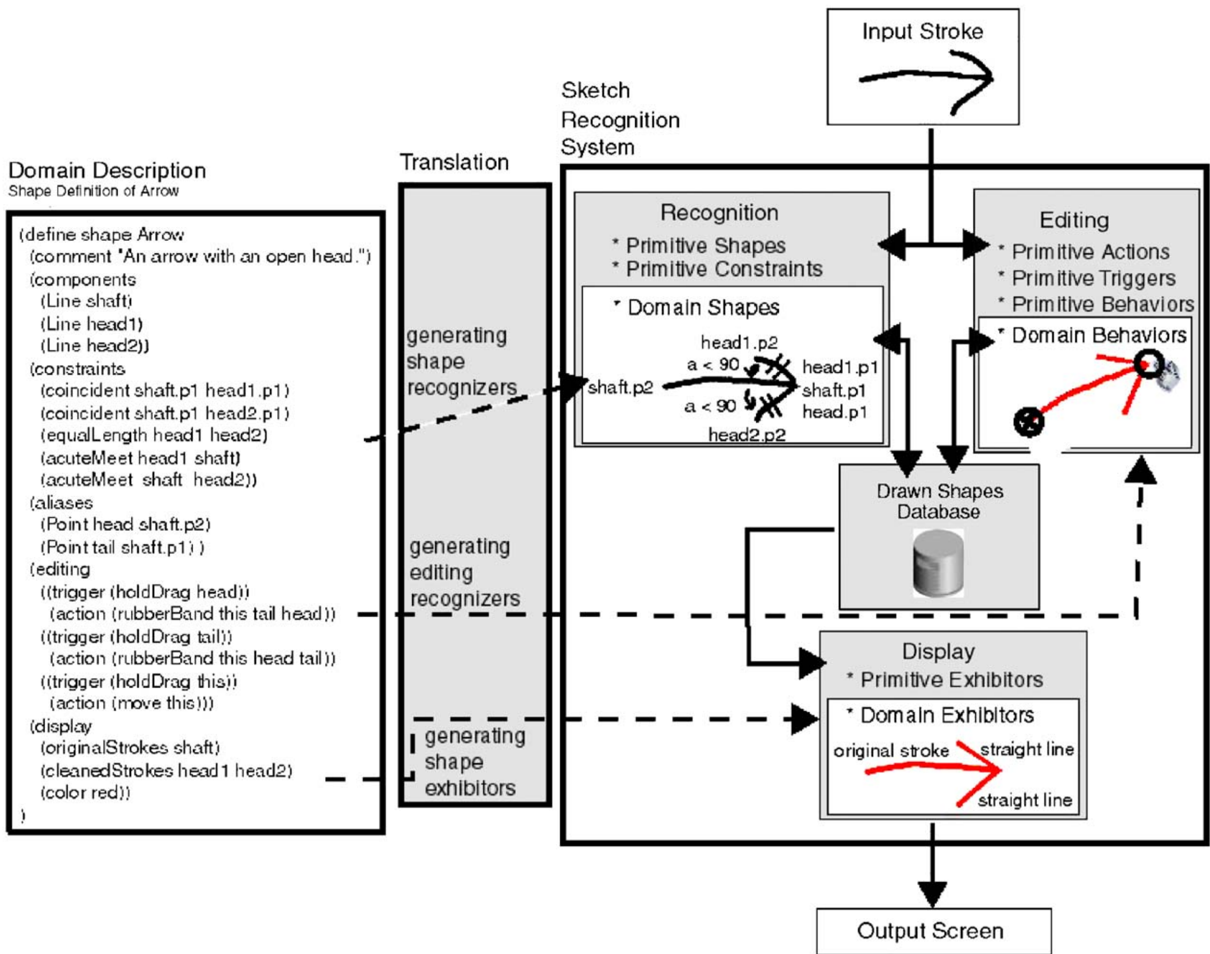

Fig. 1. System framework.

into shape recognizers, editing recognizers, and shape exhibitors for use with the customizable base domain independent recognition system creating a domain specific sketch interface that recognizes the shapes in the domain, displaying them and allowing them to be edited as specified in the description. The inspiration for such a framework stems from work in speech recognition [15,16], which has used this approach with some success.

This paper describes LADDER, the first sketch description language that can be used to describe how shapes and shape groups are drawn, edited, and displayed, and a first implemented prototype system that proves that such a framework is possible: that we can automatically generate a sketch interface for a domain from only a domain description. This work also shows that LADDER [17] is an acceptable language for describing sketch interfaces and enables us to automatically generate a sketch interface from only a LADDER domain description. To accomplish our goal, we have built (1) LADDER, a symbolic language to describe how shapes are drawn, displayed, and edited in a domain, (2) a base customizable multi-domain recognition system, and (3) a code generator [18] that parses a LADDER domain description and generates Java and Jess code to be used by the base recognition system so that it can recognize, display, and edit domain shapes. Fig. 1 shows how all three parts of the system fit together.

2. LADDER

LADDER allows interface designers to describe how shapes in a domain are drawn, displayed, and edited. LADDER descriptions primarily concern shape, but may include other information helpful to the recognition process, such as stroke order or stroke direction. The

specification of editing behavior allows the system to determine when a pen gesture is intended to indicate editing rather than a stroke. Display information indicates what to display after strokes are recognized.

The language consists of predefined shapes, constraints, editing behaviors, and display methods, as well as a syntax for specifying a domain description. The difficulty in creating such a language is ensuring that domain descriptions are easy to specify, and that the descriptions provide enough detail for accurate sketch recognition. To simplify the task of creating a domain description, shapes can be built hierarchically, reusing low-level shapes. Shapes can extend abstract shapes, which describe shared shape properties, preventing the application designer from having to redefine these properties several times. The language has proven powerful enough to describe shapes from several domains. The language enables more accurate sketch recognition by supporting both top-down and bottom-up recognition. Descriptions of how shapes may combine can aid in top-down recognition and can be used to describe "chain reaction" editing commands.

A shape definition is structural and includes primarily geometric information, but can include other drawing information that may be helpful to the recognition process, such as stroke order or stroke direction.[1] We can specify that the shaft of an arrow must be drawn before the two lines representing the head with `(drawOrder shaft head1 head2)`. We can use the same constraint to specify stroke direction; for instance `(drawOrder shaft.p2 shaft.p1)` requires that the tail of the arrow be drawn before the head.

LADDER allows the developer to specify both hard and soft constraints. Hard constraints must be satisfied for the shape to be recognized, but soft constraints may not be. Soft constraints can aid recognition by specifying relationships that usually occur. For instance, in the left box of Fig. 1, we could have specified `soft(drawOrder shaft head1 head2)` to specify that the shaft of the arrow is commonly drawn before the head, but the arrow should still be recognized even if this is not satisfied.

Before creating the language, we performed a user study where 30 people described shapes with their natural vocabulary and with increasing levels of syntactical constraints in order to ensure an intuitive vocabulary and syntax. We chose a hierarchical symbolic shape-based language as we found it to be more intuitive to describe shapes in this manner, making descriptions easier to create, understand, and correct. We also noticed that not only are shape-based geometrical properties more intuitive than feature-based properties such as those used by [19,20] (since shape is the salient feature used in human recognition), but since the features (and thus recognition) are not based on drawing style, sketchers are able to draw as they do naturally, with no constraints on stroke number, order, or direction.

