Automatically Transforming Symbolic Shape Descriptions for Use in Sketch Recognition

Tracy Hammond and Randall Davis
MIT Computer Science and Artificial Intelligence Laboratory (CSAIL)
MIT Building 32-(239,237), 32 Vassar St.
Cambridge, MA 02139
{hammond,davis} at csail.mit.edu

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. This paper presents the first translator that takes symbolic shape descriptions (written in the LADDER sketch language) and automatically transforms them into shape recognizers, editing recognizers, and shape exhibitors for use in conjunction with a domain independent sketch recognition system. This transformation allows us to build a single domain independent recognition system that can be customized for multiple domains. We have tested our framework by writing several domain descriptions and automatically created a domain specific sketch recognition system for each domain.

Introduction

As pen-based input devices have become more common, sketch recognition systems are being developed for many domains such as mechanical engineering (Alvarado 2000), UML class diagrams (Hammond & Davis 2002), webpage design (Lin *et al.* 2000), architecture (Gross, Zimring, & Do 1994), GUI design (Caetano *et al.* 2002a; Lecolinet 1998), virtual reality (Do 2001), stick figures (Mahoney & Fromherz 2002), course of action diagrams (Pittman *et al.* 1996), and many others. These systems allow users to sketch a design, which is a more naturally interaction than a traditional mouse and palette tool (Hse *et al.* 1999). But sketch recognition systems can be quite time consuming to build if they are to handle the intricacies of each domain.

We propose that rather than build a separate recognition system for each domain, we instead build a single domain independent recognition system that can be customized for each domain. To build a sketch recognition system for a new domain, the developer would need only write a domain description, describing how shapes are drawn, displayed and edited. This description would then be transformed for use in the domain independent system. The inspiration for such a framework stems from work in speech recognition, which has been using this approach with some success (Zue & Glass 2000).

In our work, we transform a grammar into a domain recognizer of hand-drawn shapes. This is analogous to work done on compiler compilers, in particular visual language compiler compilers (Costagliola *et al.* 1995). A visual language compiler compiler allows a user to specify a grammar for a visual language, then compiles it into a recognizer which can indicate whether a arrangement of icons is syntactically valid. The main difference between this 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.

In this paper we present the first translator that takes symbolic descriptions of how shapes are drawn, displayed, and edited in a domain and automatically transforms them into shape recognizers, editing recognizers, and shape exhibitors for use in a domain independent sketch recognition system. To succeed in our goal, we have created 1) LADDER(Hammond & Davis 2003), a symbolic language for describing how shapes are drawn, displayed, and edited in a domain, 2) a translator as described above, and 3) a simple domain independent recognition system that uses the newly translated components to recognize, display, and allow editing of the domain shapes. The implementation of this translator and domain independent sketch recognition system serves to show both that such a framework is feasible and that LADDER is an acceptable language for describing domain information.

The domain independent recognition system and transformer described here were designed to test whether we could in fact transform symbolic descriptions of a domain into active recognizers usable by a domain independent recognition system. Other work in our group is pursuing a more ambitious approach to both building a domain independent recognition system and studying the process of transforming descriptions into recognizers. Nevertheless, the work reported here does illustrate the plausibility of transforming descriptions into recognizers.

We have chosen a symbolic sketching language based on how shapes look rather than on features such as drawing speed, size of the bounding box, etc., (as in systems like (Rubine 1991; Long 2001)).We did this to ensure that symbols would be recognized if they looked the same, even if they weren't drawn in the same way (e.g., with a different number of strokes), allowing users to draw the shapes as they

would naturally. A high-level symbolic language based on shape offers the added advantage in being easier to read and understand, facilitating identification and correction of errors in the description such as automatically checking if a shape is impossibly constrained (which would be difficult in a low-level languages such as (Jacob, Deligiannidis, & Morrison 1999)). Shape definitions primarily concern how shapes look, but may include other information helpful to the recognition process, such as stroke order or stroke direction.

Because different domains have different ways of displaying and editing the shapes in their domain, sketch recognition systems need to know how to edit and display the shapes recognized. This motivated us to create a language that allows developers to describe editing and display, as well as how the shapes look and are drawn.

