In Proceedings of IJCAI-03

LADDER: A Language to Describe Drawing, Display, and Editing in Sketch Recognition

Tracy Hammond and Randall Davis
Massachusetts Institute of Technology
200 Technology Square, NE43
Cambridge, MA 02139
{hammond, davis}@ai.mit.edu

Abstract

We have created LADDER, the first language to describe how sketched diagrams in a domain are drawn, displayed, and edited. The difficulty in creating such a language is choosing a set of predefined entities that is broad enough to support a wide range of domains, while remaining narrow enough to be comprehensible. The language consists of predefined shapes, constraints, editing behaviors, and display methods, as well as a syntax for specifying a domain description sketch grammar and extending the language, ensuring that shapes and shape groups from many domains can be described. The language allows shapes to be built hierarchically (e.g., an arrow is built out of three lines), and includes the concept of "abstract shapes", analogous to abstract classes in an object oriented language. Shape groups describe how multiple domain shapes interact and can provide the sketch recognition system with information to be used in top-down recognition. Shape groups can also be used to describe "chain-reaction" editing commands that effect multiple shapes at once. To test that recognition is feasible using this language, we have built a simple domain-independent sketch recognition system that parses the domain descriptions and generates the code necessary to recognize the shapes.

1 Introduction

To date, sketch recognition systems have been domain-specific, with the recognition details of the domain hard-coded into the system. Developing such a sketch interface is a substantial effort. We propose instead that recognition be performed by a single domain-independent recognition system that uses a domain specific sketch grammar (an approach used with some success in speech recognition [Zue *et al.*, 1990; Hunt and McGlashan, 2002]). Programmers could then create new sketch interfaces simply by writing a sketch grammar describing the domain-specific information.

We have created LADDER, a sketch description language that can be used to describe how shapes and shape groups are drawn, edited, and displayed. These 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 and extending the language. 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.

Our contribution is in creating LADDER, the first sketching language to incorporate editing, display, and shape group information. To test our language, we have built a simple domain-independent sketch recognition system that parses the domain description and successfully recognizes shapes based on these descriptions.

Section 2 describes the components of the language, including the predefined shapes, constraints, editing behaviors, and display methods available. Section 3 describes the syntax and content of a sketch grammar designed in the language. Section 4 describes a system we have implemented to test the language and ensure that shapes in a domain can be recognized based on their descriptions. Section 5 describes related work done in the development in sketch languages.

2 Language Contents

The language consists of predefined shapes, constraints, editing behaviors, and display methods. Figure 2 shows an example description for **OpenArrow** drawn in (Figure 1). The description of a shape contains a list of *components* (the elements from which the shape is built), geometric *constraints* on those components, a set of *aliases* (names that can be used to simplify other elements in the description), *editing* behaviors (how the object should react to editing gestures), and *display* methods indicating what to display when the object is recognized.

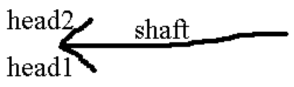

Figure 1: An open arrow.

```
(define shape OpenArrow
  (description "An arrow with an open head")
  (components
    (Line shaft)
    (Line head1)
    (Line head2))
  (constraints
    (coincident shaft.p1 head1.p1)
    (coincident shaft.p1 head2.p1)
    (coincident head1.p1 head2.p1)
    (equal-length head1 head2)
    (acute-meet head1 shaft)
    (acute-meet shaft head2))
  (aliases
    (Point head shaft.p1)
    (Point tail shaft.p2))
  (editing
   ( (trigger (click_hold_drag shaft))
     (action
       (translate this)
       (set-cursor DRAG)
       (show-handle MOVE tail head)))
    ( (trigger (click_hold_drag head))
      (action
        (rubber-band this head tail)
        (show-handle MOVE head)
        (set-cursor DRAG)))
   ( (trigger (click_hold_drag tail))
     (action
       (rubber-band this tail head)
       (show-handle MOVE tail)
       (set-cursor DRAG))))
  (display (original-strokes)))
```

Figure 2: The description for an arrow with an open head

The power of the language is derived in part from carefully chosen predefined building blocks.