LADDER is the first language that not only can define how shapes are to be recognized, but also can define how shapes are displayed and edited. Display and editing are important parts of a sketch interface, and are different in each domain. The display gives the sketcher feedback that an object was recognized and beautification can be used to remove clutter from the diagram. Because the objects are recognized we can define more powerful and intuitive editing gestures, consisting of a trigger and action, for each shape. For instance, a developer may define that an arrow can be dragged in rubber-band fashion from its head or tail, or she may define that a wheel can be moved as a whole by dragging any point within the wheel's bounding box. Although we do encourage standardization between different domains by including some predefined editing behaviors, it is important that we allow the developer to define her own editing behaviors for each domain. The same gesture, such as writing an X inside of a rectangle, may be intended as a pen stroke in the one domain (a check inside of a checkbox, or the letter X in a textbox), or as an editing command (deletion of the box).

2.1. Description limitations

LADDER can be used to describe a wide variety of shapes, but we are limited to the following class of shapes.

- LADDER can only describe shapes with a fixed graphical grammar. The shapes must be diagrammatic or iconic such that they are drawn using the same graphical components each time. For instance we cannot describe abstract shapes, such as people or cats, that would be drawn in an artistic drawing.
- The shapes must be composed solely of the primitive constraints contained in LADDER and must be differentiable from the other shapes in the language using only the constraints available in LADDER.
- Pragmatically, LADDER can only describe domains that have few curves or where the curve details are not important for distinguishing between different shapes. This is because curves are inherently different to describe in detail because of the difficulty in specifying a curve's control points. Future work includes investigating more intuitive ways of describing curves.
- Pragmatically, LADDER can only describe shapes that have a lot of regularity and not too much detail.

[1] This enables us to also describe sketching languages such as the Graffiti language for the Palm Pilot.

If a shape is highly irregular and complicated so that it cannot be broken down into subshapes which can be described, it will be cumbersome to define.

2.2. Shape definition

New shapes are defined in terms of previously defined shapes and constraints between them. An example of arrow definition is shown on the left-hand side of Fig. 1. The definition of a shape contains the following parts.

- A list of *components* specifies the elements from which the shape is built. Note that the arrow is built from three lines.
- Geometric *constraints* define the relationships on those components. The arrow definition requires that the HEAD1 and SHAFT meet at a single point and form an acute angle in a counter-clockwise direction from HEAD1 to SHAFT. (Angles are measured in a counter-clockwise direction.)
- A set of *aliases* is used to simplify other elements in the description. The HEAD and TAIL have been added as aliases in the arrow definition to more easily specify the editing behaviors.
- *Editing* behaviors specify the editing gestures triggers and how the object should react to these editing gestures. The arrow definition specifies three editing behaviors: dragging the head, dragging the tail, and dragging the entire arrow. Each editing behavior consists of a trigger and an action. Each of the three defined editing commands are triggered when the sketcher places and holds the pen on the head, tail, or shaft, and then begins to drag the pen. The actions for these editing commands specify that the object should follow the pen either in a rubber-band fashion for the head or tail of the arrow or by translating the entire shape.[2]
- *Display* methods indicate what to display when the object is recognized. A shape or its components may be displayed in any color in four different ways: (1) the original strokes of the shape, (2) the cleaned-up version of the shapes, where the best-fit primitives of the original strokes are displayed, (3) the ideal shape, which displays the primitive components of the shape with the constraints solved, or (4) another custom shape that specifies which shapes (line, circle, rectangle, etc.) to draw and where. The arrow definition specifies that the arrow should be displayed in the color red, that head1 and head2 should be drawn using CLEANEDSTROKES (a straight line in this case), and that the shaft should be drawn using the original strokes.

The domain description is translated into shape recognizers (from the *components* and *constraints* sections), exhibitors (from the *display* section), and editors (from the editing section) which are used in conjunction with a customizable recognition system to create a domain sketch interface.

2.2.1. Hierarchical shape definitions

To simplify shape definitions, shapes can be defined hierarchically. For example, the TRIANGLEARROW in Fig. 2 is composed of an ARROW and a LINE.

2.2.2. Abstract shape definitions

In the domain of UML class diagrams, there are four different types of arrows: the regular arrow, an arrow with a triangle head, an arrow with a diamond head, and an arrow with a dashed shaft. All of these arrows have the same editing behaviors. Rather than repeat the editing behaviors four times, we instead create an ABSTRACTARROW (shown in Fig. 3 which specifies the repeated editing behaviors). The *is-a* section, used in Fig. 2, specifies any class of abstract shapes that the shape may be a part of. This is similar to the extends property in Java. All shapes extend the abstract shape SHAPE. Abstract shapes have no concrete shape associated with them; they represent a class of shapes that have similar attributes or editing behaviors. An abstract shape is defined similarly to a regular shape, except it has a *required* section instead of a *components* section. Each shape that extends the abstract shape must define each variable listed in the *required* section, in its *components* or *aliases* section.

2.2.3. Shape groups

A shape group is a collection of domain shapes that are commonly found together in the domain. Defining shape groups provides two significant benefits. Shape groups can be used by the recognition system to provide top-down recognition, and "chain reaction" editing behaviors can be applied to shape groups, allowing the movement of one shape to cause the movement of another. Below we have an example describing a shape group consisting of a FORCE and a BODY (a mechanical engineering term describing a physical mass). In the

```
(define shape TriangleArrow
  (description   "An arrow with a triangle-shaped head")
  (is-a AbstractArrow)
  (components
    (Arrow a)
    (Line head3))...)
```

Fig. 2. Description for an arrow with a triangle-shaped head.

[2] Rubber-banding allows sketchers to simultaneously rotate and scale an object, assuming a fixed rotation point is defined. This action has proved useful for editing arrows and other linking shapes.

```
(define abstract-shape AbstractArrow
  (required
    (Point head)
    (Point tail)
    (Line shaft))
  (editing
   ((trigger (holdDrag shaft))
     (action  (translate this))
   ((trigger (holdDrag head))
     (action (rubberBand this head tail))
   ((trigger (holdDrag tail))
     (action (rubberBand this tail head))))
```

Fig. 3. Description for the abstract class AbstractArrow.

```
(define shape Force
  (description "An arrow is a force only if the arrow head is
 pushing an object.")
  (component (Arrow a))
  (aliases (Point head a.head) (Point tail a.tail)))

(define shape Body
  (description "Any polygon")
  (component(Polygon p)))

(define shape-group ForcePushObject
  (components
    (Force f)
    (Body b))
  (constraints
    (meet f.head b)
    (drawOrder b f) ))
```

Fig. 4. Definition of a shape group for the force/body relationship in mechanical engineering.

mechanical domain, forces *push* bodies. Forces are represented by arrows and objects are represented by polygons. If a force is said to be pushing an object, then an arrow is pointing to the polygon. The shape group ForcePushObject defined in Fig. 4 states that the head of the arrow touches the body. It also specifies that the body must be drawn before the force. If a single shape in a sketch can be part of many instances of a shape group, then we place the key word *shared* before the component shape of the shape group (e.g. if a body could have several forces we would place the word *shared* in front of the `(Body b)`) to show `shared(Body b)`. We can also define abstract shape groups.

2.3. Language contents

The power of the language is derived in part from carefully chosen predefined building blocks. The language consists of predefined shapes, constraints, editing behaviors, and display methods.