Shape description languages have been created for use in architecture, diagram parsing, as well as within the field of sketch recognition itself. However, current shape description languages lack ways for describing editing (Stiny & Gips 1972; Futrelle & Nikolakis 1995; Bimber, Encarnacao, & Stork 2000; Mahoney & Frommerz 2002; Caetano *et al.* 2002b; Gross & Do 1996), display (Futrelle & Nikolakis 1995; Bimber, Encarnacao, & Stork 2000; Mahoney & Frommerz 2002; Caetano *et al.* 2002b; Gross & Do 1996), or non-graphical information, such as stroke order or direction (Stiny & Gips 1972; Futrelle & Nikolakis 1995; Bimber, Encarnacao, & Stork 2000).

Framework Overview

Our goal is to make development of a sketch recognition system easier by enabling domain experts (rather than programmers) to describe the shapes to be recognized. Figure 1 gives an overview of the framework for our overall research effort: 1) a sketch description language, LADDER, 2) a translator that converts a domain description into components for use in conjunction with a domain independent sketch recognition system, and 3) a domain independent sketch recognition system that uses the newly generated components to recognize, edit, and display shapes in the domain. The domain description is transformed into shape recognizers, exhibitors, and editors which are used in conjunction with a domain independent recognition system to create a domain specific recognition system.

To create the domain specific recognition system, the developer writes a LADDER domain description consisting of multiple shape definitions. The left box of Figure 1 gives an example of an Arrow shape definition. The components and the constraints define what the shape looks like and are transformed into shape recognizers. The display section specifies how the shape is to be displayed when recognized and is transformed into shape exhibitors. The editing section specifies the editing behaviors that can be performed on the recognized shape and are transformed into editing recognizers.[1]

[1]The aliases section renames components or sub-components for ease of reference later.

LADDER supplies a number of predefined shapes, constraints, display methods, and editing behaviors. These predefined elements are hand-coded into the domain independent system, allowing it to recognize, display, and edit these predefined shapes. Recognition is carried out as a series of bottom up opportunistic data driven triggers in response to pen strokes.

The third box of Figure 1 shows the domain independent sketch recognition system, which contains hand-coded shape recognizers, editing recognizers, and shape exhibitors for the primitive shapes (line, ellipse, curve, arc, and point). This system also defines each of the constraints. The translator creates additional shape recognizers (which in turn call on the constraint functions), editing recognizers, and shape exhibitors. As each stroke is drawn, the system determines whether the stroke is an editing trigger for any shape. If not, it is taken to be part of the drawing, and is recognized as a collection of primitive shapes. The resulting primitives are added to the database, and the recognition module examines the database to attempt to combine the primitives into more abstract shapes. Specialized methods merge overlapping and connecting lines to account for primitives such as lines being drawn using more than one stroke. The display module then displays the result as defined by the domain description.

The domain independent recognition system, including the set of primitive shapes, constraints, display routines, and editing gesture handlers, and the links between them, provides a substantial foundation for the domain specific recognition system, helping to greatly simplify the translation process.

Transformation

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. We describe the translation process in detail for each part of the shape definition.

Generating Shape Recognizers

The job of the shape parser is to transform a shape definition into rules that recognize that shape. The shape definition specifies the components that make up the shape as well as the constraints on these components, including any requirements about stroke order or direction.[2] We use the Jess (Friedman-Hill 2001) rule engine for recognition.

Jess is a forward-chaining pattern/action rule engine for Java. In our system, Jess facts represent recognized drawn shapes. Each stroke is segmented into a point, line, curve,

[2]LADDER allows the user 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 Figure 1, we could have specified (draw-order shaft head1 head2) to specify that the the shaft of the arrow is commonly drawn before the head, but the arrow should still be recognized even if this is not satisfied. Our current implementation does not yet support soft constraints.