2.1 Predefined Shapes

The language includes a number of predefined primitive and non-primitive shapes, usable as building blocks in describing other shapes. The primitive shapes are **Shape, Point, Path, Line, BezierCurve**, and **Spiral**. **Circle, Arc**, and **Ellipse** are examples of non-primitive shapes included in the language library; all three are more specific versions of the primitive shape **Spiral**. The **OpenArrow** in Figure 2 is a non-primitive shape built out of three primitive shapes.

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.

2.2 Predefined Constraints

New shapes are defined in terms of previously defined shapes and constraints between them. For instance, the **OpenArrow** in Figure 2 contains the constraint (*acute-meet head1 shaft*), which indicates that *head1* and *shaft* meet at a point and form an acute angle in a counter-clockwise direction from *head1* to *shaft*. (Angles are measured in a counter-clockwise direction.)

A number of predefined constraints are included in the language, including: *perpendicular, parallel, collinear, same-side, opposite-side, coincident, connected, meet, intersect, tangent, contains, concentric, larger, near, draw-order, equal-length*, $=$, $<$, $<=$, *angle, angle-dir, acute, obtuse, acute-meet*, and *obtuse-meet*. 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, pos-slope, neg-slope, left-of, right-of, above, below, same-H-pos, same-V-pos, above-left, above-right, below-left, below-right, centered-below, centered-above, centered-left, centered-right*, and *angleL*, where (*angleL line1 degrees*) specifies that the angle between a horizontal line pointing right and *line1* is *degrees*.

There is an additional constraint: *is-rotatable*, which implies the shape can be found in any orientation. If *is-rotatable* 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 *angle-meet* constraints could have been replaced with:

```
(is-rotatable) (horizontal shaft)(neg-slope head1) (pos-slope
head2) (left-of shaft.p1 shaft.p2) (left-of head1.p2 shaft.p2)
(left-of head2.p2 shaft.p2),
```

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

2.3 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*, defines that if you *click-hold-drag* the pen starting inside of the bounding box of a shape, the entire shape automatically moves with it.

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.

For instance, in Figure 2, the **OpenArrow** contains three editing behaviors. The first editing behavior says that if you click and hold the pen over the **shaft** of the **OpenArrow**, when you drag the pen, the entire **OpenArrow** 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 **OpenArrow** will stretch like a rubber band (translating, scaling, and rotating) to satisfy these two constraints and keep the **OpenArrow** 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 performing an editing command.