2.3.1. Predefined shapes

The language includes the primitive shapes Shape, Point, Path, Line, BezierCurve, Curve, Arc, Ellipse, and Spiral. The language also includes a library of predefined shapes built from these primitives including Rectangle, Diamond, etc. The language uses an inheritance hierarchy; Shape is an abstract shape which all other shapes extend. Shape provides a number of components and properties for all shapes, including *boundingbox*, *centerpoint*, *width*, and *height*. Each predefined shape may have additional components and properties; a Line, for example, also has *p1*, *p2* (the endpoints), *midpoint*, *length*, *angle*, and *slope*. Components and properties for a shape can be used hierarchically in shape descriptions. When defining a new shape the components and properties are those defined by Shape, and those defined by the *components* and *aliases* section.

2.3.2. Predefined constraints

A number of predefined constraints are included in the language, including *perpendicular*, *parallel*, *collinear*, *sameSide, oppositeSide*, *coincident*, *connected*, *meet*, *intersect*, *tangent, contains*, *concentric*, *larger*, *near*, *drawOrder*, *equalLength, equal*, *lessThan*, *lessThanEqual*, *angle*, *angleDir*, *acute*, *obtuse, acuteMeet,* and *obtuseMeet*. If a sketch grammar consists of only the constraints above, the shape is rotationally invariant.

There are also predefined constraints that are valid only in a particular orientation, including *horizontal*, *vertical, posSlope*, *negSlope*, *leftOf*, *rightOf*, *above*, *below*, *sameXPos, sameYPos*, *aboveLeft*, *aboveRight*, *belowLeft*, *belowRight*, *centeredBelow*, *centeredAbove*, *centeredLeft*, *centeredRight*, and *angleL*, where (*angleL line1 degrees*) specifies that the angle between a horizontal line pointing right and *line1* is *degrees*.

We have found that it is easier for many developers to describe shapes in an orientation dependent fashion. However, since the developer may still want a shape to be recognizable in any orientation, the language allows a developer to describe shapes in an orientation dependent fashion and then specify that the shape is rotatable. For this purpose, the language contains an additional constraint: *isRotatable*, which implies the shape can be found in any orientation. If *isRotatable* is specified along with an orientation dependent constraint, there must be an *angleL*, *horizontal*, or *vertical* constraint specified, which serves to define the orientation and set a relative coordinate system. For example, the two *angleMeet* constraints could have been replaced with:

```
(isRotatable)
(horizontal shaft)
(negSlope head1)
(posSlope head2)
(leftOf shaft.p1 shaft.p2)
(leftOf head1.p2 shaft.p2)
(leftOf head2.p2 shaft.p2),
```

in which case the *shaft* is the reference line.

2.3.3. Predefined editing behaviors, actions, and triggers

Describing editing gestures permits the recognition system to discriminate between sketching (pen gestures intended to leave a trail of ink) and editing gestures (pen gestures intended to change existing ink), and permits us to describe the desired behavior in response to a gesture.

In order to encourage interface consistency, the language includes a number of predefined editing behaviors described using the actions and triggers above. One such example is *dragInside*, and defines that if you hold the pen for a brief moment inside the bounding box of a shape and then start to drag the pen (specified by the trigger `(holdDrag Shape)`, the entire shape automatically translates along with the motion of the pen.

When defining a new editing behavior particular to a domain, there are two things to specify: the trigger—what signals an editing command—and the action—what should happen when the trigger occurs. The language has a number of predefined triggers and actions to aid in describing editing behaviors.

The arrow definition in Fig. 1 defines three editing behaviors. The first editing behavior says that if you click and hold the pen over the SHAFT of the ARROW, when you drag the pen, the entire ARROW will translate along with the movement of the arrow. The second editing behavior states that if you click and hold the pen over the HEAD of the arrow, the HEAD of the arrow will follow the motion of the pen, but the TAIL of the arrow will remain fixed and the entire ARROW will stretch like a rubber-band (translating, scaling, and rotating) to satisfy these two constraints and keep the ARROW as one whole shape. All of the editing behaviors also change the pen's cursor as displayed to the sketcher, and display moving handles to the sketcher to let the sketcher know that she is performing an editing command.

The possible editing actions include *wait*, *select*, *deselect*, *color*, *delete*, *translate*, *rotate*, *scale*, *resize*, *rubberBand*, *showHandle*, and *setCursor*. To give an example: `(rubberBand shape-or-selection fixed-point move-point [new-point])` translates, scales, and rotates the *shape-or-selection* so that the *fixed-point* remains in the same spot, but the *move-point* translates to the *new-point*. If *new-point* is not specified, *move-point* translates according to the movement of the pen.

The possible triggers include *click*, *doubleClick*, *hold*, *holdDrag*, *draw*, *drawOver*, *scribbleOver*, and *encircle*. Possible triggers also include any action listed above, to allow for "chain reaction" editing.

Shape groups allow designers to define "chain reaction" editing behaviors. For instance, the designer may want to specify that when we move a rectangle, if there is an arrowhead inside of this rectangle, the arrow should move with the rectangle.

2.3.4. Predefined display methods

An important part of a sketching interface is controlling what the sketcher sees after shapes are recognized, both of which can be used to clean up the sketch as desired for the domain and provide feedback to the sketcher that a shape has been recognized. The designer can specify that the original strokes should remain, or instead that a cleaned version of the strokes should be displayed. In the cleaned version, the original strokes are fit to straight lines, clean curves, clean arcs, or a combination.

Another option is to display the ideal version of the strokes where the constraints listed in the definition are solved. In this case, lines that are supposed to connect at their end points actually connect and lines that are supposed to be parallel are actually shown as parallel. In the ideal version of the strokes, all of the low-level signal noise from sketching is removed.

It may be that we do not want to show any version of the strokes at all, but some other picture. In this case, we can either place an image at a specified location, size, and rotation (using the method IMAGE), or we can create a picture built out of predefined shapes, such as circles, lines, and rectangles.

The predefined display methods include *originalStrokes*, *cleanedStrokes*, *idealStrokes*, *circle*, *line*, *point*, *rectangle*, *text*, *color*, and *image*. Each method includes color as an optional argument.

2.4. Vectors

The arrow defined in Fig. 1 contains a fixed number of components (3). However, many shapes that we would like to define, such as a POLYGON, POLYLINE, or DASHEDLINE, contain a variable number of components. A POLYLINE may contain a variable number of line segments. A variable number of components is specified by the key word *vector* and must specify the minimum and maximum number of components. If the maximum number can be infinite, the variable *n* is listed. For instance, the POLYLINE must contain at least two lines, and each line must be connected with the previous. The definition of a POLYGON easily follows from the definition of the POLYLINE (both are defined in Fig. 5).

Likewise, a DASHEDARROW is made from an ARROW, and a DASHEDLINE (both defined in Fig. 6), which in turn contains at least two line segments. When given a third argument specifying a length, the constraint *near* states that two points are near to each other relative to a given length.

```
(define shape PolyLine
  (components (vector Line vl[2,n]))
  (constraints (coincident vl[i].p2 vl[i+1].p1))
  (aliases (Point head vl[0].p1)(Point tail vl[n].p2)))

 (define shape Polygon
  (components(PolyLine poly))
  (constraints(coincident poly.head poly.tail)))
```

Fig. 5. Shape description of a polygon.

```
(define shape DashedLine
  (components (vector Line vl[2,n]))
  (constraints (collinear vl[i].p1 vl[i].p2 vl[i+1].p1)
    (not(intersect vl[i] vl[i+1])
    (near vl[i].p2 vl[i+1].p1 vl[i].length))
  (aliases (Point head vl[0].p1)(Point tail vl[n].p2)))

(define shape DashedOpenArrow
  (components (OpenArrow oa)(DashedLine dl))
  (constraints (near oa.tail dl.head oa.shaft))
  (aliases
    (Point head oa.head)(Point tail dl.tail)))
```

Fig. 6. Description of a dashed line and a dashed open arrow.