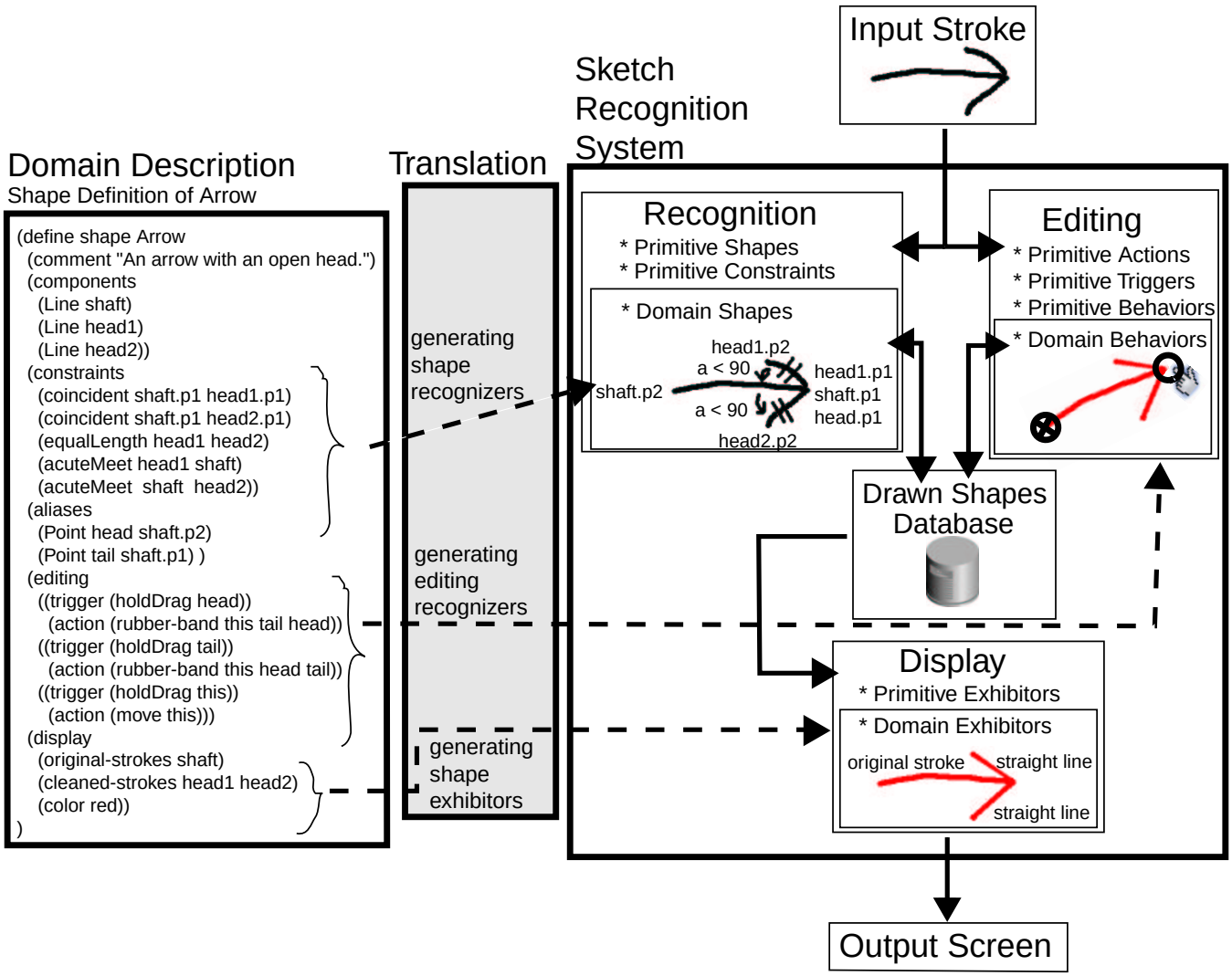

Figure 1: Framework Overview showing LADDER Domain Description, Translator, and Domain Independent Sketch Recognition System.

arc, ellipse, spiral, or some combination using techniques from (Sezgin 2001), and the primitive shapes are added to the Jess fact database. For instance, if a line is drawn, a fact is added of the form `(Line 342 23 24 25 10.6)`, where 342 indicates the line's ID, 23, 24, and 25 are the IDs of the endpoints and midpoint, and 10.6 indicates the length of the line. [3] An additional fact, `(Subshapes Line 342 342)`, is also sent to indicate the primitive shapes that make up the drawn shape.