The possible editing actions include *wait, select, deselect, color, delete, translate, rotate, scale, resize, rubber-band, show-handle*, and *set-cursor*. To give an example:

```
(rubber-band 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 that 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, double-click, click-hold, click-hold-drag, draw, draw-over, scribble-over*, 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 arrow head inside of this rectangle, the arrow should move with the rectangle.

2.4 Predefined Display Methods

An important part of a sketching interface is controlling what the user sees after shapes are 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. 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 noise from sketching is removed.

It may be that we don't 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, or we can create a picture built out of predefined shapes, such as circles, lines, and rectangles.

The pre-defined display methods include: *original-strokes, cleaned-strokes, ideal-strokes, circle, line, point, rectangle, text, color*, and *image*. Each method includes color as an optional argument.

3 Specifying a Domain Description

A domain description contains a list of the domain shapes and shape groups, as well as definitions for each of them. Descriptions can be hierarchical and can refer to other shapes in the language. This section provides examples from the domain description sketch grammar of UML (Unified Modelling Language) class diagrams [Booch *et al.*, 1998].

3.1 Indicating Domain Definitions

The compiler uses a list of domain shapes and shape groups to confirm that each shape is properly defined and to speed recognition by creating recognizers only for sub-shapes needed by the domain.

3.2 Defining Shapes

A domain shape is a shape that is meaningful in the domain. Geometric shapes usually occur in several domains and are the building blocks of the domain shapes. For instance, in the domain of UML class diagrams, the domain shapes (followed by their geometric shape component) are: general classes (represented by rectangles), interface classes (circles), interface associations (lines), dependency associations (open-headed arrows), aggregation associations (diamond-headed arrows), inheritance associations (triangle-headed arrows), information associations (dotted lines or dotted open arrows).

A shape definition 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. A shape definition is composed of seven sections. All sections are optional except the *components* section.

1. The *description* contains a textual description of the shape, e.g., "an arrow with a triangle-shaped head."
2. The *is-a* section specifies any class of abstract shapes (Section 3.3) that the shape may be a part of. This is similar to the extends property in Java. All shapes extend the abstract shape **Shape**.
3. The *components* section lists the components of the shape. For example, the **TriangleArrow** in Figure 3 is built out of the **OpenArrow** from Figure 2 and a **Line**. Components can be accessed hierarchically.
4. The *constraints* section specifies relationships between the components. For example, in the **TriangleArrow** in Figure 3, *(coincident head3.p1 head1.p2)* specifies that an endpoint of *head3* and an endpoint of *head1* are located at the same point.

   The *constraints* section can specify both hard constraints, such as the one listed above, and soft constraints, which are specified by the keyword *soft*. Hard constraints are always satisfied in the shape, but soft constraints may not be. Soft constraints can aid recognition by specifying relationships that usually occur. For instance, in Figure 3 the shaft of the arrow is commonly drawn before the head of the arrow, but the arrow should still be recognized even if this constraint is not satisfied.
5. The *aliases* section allows us to compute certain properties and name them for use later. For instance, in Figure 3, *head1* is defined and used in a constraint for simplicity. Components specified in the *aliases* section can be accessed hierarchically. For instance, **TriangleArrow** uses *head* and *tail* from the **OpenArrow** in Figure 2.
6. An *editing* section specifies how the shape can be edited. Common editing commands involve movement

and deletion of the shape. Each editing behavior must specify a trigger and an action listed in Section 2.3.

Shapes can be defined to be moved as a whole rather than having to move individual lines. For instance, in Figure 3, one editing behavior for the **TriangleArrow** indicates that if the user presses and holds the pen on the **shaft** for a brief period, the pen will drag the entire **TriangleArrow** when moved.

7. A *display* section specifies what should be displayed on the screen when the shape is recognized. This section is generally included only for domain shapes, not for geometric shapes. In the **UMLInheritanceAssociation** in Figure 4, the arrow will be displayed using straight lines for the arrow head and the original stroke for the shaft.

```
(define shape TriangleArrow
  (description   "An arrow with a triangle-shaped head")
  (components
    (OpenArrow oa)
    (Line head3))
  (aliases
    (Line shaft oa.shaft)
    (Line head1 oa.head1)
    (Line head2 oa.head2)
    (Point head oa.head)
    (Point tail oa.tail))
  (constraints
    (coincident head3.p1 head1.p2)
    (coincident head3.p2 head2.p2)
    (soft draw-order shaft head1)
    (soft draw-order shaft head2))
  (editing
   ( (trigger (click_hold_drag shaft))
     (action
       (translate this)
       (set-cursor DRAG)
       (show-handle MOVE tail head)))...))
```

Figure 3: The description for an arrow with a triangle-shaped head.

```
(define shape UMLInheritanceAssociation
  (is-a UMLGeneralAssociation)
  (components
    (TriangleArrow arrow))
  (aliases
    (Point head arrow.head)
    (Point tail arrow.tail)
    (Line shaft arrow.shaft))
  (display
    (original_strokes arrow.shaft)
    (cleaned_strokes arrow.head1 arrow.head2 arrow.head3))
```

Figure 4: The domain shape UML Inheritance Association is defined by the geometrical shape **TriangleArrow** from Figure 3.

3.3 Defining Abstract Shapes

In the **UMLInheritanceAssociation** defined in Figure 4, the *is-a* section specifies that the **UMLInheritanceAssociation** is an extension of the abstract shape **UMLGeneralAssociation**. Abstract shapes have no concrete shape associated with them; they represent a class of shapes that have similar attributes or editing behaviors. These attributes can be defined once in the abstract shape description rather than for each domain shape. For instance, in Figure 2 and Figure 3, notice that the **OpenArrow** and the **TriangleArrow** have identical editing behaviors defined. Rather than repeatedly listing these editing behaviors in each shape, we could create an abstract shape which specifies these 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.

Figure 5 presents a diagram of the inheritance hierarchy for the abstract and non-abstract shapes in the UML class diagrams domain. In UML, **UMLDependencyAssociation**, the **UMLInheritanceAssociation**, the **UMLAggregationAssociation**, the **UMLInformationAssociation**, and the **InterfaceAssociation** are all links represented by arrows or lines and all have the same editing behavior. Thus, we can create the abstract shape **UMLAssociation**, which lists the editing behavior of these shapes. Figure 6 shows the abstract shape description of the **UMLAssociation**. Notice that the required variables are used when defining editing behaviors.