3. Multi-domain recognition system

3.1. Recognition of primitive shapes

The base customizable recognition system contains domain independent modules that can recognize, exhibit, and edit all of the primitive shapes in LADDER. These modules are noted by the shaded boxes without their inner white domain modules on the right side of Fig. 1.

When a stroke is drawn (and has not been identified as an editing gesture, described below), low-level recognition is performed on the stroke. The domain independent modules determine if the stroke can be classified as an ELLIPSE, LINE, CURVE, ARC, POINT, POLYLINE or some combination using techniques by [21]. In many cases the stroke is ambiguous and has more than one interpretation. When this happens both interpretations are produced and sent off to the higher level recognizer.

We want to ensure that the domain shape recognition system only chooses one interpretation of a single stroke. In order to ensure that only one interpretation is chosen, each shape has an ID, and each shape keeps a list of its subshapes, including its strokes. At any particular time, each subshape is allowed to belong to only one final recognized domain shape. (A final shape is a chosen interpretation as opposed to the myriad of possible interpretations that are created and kept until one is finally chosen.) To give an example, the STROKE in Fig. 7A has two interpretations: a LINE and an ARC. The figure specifies each shape's ID followed by the IDs of all of the subshapes. Note that the LINE and the ARC share the same STROKE subpart. If a shape has a

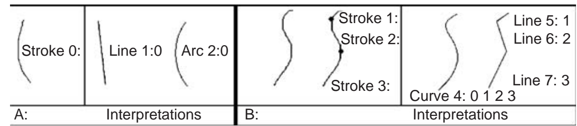

Fig. 7. (A) An ambiguous stroke that can be a LINE or an ARC. (B) An ambiguous stroke that can be a CURVE or a POLYLINE.

POLYLINE interpretation, or some other combination interpretation, the STROKE must be divided into segments. The original full STROKE then has the substrokes added as subparts. These substrokes are then included in any interpretation that uses the full stroke. For example in Fig. 7B the CURVE contains all of the substrokes as subshapes. Since the set of final shapes cannot share any subshapes, this prevents the CURVE and the POLYLINE from both being chosen in a final interpretation.

A limitation with this bottom-up recognition method is that if the primitive shape recognizer does not provide the correct interpretation of a stroke, the domain shape recognizer will never be able to correctly recognize a higher level shape using this stroke. In the future, it may be advantageous to add a top-down recognition process that re-examines lower level shapes if an item is missing from the template, such as that done in [22].

3.2. Recognition of domain shapes

Recognition of domain shapes occurs as a series of bottom-up opportunistic data driven triggers where the recognized shapes in the drawing represent the facts about the world. Domain shape recognition is performed by the Jess rule-based system [23]. When a new shape primitive shape is recognized, it is added as a fact into the Jess rule-based system. We created several Jess rules to perform higher level clean-up on the shapes, such as merging lines together.

Each domain shape recognizer is actually a Jess rule defined to recognize the shape (the translation process creating the Jess rule is explained below). The Jess rule-based system searches for all possible combinations of shapes that can satisfy the rule. When choosing between competing shapes we use Okhams razor, choosing the shape that accounts for more of the underlying data. If two choices are equivalent, we choose the shape created first, assuming that a sketcher prefers his shapes to remain constant on the screen. As the number of lower level shapes increases, the rule-based system slows down exponentially, since as each new stroke is drawn the system tries to join it with every other existing stroke to attempt to form higher level shapes. To improve efficiency and to allow the application to continue to react in real time, we have implemented a greedy algorithm that removes subshapes from the rule-based

system once a final higher level shape is chosen. Our greedy approach is limited in that in cases of higher level ambiguity, the system may select the wrong higher level shape.

For the same reasons as above, the higher level recognizer also slows down if there are a large number of unrecognized strokes on the screen since the system continues to try to match each of the unrecognized strokes with every new stroke. A quick fix to this is to simply prune unrecognized strokes from the recognition tree after some time. However, this is not ideal as the user may decide to finish a shape after some time, and we would like to be able to recognize this shape. Future work will investigate ways of solving this problem in the future.

When the shapes are more constrained, the recognizer performs faster since the system will hold fewer partial templates to examine later. For instance, if a line direction is constrained, the recognizer only tries to fit the line in one direction, and if a line is horizontal, the recognizer only needs to try the subset of the lines that are horizontal.

We should note that even during these extreme cases when the higher level domain shape recognition begins to slow down and stops reacting in real time, the rest of the system continues to run in real time since the drawing panel, primitive recognition and, domain shape recognition all run in three separate threads, with the drawing panel being given priority to ensure that the pen markings always appear in real time.

3.3. Editing recognition

A stroke may be intended as an editing gesture rather than a drawing gesture. If an editing gesture such as click-and-hold or double-click occurs, the system checks to see (1) if an editing gesture for that trigger is defined for any shape, and (2) if the mouse is over the shape the gesture is defined for. If so then the drawing gesture is short-circuited and the editing gesture takes over (for instance, the shape may then be dragged). Other triggers, such as shape-over, may require that the drawing gesture be completed and recognized before the action, such as deleting the shape underneath, occurs. For example, consider the editing gesture `(trigger (drawOver Cross Shape)) (action (delete Shape) (delete Cross))`; in this example when a Cross is drawn over a Shape, the Shape is deleted (as is the Cross).

3.4. Constraint solver

As mentioned earlier, each shape can be displayed by its original strokes, best-fit primitives, the best-fit primitives with all of the constraints solved (which we will refer to as the shape's ideal strokes), or through Java Swing objects.

To display a shape's ideal strokes, the system uses a shape constraint solver which takes in a shape description and initial locations for all of the subshapes and outputs the shape with all of the constraints satisfied while moving the points as little as possible. Because the positions of the shape's components, its properties, and its constraints are all interrelated, we need to generate and solve algebraic equations demonstrating these relations. We have constructed this shape constraint solver using optimization functions from Mathematica.

To generate a shape we first convert each shape's components, properties (such as, width, height, area), and constraints into a set of algebraic equations. These equations are then solved to find a mathematical solution representing a shape that satisfies the description.

We translate each shape, its components, and its properties using the schema listed below. This produces a set of equations describing the object. For example, one equation produced is `arrow.area == arrow.width * arrow.height`.