Our domain shape recognizers are implemented as Jess rules. In the transformation process we create a rule whose pattern specifies the components of the shape and the constraints that must hold between the components.[4] An example of the rule generated for the arrow given in the left box in Figure1 is shown in Figure 2. The translation process is straightforward because of the foundation of primitives built into the domain independent system. The rule engine searches for all possible subsets of facts for the collection specified in its premise. In the case of Figure 1, the rule engine searches for three lines to make up the shaft, head1, and head2. The rule engine then tests whether the constraints hold for each subset. Then, for each collection of three lines labelled shaft, head1, and head2, the Jess engine will check that head1.p1 and shaft.p1 are coincident. Each LADDER constraint is defined as a Jess function to simplify transfor-

[3] Because we do not want to place any unspecified drawing order constraints, each line, arc, and curve is actually added twice to the Jess rule based system to take into account the fact that the endpoints may be assigned in either direction.

[4] The pattern also specifies that all components be distinct, to prevent the rule engine from returning three copies of the same line when trying to find an arrow.

```
(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
)
```

Figure 2: Automatically Generated Jess rule for the arrow definition in the left box of Figure 1.

mation of a shape definition into a Jess rule.

If a shape is recognized (i.e., the rule pattern is satisfied), the rule action is fired. In the transformation process the rule action is specified to do three things: 1) add the new shape to the database of recognized shapes so it can be displayed correctly and edited; 2) add the new fact to the rule based system so more complicated shapes can be formed from it; 3) remove recognized components from the database. This last step improves efficiency at the expense of possibly missing some shape interpretations.

Jess then confirms that this newly created shape does not share any subcomponents with any other domain shape. If it does, then only one of the domain shapes will remain, either the shape containing more primitive components (Ockhams razor), or (in the case of a tie) the shape that was drawn first.

While the arrow example used throughout this paper is composed of a fixed number of components (3 lines), our architecture can also support shapes composed of a variable number of components, such as a polyline or polygon. A shape with a variable number of components is transformed into two Jess rules, the first recognizes the base case, and the second rule handles the recursive case. For example, the base case rule of a polyline recognizes a polyline consisting of two lines, while the recursive case rule recognizes a polyline consisting of an existing polyline and an additional line.

Generating Editing Recognizers

The shape definition includes information on how a shape can be edited. The arrow definition from Figure 1 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 user places and holds their 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.

The drawing panel watches for all of the possible editing triggers predefined in LADDER. When one occurs it calls the appropriate method to check if an editing behavior should occur. Each of the editing actions (such as translate, rotate, scale, rubber-band, or delete) is predefined for all shapes.

During the translation process, we transform all of the editing specifications of the shape definitions into one Java class. After a trigger first occurs (such as click or holdDrag), the Java class examines all of the viewable shapes with that trigger defined to see if an editing behavior has begun. For instance, holdDrag is defined as the pen initially resting on the screen for .4 seconds, and then dragging across the screen. After holdDrag is detected, the system looks to see if the pen is located over the head, tail, or shaft of an arrow. If not, the system treats the stroke as a drawing gesture. If there is a shape underneath the stylus that has that trigger specified, the editing action occurs. In our example, if the head of an arrow is underneath the pen, the arrow will rubber-band with the head following the path of the pen and the tail remaining fixed. [5]

Generating Shape Exhibitors

The shape definition includes information on how a shape should be displayed once it 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[6], or 4) another custom shape which specifies which shapes (line, circle, rectangle, etc.) to drawn and where. The arrow definition from Figure 1 specifies that the arrow should be displayed in the color red, that head1 and head2 should be drawn using cleaned-strokes (a straight line in this case), and that the shaft should be drawn using the original strokes.

LADDER has a number of predefined display methods, including color, original-strokes, and cleaned-strokes, which are hand-coded into the domain independent recognition system for use with all shapes. The domain independent recognition system also defines how to handle hierarchical display of shape, and provides generalized methods for al-

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

[6] We currently solve only a subset of the constraints and are attempting to develop and integrate a more sophisticated geometric constraint solver.

tering the display of a shape, including translate, scale, and rotate.