```
(define abstract-shape UMLAssociation
  (is-a Shape)
  (required
    (Point head)
    (Point tail)
    (Line shaft))
  (editing
   ( (trigger (click_hold_drag shaft))
     (action
       (translate this)
       (set-cursor DRAG)
       (show-handle MOVE tail head)))...))
```

Figure 6: The description for the abstract class UMLAssociation.

3.4 Defining 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.

In the domain of UML class diagrams, there are six legal shape groups that describe the visual relationship between UML associations and UML classes. For example, one shape group consists of an association combined with a general class, such that the tail of the association is inside or near the general class shown in Figure 7 and described in Figure 8. A shape group definition is similar to that of a shape definition.

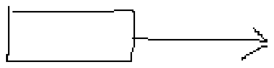

Figure 7: An association attached to a class at its tail.

If a single shape in a sketch can be part of many instances of a shape group, then we place the key word *multiple* before the component shape of the shape group. In UML Class Diagrams, for example, a single **UMLAssociation** can be part of only one instance of a shape group, while a single **UMLClass** can be part of many instances of **UMLGenClassGenAssociationTail**.

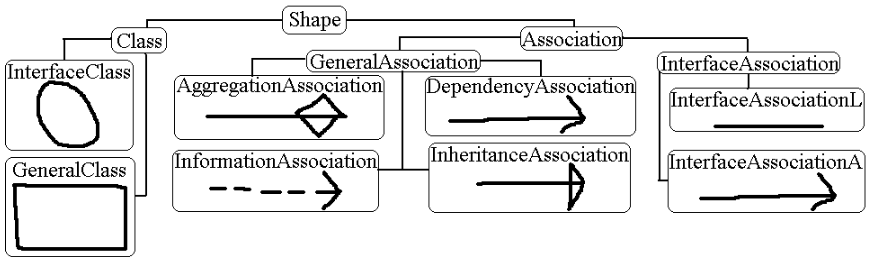

Figure 5: The inheritance diagram of UML Class Diagram shapes.

```
(define shape-group
   UMLGenClassGenAssociationTail
  (description "A general class attached to
    the tail of a general association")
  (is-a UMLAssociationAttachedTail)
  (components
    (multiple (GeneralClass ct))
    (GeneralAssociation r)))
```

Figure 8: Description of the shape group from Figure 7.

3.5 Defining Abstract Shape Groups

Abstract shape groups definitions allow the reuse of shared properties across multiple shape groups. The definition of a **UMLAssociationAttachedTail** in Figure 9 indicates that the tail, but not the head, of the association is inside the class, preventing us from having to redefine the constraints many times, and allows us to define one general editing behavior for many shapes. An editing behavior for the **UMLAssociationAttachedTail** indicates that whenever you move a **UMLClass** that is attached to the tail of a **UMLAssociation**, the head of the **UMLAssociation** remains fixed in its original location, but the tail of the **UMLAssociation** remains attached to the **UMLClass** as it moves; the **UMLAssociation** acts like a rubber band (translating, scaling, and rotating) to satisfy these constraints.

```
(define abstract-shape-group
   UMLAssociationAttachedTail
  (required
    (Association r)
    (Class ct))
  (constraints
    (contains ct r.tail)
    (!contains ct r.head))
  (editing
    (trigger (translate ct))
    (action (rubber-band r r.head r.tail))))
```

Figure 9: Definition for an abstract shape group.

3.6 Defining Constraints

We believe the language contains sufficient constraints to define a broad range of domains. When an additional constraint is needed, it can be defined using a macro facility. The sections of a sketch-constraint definition include *description*, *components*, and *constraints*.