- Minimize: To prevent the shape from shifting too much, we minimize the distance from the value in the initial hand-drawn example and the final solved value for each component.
- Require: To prevent the lines from collapsing to a point, all lines must have a length greater than 10 pixels.
- We define the bounding box of a shape (*minx*, *miny*, *maxx*, *maxy*), so that we can enforce area-related constraints such as *equalArea*, *larger*, *contains*, as follows:
  - Define shape.minx recursively:
    if (shape is line):
    Require: shape.minx < shape.p1.x
    Require: shape.minx < shape.p2.x
    else for each component:
    Require: shape.minx < shape.component.minx
  - Define shape.miny, shape.maxx, shape.maxy similarly
  - Minimize: shape.maxx
  - Minimize: shape.maxy
  - Minimize: −1 * shape.minx
  - Minimize: −1 * shape.miny
- Require: shape.width = = shape.maxx - shape.minx
- Require: shape.height = = shape.maxy - shape.miny
- Require: shape.area = = shape.width * shape.height
- Require: shape.center.x = = (shape.minx + shape.maxx)/2

- Require: shape.center.y = = (shape.miny + shape.maxy)/2

Next we translate each constraint into a set of equations on the variables defined above. For example:

horizontal line1 becomes $line1.p1.y == line1.p2.y$
contains shape1 shape2 becomes
$(shape1.minx < shape2.minx)$ &&
$(shape1.miny < shape2.miny)$ &&
$(shape1.maxx > shape2.maxx)$ &&
$(shape1.maxy > shape2.maxy)$
equalLength line1 line2 becomes
$(line1.p1.x - line1.p2.x)^2 + (line1.p1.y - line1.p2.y)^2 ==$
$(line2.p1.x - line2.p2.x)^2$
$+(line2.p1.y - line2.p2.y)^2$
not sameX shape1 shape2 becomes
$(shape1.center.x + 20 < shape2.center.x)$
$||(shape1.center.x > shape.center.x + 20)$
(Because perceptually, small differences in 'x' values may not be detected, when constraining the shape to have different 'x' values we require the difference to be at least 20 pixels.)

Finally, we use the NMinimize function in Mathematica, which finds constrained global optima, to find a solution that satisfies all of the equations above. We now have new positions for each of the shape's components which satisfy the constraints in the description. The system will then display the beautified shape.

4. Code generation

Domain shape recognizers, exhibitors, and editors are automatically generated during the translation process shown in the middle of Fig. 1. A shape definition is composed of three parts: how to recognize the shape, how to display the shape once it is recognized, and how to edit the shape once it is recognized. The translation process parses the description and generates code specifying how to recognize shapes and editing triggers as well as how to display the shapes once they are recognized and what action to perform once an editing trigger occurs.

The components and constraints sections of a shape description are automatically translated into a Jess rule defining how to recognize that shape. The Jess rule created for the arrow definition listed in Fig. 1 is shown in Fig. 9. The Jess rule created first searches for the appropriate combination of subshapes, and then tests the constraints between them.[3] We have built our customizable base recognition system in an effort to keep the translation process as simple as possible.

If a shape consists of a variable number of components such as a polyline (as opposed to an arrow which is composed of a fixed (3) number of components), the shape description is translated into two Jess rules, one recognizing the base case (a polyline composed of two lines) and the other recognizing the recursive case (a polyline composed of a line and a polyline).

A shape exhibitor is automatically generated as a Java paint method for the shape, which calls functions in the base recognition system defined to work for any shape. A shape can be displayed by one or more of the following: its original strokes, its best-fit primitives, its best-fit primitives with the constraints solved, a collection of Java Swing shapes, or a bitmap image. To display the ideal strokes (the best-fit primitives with the constraints solved), an IDEAL-PAINT method is automatically generated that defines the constraints to be solved by the shape-based constraint solver.

A shape editor is automatically generated defining which triggers are turned on for the shape or its subshapes. If the trigger is turned on, then the corresponding actions are defined in an automatically generated method. The base recognition system includes methods to identify triggers and perform actions for each shape, and the generated method needs only to turn them on.

5. Evaluation

We have written LADDER domain descriptions for a variety of domains including UML class diagrams, mechanical engineering, finite state machines, flowcharts, and a simplified version of the course of action diagrams (Fig. 8). Using the system presented in this paper, the descriptions have been automatically translated into a sketch interface which recognizes, displays, and allows editing in real time as specified by the domain description. These descriptions include over one hundred shapes, some containing text.[4] Figs. 10–12 show the unrecognized and recognized strokes from a drawing made in an automatically generated mechanical engineering, flowchart, and finite state machine sketch interfaces.

6. Related work

This paper discusses in more detail work from [17,18]. In particular this paper gives more details on the

[3] Our current implementation does not yet support soft constraints.

[4] Text is also a primitive shape. Text can be entered using a keyboard or a handwriting recognizer GUI provided in Microsoft Tablet XP. The text appears at the last typed place.

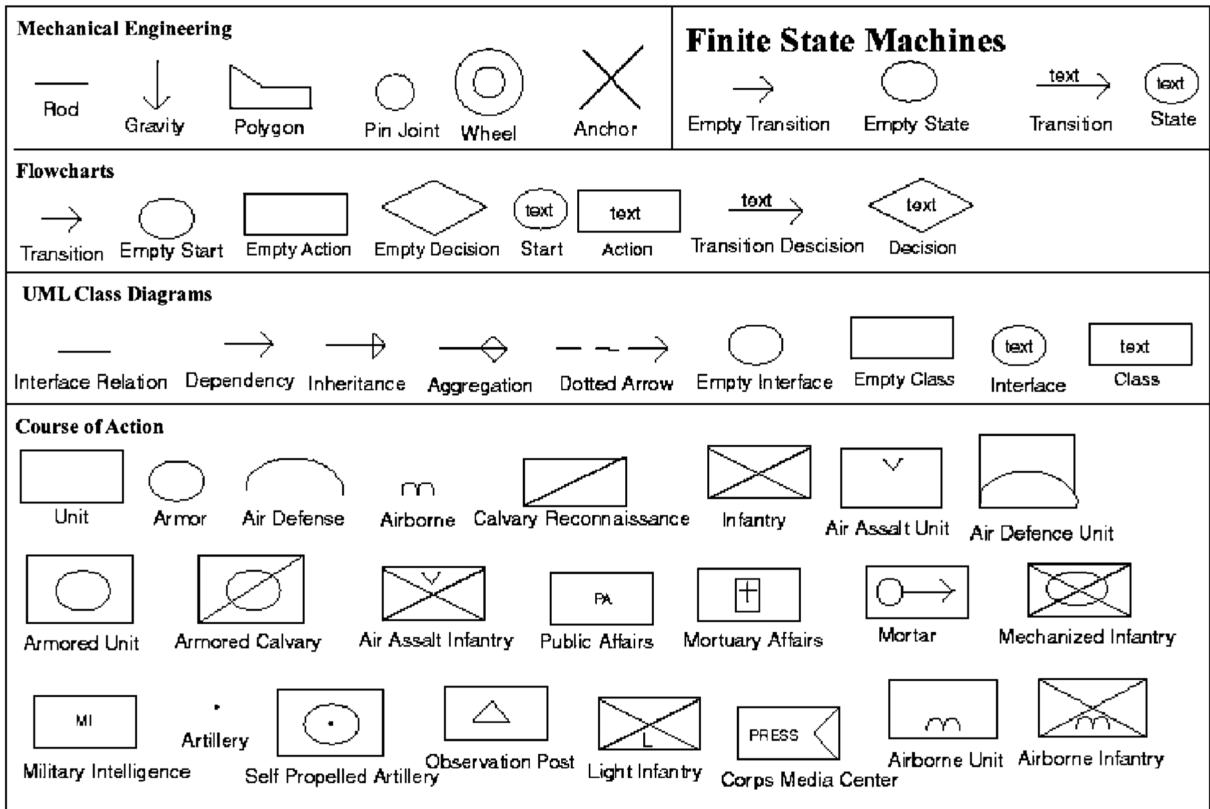

Fig. 8. Variety of shapes and domains described and auto-generated.

primitive and domain shape recognizers, the handling of ambiguous primitive shapes, and the techniques used for beautifying shapes.