During the transformation process we create a shape exhibitor (a Java class to display the shape) for each shape. This shape exhibitor specifies how the shape is to be drawn (original, cleaned, ideal, or custom). If the ideal strokes are to be drawn, the translator creates a method that displays the ideal strokes by attempting to solve the constraints of the shape (e.g., ensuring that points are coincident if the description indicates so) and then redraws the shape with the constraints solved. Likewise, if a custom display is defined, the translator creates a method that displays the specified shapes.

The shape exhibitor controls the displaying of the newly created shape and ensures that the components (e.g., the original strokes) are not drawn, but only the abstract shape (e.g., arrow) itself. The shape exhibitor keeps track of the location of the accessible components and aliases of a shape, which 1) can be used by the editing module to determine if an editing gesture is occurring, and 2) ensures that when a shape is moved or edited its components are moved or deleted with it. The shape exhibitor also keeps a original copy of each of the accessible components and aliases for use when scaling an object to ensure that we don't lose any precision after several scalings.

Testing

To test our approach we have built and transformed domain descriptions for a variety of domains including UML class diagrams, flow charts, finite state machines, a simplified version of course of action diagrams, and a simple subset of 2-dimensional mechanical engineering diagrams. Figure 3 shows the variety of shapes recognized to date.

Figure 4, a flowchart of the domain independent recognition system described in this paper, was created using the flowchart domain sketch recognition system. The domain description specified that Actions, Decisions, Start/Ends should be displayed in blue and the Links in red, both using cleaned-strokes. [7]

Related Work in our group

(Sezgin 2003) is working on a translation mechanism that can improve recognition efficiency using HMM techniques. (Alvarado 2003) is developing a more sophisticated domain independent recognition system using Bayesian networks to more effectively deal with the large amount of uncertainty present in messy hand-drawn sketches. (Veselova 2003) has developed a system to generate a symbolic shape definition from a single drawn example.

Conclusions

Future Work

We would like to improve our translator and domain independent recognition system to include the handling of soft constraints and continue to test it on additional domains. While we are attempting to make LADDER as intuitive as possible, shape definitions can be difficult to describe textually, and we would like to integrate work from (Veselova 2003) to automatically generated shape descriptions from a drawn example. However, even automatically generated shapes will need to be checked and modified. Thus we would like to create a GUI to debug a domain description, providing interfaces to test whether the shape is under-constrained (by automatically generating valid example shapes) or over-constrained (by allowing the user to draw several test examples).

Contributions

We suggest an innovative framework for sketch recognition that uses a single, customizable domain independent recognition system for use with many domains. This paper presents the first translator which takes symbolic descriptions of how shapes are drawn, displayed, and edited in a domain and automatically transforms them into shape recognizers, editing recognizers, and shape exhibitors for use in a domain independent sketch recognition system. To accomplish this, we created 1) LADDER, a symbolic language for describing how shapes are drawn, displayed, and edited in a domain, 2) the translator described above, and 3) a simple domain independent recognition system that uses the newly translated components to recognize, display, and allow editing of the domain shapes. The implementation of this translator and domain independent sketch recognition system serves to show both that such a framework is feasible and that LADDER is an acceptable language for describing domain information.

[7] 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.

References

Alvarado, C. 2000. A natural sketching environmant: Bringing the computer into early stages of mechanical design. Master's thesis, MIT.

Alvarado, C. 2003. Dynamically constructed bayesian networks for sketch understanding. *Proceedings of the 3nd Annual MIT Student Oxygen Workshop*.

Bimber, O.; Encarnacao, L. M.; and Stork, A. 2000. A multilayered architecture for sketch-based interaction within virtual environments. *Computer and Graphics: Special Issue on Calligraphic Interfaces: Towards a New Generation of Interactive Systems* 24(6):851–867.

Caetano, A.; Goulart, N.; Fonseca, M.; and Jorge, J. 2002a. JavaSketchIt: Issues in sketching the look of user interfaces. *Sketch Understanding, Papers from the 2002 AAAI Spring Symposium*.