```
(define constraint centered-in
  (description "Tests if shape1 is centered inside shape2")
  (components
    (Shape s1)
    (Shape s2))
  (constraints
    (contains s2 s1)
    (coincident s1.center s2.center)))
```

Figure 10: Definition for the constraint centered-in.

4 Testing

4.1 Examples of Shapes Described in the Language

We have evaluated the language by showing that it can describe a wide variety of symbols from a number of different domains. We have used it to describe over a hundred shapes from the domains of UML class diagrams, mechanical engineering diagrams, course of action diagrams, and letters of the alphabet. Illustrative examples are given below.

Polygon

A **PolyLine** (shown in Figure 11a), 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**

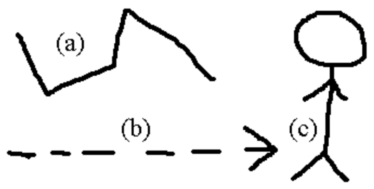

Figure 11: (a)Poly Line (b) Dashed Open Arrow (c) Stick Figure

```
(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)))
```

Figure 12: Shape Description of a Polygon.

Dashed Open Arrow

A **DashedOpenArrow** (Figure 11b) is made from a **OpenArrow**, and a **Dashedline**, 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.

Stick Figure

The definition of a stick figure (Figure 11c) shows how we can create new components to help describe shapes. It creates

```
(define shape DashedLine
  (components (vector Line vl[2,n]))
  (constraints (collinear vl[i].p1 vl[i].p2 vl[i+1].p1)
    (!(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)))
```

Figure 13: Description of a dashed line and a dashed open arrow.

a new line between the feet for use in defining constraints, ensuring that both feet lie below the body.

```
(define shape StickFigure
  (description "a stickfigure with two arms and two legs
    all sloping down at 45 degrees")
  (components (Circle head)(Line body)
    (Line larm)(Line rarm)(Line lleg)(Line rleg))
  (alias (Line feet_space (new Line (lleg.p2 rleg.p2))))
  (constraints (meet head body.p1)(!(intersect body head))
    (is-rotatable) (vertical body) (meet body larm.p1)
    (meet body rarm.p1) (coincident larm.p1 rarm.p1)
    (acute larm body)(acute body rarm)(left-of larm rarm)
    (coincident body.p2 lleg.p1)(coincident body.p2 rleg.p1)
    (obtuse body lleg)(obtuse rleg body)
    (perpendicular larm rarm)(perpendicular lleg rleg)
    (near body.p1 rarm.p1)(parallel rarm rleg)
    (parallel larm lleg)(!(intersect feet_space body))
    (equal-length lleg rleg)
    (equal-length larm rarm))
```

Figure 14: Description of a stick figure.

4.2 System Implementation

We built a simple domain-independent recognition system to test whether sketches can be recognized from our domain descriptions. The system parses a domain description into Java code and Jess (a rule-based system that interfaces with Java) [Friedman-Hill, 1995] rules, and uses them to recognize sketches. For example, using the domain description for UML, the system successfully recognized hand-drawn sketches of all of the shapes in Figure 5 regardless of overlap.

Domain Description Parsing

The domain description is parsed to create recognition code, creating at least one Jess rule (containing the shape recognition information), and one Java file (describing the shape), for each shape description. The system then uses the Jess rules to recognize sketches.

Jess Rule Example

The rule automatically generated for the **TriangleArrow** from Figure 3 is shown in Figure 15. If a shape description contains a vector, such as that of the **DashedArrow** in Figure 13, two Jess rules are created, one containing the base case, and the second containing a recursive rule.

Stroke Preprocessing

The recognition system has several stages of recognition. First, each time a stroke is drawn, the stroke is pre-processed [Sezgin *et al.*, 2001] into either Point, Line, Curve, Arc, or a combination thereof, allowing users to draw objects in a single stroke or with multiple strokes. These primitive shapes are then asserted to a Jess database.