6.1. Visual or sketching languages

Shape definition languages, such as shape grammars, have been around since the early 1970s [24]. Shape grammars are studied widely within the field of architecture, and many systems are continuing to be built using shape grammars [25]. Shape grammars have, however, been used largely for shape generation rather than recognition, and do not provide for non-graphical information, such as stroke order, that may be helpful in recognition. They also lack ways for specifying shape editing.

More recent shape definition languages have been created for use in diagram parsing [26]. These shape definition languages are not intended for use with an online system and do not provide ways for specifying how to display or edit a shape. Also, since they are not created with sketching in mind they do not provide ways for describing non-graphical information, such as stroke order or direction.

Within the field of sketch recognition, there have been other attempts to create languages for sketch recognition. Bimber describes a simple sketch language using a BNF-grammar [27]. The language describes three-dimensional shapes hierarchically. This language allows a programmer to specify only shape information and lacks the ability to specify other helpful domain information such as stroke order or direction and editing behavior, display, or shape interaction information.

Mahoney uses a language to model and recognize stick figures. The language currently is not hierarchical, making large objects cumbersome to describe [28]. Caetano et al. use fuzzy relational grammars to describe shape [29]. Both Mahoney and Caetano lack the ability to describe editing, display, or shape grouping information.

Shilman has developed a statistical language model for ink parsing with a similar intent of facilitating development of sketch recognizers. The language consists of seven constraints: *distance*, *delta X*, *delta Y*, *angle*, *width ratio*, *height ratio*, and *overlap*, and allows you to specify concrete values using either a range or gaussian [30]. We find it difficult to describe some shapes

using this technique as the language requires providing quantitative discrete values about a shape's probable location. We feel it is more intuitive to say `(contains shape1 shape2)`, rather than having to specify two *deltaX* and two *deltaY* constraints using discrete constraints, each of the form `deltaX (shape1.WEST`

```
(defrule ArrowCheck
  ;; get three lines
  ?f0 $<$- (Subshapes Line ?shaft \$?shaft_list)
  ?f1 $<$- (Subshapes Line ?head1 \$?head1_list)
  ?f2 $<$- (Subshapes Line ?head2 \$?head2_list)
  ;; make sure lines are unique
  (test (uniquefields \$?shaft_list \$?head1_list))
  (test (uniquefields \$?shaft_list \$?head2_list))
  (test (uniquefields \$?head1_list \$?head2_list))
  ;; get accessible components of each line
  (Line ?shaft ?shaft_p1 ?shaft_p2 ?shaft_midpoint ?shaft_length)
  (Line ?head1 ?head1_p1 ?head1_p2 ?head1_midpoint ?head1_length)
  (Line ?head2 ?head2_p1 ?head2_p2 ?head2_midpoint ?head2_length)
  ;; test constraints
  (test (coincident ?head1_p1 ?shaft_p1))
  (test (coincident ?head2_p1 ?shaft_p1))
  (test (equalLength ?head1 ?head2))
  (test (acuteMeet ?head1 ?shaft))
  (test (acuteMeet ?shaft ?head2))
  ;;deleted code: get line with endpoints swapped

=>  ;; FOUND ARROW (ACTION TO BE PERFORMED)
  ;; set aliases
  (bind ?head ?shaft_p1)
  (bind ?tail ?shaft_p2)
  ;; add arrow to sketch recognition system to be displayed properly
  (bind ?nextnum (addshape Arrow ?shaft ?head1 ?head2 ?head ?tail))
  ;; add arrow to Jess fact database
  (assert (Arrow ?nextnum  ?shaft ?head1 ?head2 ?head ?tail))
  (assert (Subshapes Arrow ?nextnum (union\$ \$?shaft_list
          \$?head1_list \$?head2_list)))
  (assert (DomainShape Arrow ?nextnum (time)))
  ;; remove Lines from Jess fact database for efficiency
  (retract ?f0) (assert (CompleteSubshapes Line ?shaft \$?shaft_list))
  (retract ?f1) (assert (CompleteSubshapes Line ?head1 \$?head1_list))
  (retract ?f2) (assert (CompleteSubshapes Line ?head2 \$?head2_list))
  ;;deleted code: retract line with endpoints swapped
)
```

Fig. 9. Automatically generated Jess rule for the arrow definition in the left box of Fig. 1.

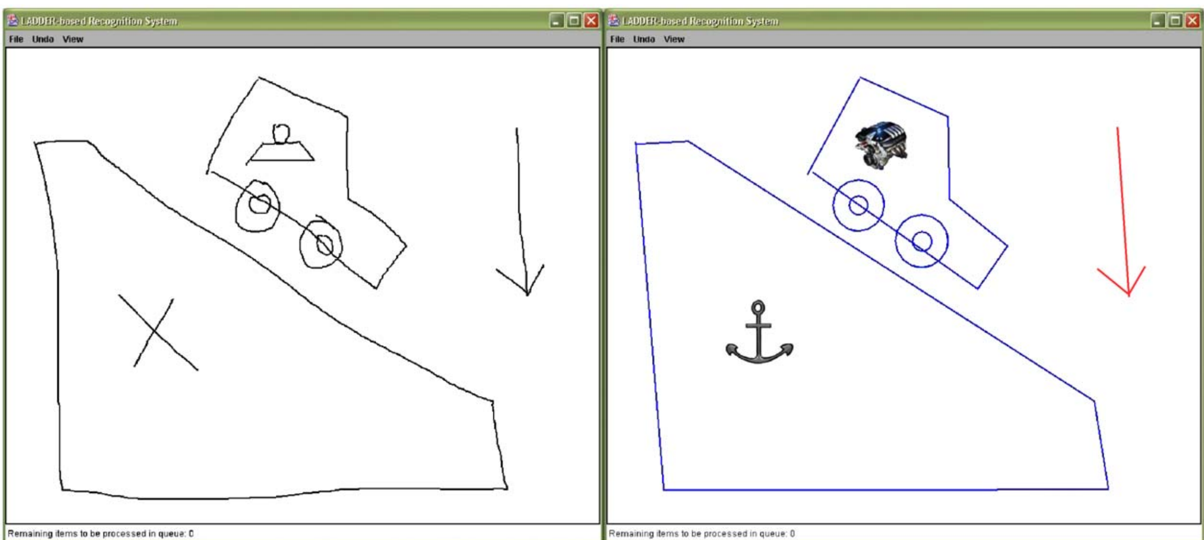

Fig. 10. Auto-generated mechanical engineering interface.