Caetano, A.; Goulart, N.; Fonseca, M.; and Jorge, J. 2002b. Sketching user interfaces with visual patterns. *Proceedings of the 1st Ibero-American Symposium in Computer Graphics (SIACG02)* 271–279.

Costagliola, G.; Tortora, G.; Orefice, S.; and Lucia, D. 1995. Automatic generation of visual programming environments. In *IEEE Computer*, 56–65.

Do, E. Y.-L. 2001. Vr sketchpad - create instant 3d worlds by sketching on a transparent window. *CAAD Futures 2001, Bauke de Vries, Jos P. van Leeuwen, Henri H. Achten (eds)* 161–172.

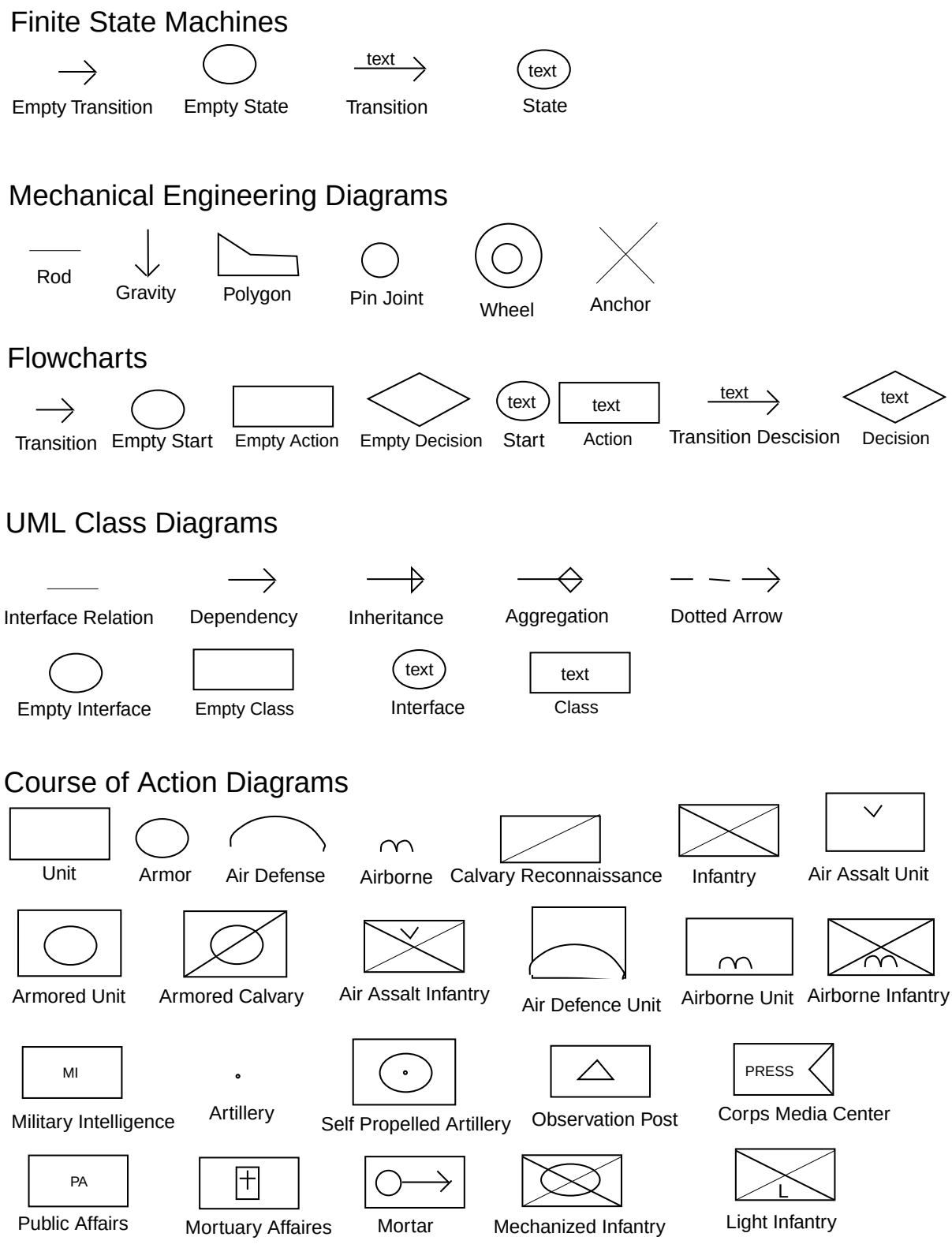

Figure 3: Domains and their shapes for which we have automatically generated sketch recognition systems.