```
(defrule TriangleArrowCheck
  ;;Get the parts of the triangle arrow
  ?f0 <- (Subshapes OpenArrow ?oa $?oa_list)
  ?f1 <- (Subshapes Line ?head3 $?head3_list)
  (OpenArrow ?oa ?oa_shaft ?oa_head1 ?oa_head2 ?oa_head ?oa_tail)
  (Line ?head3 ?head3_p1 ?head3_p2)

  ;;Make sure that the openarrow and line don't share any subparts.
  (test (uniquefields $?oa_list $?head3_list))

  ;; Get the subpart of the open arrow since they are referenced
  (Line ?oa_shaft ?shaft_p1 ?shaft_p2)
  (Line ?oa_head1 ?head1_p1 ?head1_p2)
  (Line ?oa_head2 ?head2_p1 ?head2_p2)

  ;; test for that the constraints hold
  (test (coincident ?head3_p1 ?head1_p2))
  (test (coincident ?head3_p2 ?head2_p2))
=>
  ;; Triangle Arrow has been successfully found
  ;; Set the aliases
  (bind ?shaft ?oa_shaft)(bind ?head1 ?oa_head1)
  (bind ?head2 ?oa_head2)(bind ?head ?oa_head)
  (bind ?tail ?oa_tail)

  ;; Tell the recognition system that there is a Triangle Arrow
  (bind ?nextnum (addshape TriangleArrow ?oa $?head3_list ?shaft
    ?head1 ?head2 ?head ?tail))

  ;; Tell the Jess system that there is a Triangle arrow
  (assert (TriangleArrow ?nextnum  ?oa ?head3 ?shaft ?head1 ?head2
    ?head ?tail))
  (assert (Subshapes TriangleArrow ?nextnum (union$ $?oa_list
    $?head3_list)))