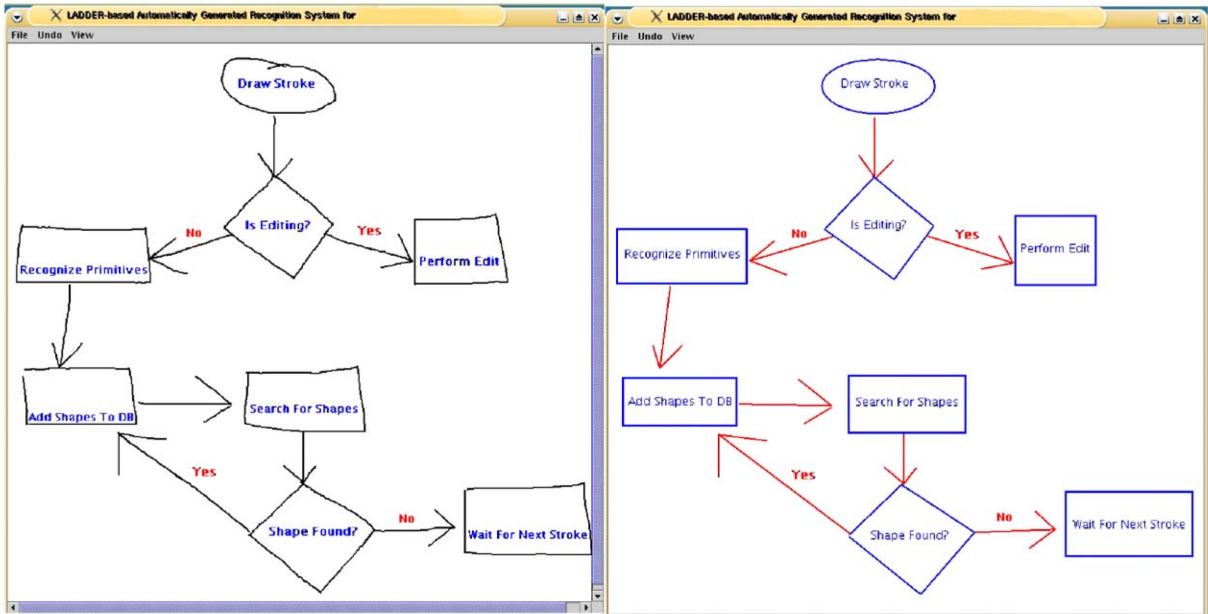

Fig. 11. Auto-generated flowchart interface.

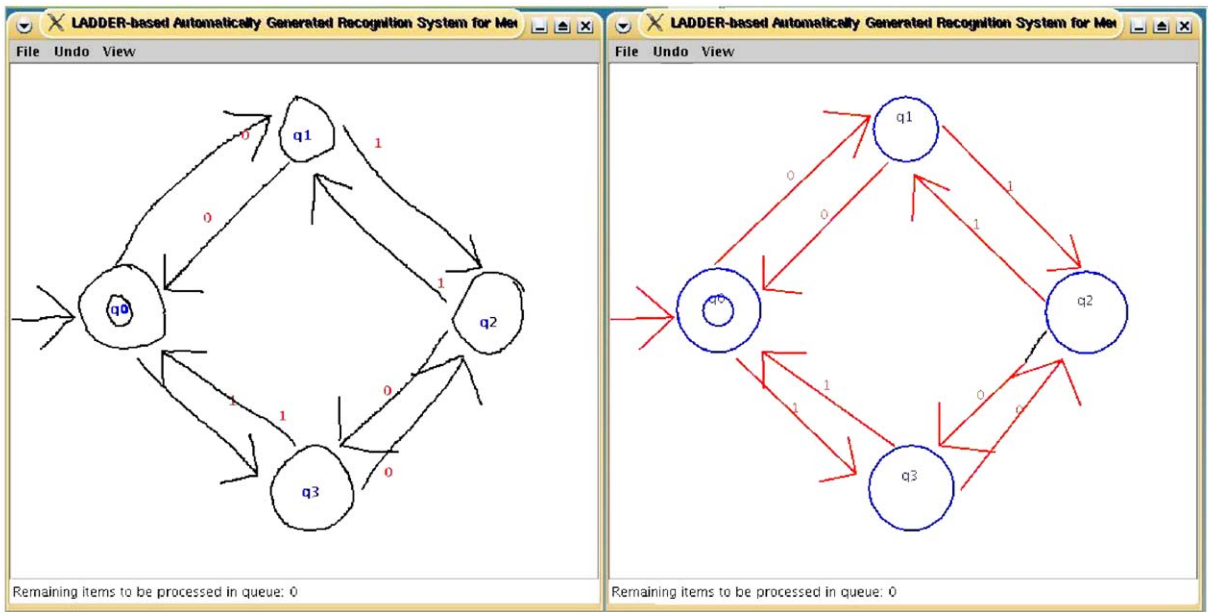

Fig. 12. Auto-generated finite state machine interface.

`< shape2.WEST).range(0, 100))` This work also lacks the ability to describe editing and display.

Our recognition system is based on template filling of a shape's structural description. These structural descriptions are often represented in relational graphs. Lee performs recognition using attribute relational graphs [31]. Their attribute language differs from ours in that ours is more topological or geometrical, whereas their language is more quantitative, requiring specific details of the shape's position. Keating also performs recognition by matching a graph representation of a shape; the main difference between their limited graphical language

and ours is that their language is statistical and specifies the probable location of each subpart, whereas our language is categorical and describes the ideal location of the shape [32]. Calhoun also uses a semantic network representing the shape in recognition, but as far as we can tell the language is limited, specifying only relative angles and the location of intersections [33].

6.2. Building recognition systems

Quill [20] is a feature-based graffiti-type domain-independent gesture recognition system that allows designers of a gesture recognition system to sketch the gestures to be recognized. The system then provides advice about how well the gestures will be recognized by the computer and how well they will be learned and remembered by people. The Quill framework differs from ours in using recognizers based on features and in focusing on the way the shape is drawn (e.g., the number of strokes, as well as stroke, speed, curvature, order, direction, etc.). In order for their strokes to be recognized, sketchers of this system must sketch a gesture in the same way as the developer who trained the system. Our focus is on removing as many sketching restrictions as possible, to provide a more natural sketching medium. We want users' sketches to be recognized no matter how many strokes they used or in what direction or order they were drawn. Thus, our framework includes a symbolic language for describing the geometry of shapes from which to base recognition.

The Electronic Cocktail Napkin project [34] allows users to define domain shapes by drawing them. A shape is described by the shapes it is built out of and the constraints between them. The Cocktail Napkin's language is able to describe only shape.

Jacob [35] has created a software model and language for describing and programming fine-grained aspects of interaction in a non-WIMP user interface, such as a virtual environment. The language is low level, making it difficult to define new interactions, and, in the domain of sketching, does not provide a significant improvement in comparison to coding the domain dependent recognition system from scratch.

6.3. Translation

The translation process is analogous to work done on compiler compilers, in particular, visual language compiler compilers by Costagliola et al. [36]. A visual language compiler compiler allows a user to specify a grammar for a visual language, and then compiles it into a recognizer which can indicate whether an arrangement of icons is syntactically valid. The main difference between Costagliola's work and ours is that (1) ours handles hand-drawn images and (2) their primitives are the iconic shapes in the domain whereas our primitives are geometric.

7. Future work

While we have attempted to make LADDER as intuitive as possible, shape definitions can be difficult to describe textually, and we are currently integrating Veselova's work to automatically generate shape descriptions from a drawn example [37]. However, even automatically generated shapes will need to be checked and modified. Thus we are in the process of building a graphical debugger which tests if shapes are over- or under-constrained by generating suspected near-miss example shapes [38].

We want to ensure that our framework and language are robust and thus we are continuing to test our system on more domains. For the same reason, we would like to test our syntax on a wide user base.

We are in the process of building an API to allow the designer to connect to a CAD system to build more sophisticated sketch systems. We would also like to allow users building systems in languages other than Java to access recognition events by registering for them.

8. Contributions

We have developed an innovative framework in which developers need to write only a LADDER domain description, and then this description is automatically transformed into a sketch recognition interface for that domain. We have implemented a prototype system and tested our framework by writing descriptions for several domains and then automatically generating a sketch interface for each of these domains. To accomplish our goal, we have created (1) LADDER, the first symbolic domain description language to describe how sketched diagrams in a domain are drawn, displayed, and edited, (2) a customizable base recognition system, which performs the domain independent parts of recognition usable for many domains, and (3) a code generator that translates a domain description into higher level domain specific recognition code to be used by the customizable base recognition system.