  ;; Triangle Arrow is a domain shape. Assert it.
  ;; Conflicts will be resolved elsewhere.
  (assert (DomainShape TriangleArrow ?nextnum (time)))
  (assert (CompleteSubshapes OpenArrow ?oa $?oa_list)))
```

Figure 15: Automatically generated Jess Rule for the Triangle Arrow.

Recognition of Shapes

Recognition is handled by the Jess rule based system. We have automatically generated templates for the Jess system to fill in. Once Jess finds the appropriate components, the rule is fired and the constraints are tested. The constraints are Java functions with which Jess interacts. All possible shapes are found, even if the shapes share lines or other components. If the shape can be a domain shape (i.e., a final shape in the domain), the shape is asserted as a domain shape.

Domain Shapes

When domain shapes are created, a rule fires in Jess confirming that no two found domain shapes share the same subcomponents. If two domain shapes do share subcomponents, one domain shape is retracted. The domain shape chosen to remain is the one containing more primitive components (following Ockham's Razor); if the two shapes contain the same number of components, the shape created first is chosen, since previously chosen recognitions have higher precedence. If a domain shape is found, then the recognition system displays the shape as the designer specified and editing commands can then be performed on the shape.

5 Related Work

Shape description languages, such as shape grammars, have been around for a long time [Stiny and Gips, 1972]. Shape grammars are studied widely within the field of architecture, and many systems are continuing to be built using shape grammars [Gips, 1999]. However, shape grammars were developed for shape generation rather than recognition, and don't provide for non-graphical information, such as stroke order, that may be helpful in recognition. They also lack ways for specifying shape editing.

Within the field of sketch recognition, there have been other attempts to create languages for sketch recognition. Bimber et. al [2000] describe a simple sketch language using a BNF-grammar. 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 [2002] uses a language to model and recognize stick figures. The language currently is not hierarchical, making large objects cumbersome to describe. Caetano et. al. [2002] use fuzzy relational grammars to describe shape. Both Mahoney and Caetano lack the ability to describe editing, display, or shape grouping information.

The Electronic Cocktail Napkin project [Gross and Do, 1996] 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 [Jacob *et al.*, 1999] 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 very low-level making it difficult to define new interactions, and, in the domain of sketching, does not provide a significant improvement to coding the domain-dependent recognition system from scratch.

The language described in this paper is being used in several other systems, including sketch learning from example [Veselova, 2002], smart compiling using HMM's [Sezgin, 2002], and an intelligent Bayesian network context oriented sketch recognition system [Alvarado *et al.*, 2002].

6 Conclusion

6.1 Future Work

We plan to examine more domains to ensure that the language contains the appropriate primitives. We would also like to test our syntax on a wide user base. Large domains benefit from visual diagrams, such as the one in Figure 5. We plan to automatically generate some of these visual diagrams to help with the grammar-writing process.

6.2 Contributions

We have created LADDER, the first language to describe how sketched diagrams in a domain are drawn, displayed, and edited. The language consists of pre-defined shapes, constraints, editing-behaviors, and display methods, as well as a syntax for specifying a sketch grammar and extending the language, ensuring that shapes and shape groups from many domains can be described. The syntax simplifies the definition of new shapes by allowing shapes to be built hierarchically and by providing abstract shapes to contain common shape properties. Shape groups describe how domain shapes interact, and can provide information to be used in top-down as well as bottom-up recognition. Shape groups can also be used to describe "chain-reaction" editing commands. We have built a simple domain-independent sketch recognition system for testing that recognition is feasible based on the descriptions provided.

Acknowledgments

This work is supported in part by the MIT Oxygen Collaboration. The authors would like to thank Jacob Eisenstein for the name LADDER and Jacob Eisenstein, Michael Oltmans, Mark Foltz, and Vineet Sinha for their help in reviewing this paper.

References

[Alvarado *et al.*, 2002] C Alvarado, M Oltmans, and R Davis. A framework for multi-domain sketch recognition. *AAAI Spring Symposium on Sketch Understanding*, pages 1–8, March 25-27 2002.

[Bimber *et al.*, 2000] O Bimber, LM Encarnao, and A Stork. A multi-layered architecture for sketch-based interaction within virtual environments. *Computer and Graphics*, 2000.

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

[Caetano *et al.*, 2002] A Caetano, N Goulart, M Fonseca, and J Jorge. Javasketchit: Issues in sketching the look of user interfaces. *AAAI Spring Symposium on Sketch Understanding*, 2002.

[Friedman-Hill, 1995] E Friedman-Hill. Jess, the rule engine for the java platform. http://herzberg.ca.sandia.gov/jess/, 1995.

[Gips, 1999] J Gips. Computer implementation of shape grammars. *NSF/MIT Workshop on Shape Computation*, 1999.

[Gross and Do, 1996] MD Gross and EYL Do. Demonstrating the electronic cocktail napkin: a paper-like interface for early design. *ACM Conference on Human Factors in Computing*, pages 5–6, 1996.

[Hunt and McGlashan, 2002] A Hunt and S McGlashan. Speech recognition grammar specification version 1.0, w3c candidate recomentation. http://www.w3.org/TR/speech-grammar, 26 June 2002.

[Jacob *et al.*, 1999] RJK Jacob, L Deligiannidis, and S Morrison. A software model and specification language for non-WIMP user interfaces. *ACM Transactions on Computer-Human Interaction*, 6(1):1–46, 1999.

[Mahoney and Fromherz, 2002] JV Mahoney and MPJ Fromherz. Three main concerns in sketch recognition and an approach to addressing them. *AAAI Spring Symposium on Sketch Understanding*, pages 105–112, March 25-27 2002.

[Sezgin *et al.*, 2001] TM Sezgin, T Stahovich, and R Davis. Sketch based interfaces: Early processing for sketch understanding. *Perceptive User Interfaces Workshop*, 2001.

[Sezgin, 2002] M Sezgin. Generating domain specific sketch recognizers from object descriptions. *MIT Student Oxygen Workshop*, 2002.

[Stiny and Gips, 1972] G Stiny and J Gips. Shape grammars and the generative specification of painting and sculpture. *Information Processing*, pages 1460–1465, 1972.

[Veselova, 2002] O Veselova. Perceptually based learning of shape descriptions from one example. *MIT Student Oxygen Workshop*, 2002.

[Zue *et al.*, 1990] V Zue, J Glass, M Phillips, and S Seneff. The mit speech recognition system: phonological modeling and lexical access. *1990 International Conference on Acoustics, Speech and Signal Processing*, pages 49–52, March 25-27 1990.