Acknowledgements

The authors would like to thank Eric Saund, Mike Oltmans, Jacob Eisenstein, Aaron Adler, Metin Sezgin, Christine Alvarado, Sonya Cates, Mark Finlayson, Raghavan Parthasarthy, Jan Hammond, Rob Miller, James Landay, and Tom Stahovich for their feedback. This work is supported in part by the MIT/Microsoft iCampus initiative and in part by MIT's Project Oxygen.

References

[1] Alvarado C. A natural sketching environment: bringing the computer into early stages of mechanical design. Master's thesis, MIT, 2000.

[2] Landay JA, Myers BA. Interactive sketching for the early stages of user interface design. In: Proceedings of CHI '95: Human Factors in Computing Systems, 1995. p. 43–50.

[3] Stahovich T. Sketchit: a sketch interpretation tool for conceptual mechanism design. Technical Report, MIT AI Laboratory, 1996.

[4] Hammond T, Davis R. Tahuti: a geometrical sketch recognition system for uml class diagrams, AAAI Spring Symposium on Sketch Understanding 2002; 59–68.

[5] Damm CH, Hansen KM, Thomsen M. Tool support for cooperative object-oriented design: gesture based modeling on an electronic whiteboard. In: CHI 2000, CHI; 2000. p. 518–25.

[6] Ideogramic, Ideogramic UML™, Ideogramic ApS, Denmark, http://www.ideogramic.com/products/, 2001.

[7] Lank E, Thorley JS, Chen SJ-S. An interactive system for recognizing hand drawn UML diagrams. In: Proceedings for CASCON 2000; 2000. p. 7.

[8] Lin J, Newman MW, Hong JI, Landay J. DENIM: finding a tighter fit between tools and practice for web site design. In: CHI Letters: Human Factors in Computing Systems, CHI; 2000. p. 510–7.

[9] Caetano A, Goulart N, Fonseca M, Jorge J. JavaSketchIt: issues in sketching the look of user interfaces, Sketch Understanding. Papers from the 2002 AAAI Spring Symposium.

[10] Lecolinet E. Designing guis by sketch drawing and visual programming. In: Proceedings of the International Conference on Advanced Visual Interfaces (AVI 1998), AVI, New York: ACM Press, 1998. p. 274–6. URLciteseer.nj.nec.com/lecolinet98designing.html

[11] Do EY-L. Vr sketchpad—create instant 3d worlds by sketching on a transparent window. In: de Vries B, van Leeuwen JP, Achten HH, editors. CAAD Futures 2001. 2001. p. 161–72.

[12] Mahoney JV, Fromherz MPJ. Handling ambiguity in constraint-based recognition of stick figure sketches. SPIE Document Recognition and Retrieval IX Conference, San Jose, CA.

[13] Pittman J, Smith I, Cohen P, Oviatt S, Yang T. Quickset: a multimodal interface for military simulations. Proceedings of the Sixth Conference on Computer-Generated Forces and Behavioral Representation, 1996. p. 217–24.

[14] Hse H, Shilman M, Newton AR, Landay J. Sketch-based user interfaces for collaborative object-oriented modeling. Berkley CS260 Class Project (December 1999).

[15] Zue, Glass, Conversational interfaces: advances and challenges. Proceedings of IEEE, 2000. p. 1166–80.

[16] VoiceXML Forum, http://www.voicexml.org/specs/VoiceXML-100.pdf, Voice eXtensible Markup Language, 1st ed. (07 March 2000).

[17] Hammond T, Davis R. LADDER: a language to describe drawing, display, and editing in sketch recognition. Proceedings of the 2003 International Joint Conference on Artificial Intelligence (IJCAI).

[18] Hammond T, Davis R. Automatically transforming symbolic shape descriptions for use in sketch recognition. Proceedings of the 19th National Conference on Artificial Intelligence (AAAI-04).

[19] Rubine D. Specifying gestures by example. Computer Graphics 1991;25(4):329–37.

[20] Long AC. Quill: a gesture design tool for pen-based user interfaces, Eecs department, computer science division, UC Berkeley, Berkeley, CA, December 2001.

[21] Sezgin TM. Feature point detection and curve approximation for early processing in sketch recognition. Master's thesis, Massachusetts Institute of Technology, June 2001.

[22] Alvarado C, Davis R. Sketchread: a multi-domain sketch recognition engine. In: Proceedings of UIST '04, 2004. p. 23–32.

[23] Friedman-Hill E, Jess, the java expert system shell. http://herzberg.ca.sandia.gov/jess, 2001.

[24] Stiny G, Gips J. Shape grammars and the generative specification of painting and sculpture. In: Freiman CV, editor. Information processing, vol. 71. Amsterdam: North-Holland, 1972. p. 1460–5.

[25] Gips J. Computer implementation of shape grammars. NSF/MIT Workshop on Shape Computation.

[26] Futrelle RP, Nikolakis N. Efficient analysis of complex diagrams using constraint-based parsing. In: ICDAR-95 (International Conference on Document Analysis and Recognition), Montreal, Canada; 1995. p. 782–90.

[27] Bimber O, Encarnacao LM, Stork A. A multi-layered architecture for sketch-based interaction within virtual environments. Computer and Graphics (Special Issue on Calligraphic Interfaces: Towards a New Generation of Interactive Systems) 2000; 24(6): 851–67.

[28] Mahoney JV, Fromherz MPJ. Three main concerns in sketch recognition and an approach to addressing them. In: Sketch Understanding, Papers from the 2002 AAAI Spring Symposium. Stanford, CA: AAAI Press, 2002. p. 105–12.

[29] Caetano A, Goulart N, Fonseca M, Jorge J. Sketching user interfaces with visual patterns. Proceedings of the First Ibero-American Symposium in Computer Graphics (SIACG02), 2002. p. 271–9.

[30] Shilman M, Pasula H, Russell S, Newton R. Statistical visual language models for ink parsing. In: Sketch Understanding, Papers from the 2002 AAAI Spring Symposium. Stanford, CA: AAAI Press, 2002. p. 126–32.

[31] Lee S-W. Recognizing circuit symbols with attributed graph matching. In: Baird H, Bunke H, Yamamoto K, editors. Structured document image analysis, 1992. p. 340–58.

[32] Keating JP, Mason RL. Some practical aspects of covariance estimation. Pattern Recognition Letters 1985; 3(5):295–350.

[33] Calhoun C, Stahovich TF, Kurtoglu T, Kara LB. Recognizing multi-stroke symbols. Sketch Understanding. Papers from the 2002 AAAI Spring Symposium, 2002. p. 15–23.

[34] Gross MD, Do EY-L. Demonstrating the electronic cocktail napkin: a paper-like interface for early design. 'Common Ground' appeared in ACM Conference on Human Factors in Computing (CHI), 1996. p. 5–6.

[35] Jacob RJK, Deligiannidis L, Morrison S. A software model and specification language for non-WIMP = user interfaces. ACM Transactions on Computer-Human Interaction 1999; 6(1):1–46 URLciteseer.nj.nec.com/jacob99software.html.
[36] Costagliola G, Tortora G, Orefice S, Lucia D. Automatic generation of visual programming environments. IEEE Computer 1995; 56–65.
[37] Veselova O. Perceptually based learning of shape descriptions. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, 2003.
[38] Hammond T, Davis R. Shady: a shape description debugger for use in sketch recognition. AAAI Fall Symposium on Making Pen-Based Interaction Intelligent and Natural.