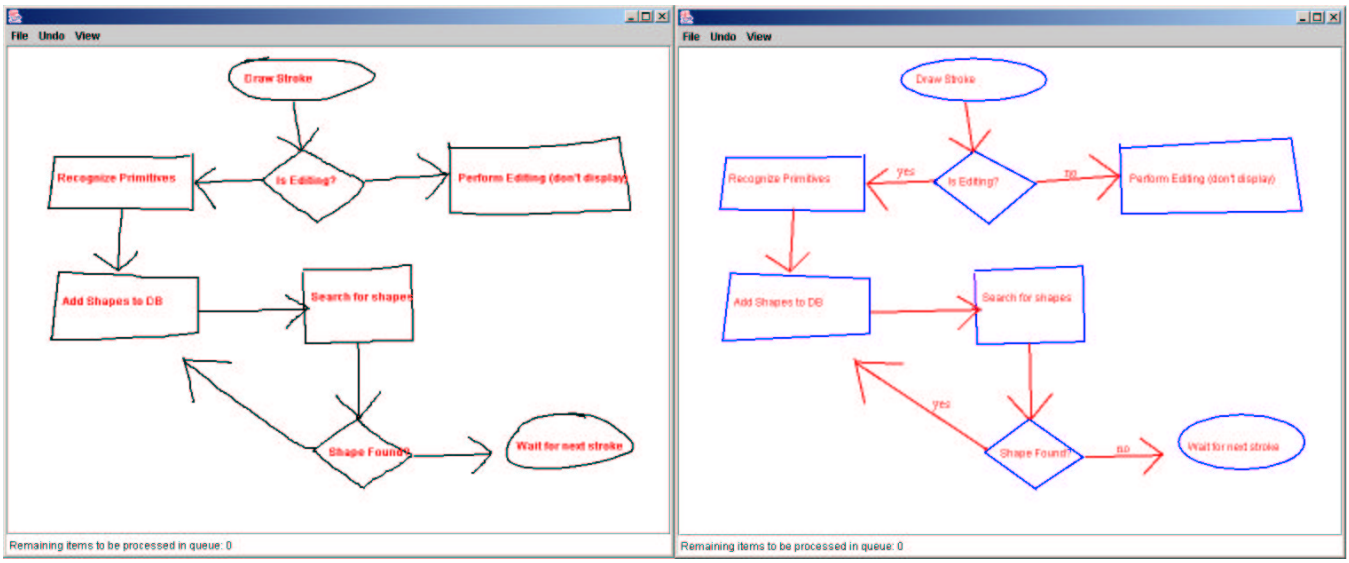

Figure 4: Flowchart of domain independent system drawn using system. Left side represents the original strokes. Right side represents the cleaned up drawing.

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

Futrelle, R. P., and Nikolakis, N. 1995. Efficient analysis of complex diagrams using constraint-based parsing. In *ICDAR-95 (International Conference on Document Analysis and Recognition)*, 782–790.

Gross, M. D., and Do, E. Y.-L. 1996. Demonstrating the electronic cocktail napkin: a paper-like interface for early design. *'Common Ground' appeared in ACM Conference on Human Factors in Computing (CHI)* 5–6.

Gross, M.; Zimring, C.; and Do, E. 1994. Using diagrams to access a case library of architectural designs. In Gero, J., and Sudweeks, F., eds., *Artificial Intelligence in Design '94*. Netherlands: Kluwer Academic Publishers. 129–144.

Hammond, T., and Davis, R. 2002. Tahuti: A geometrical sketch recognition system for uml class diagrams. *AAAI Spring Symposium on Sketch Understanding* 59–68.

Hammond, T., and Davis, R. 2003. Ladder: A language to describe drawing, display, and editing in sketch recognition. *Proceedings of the 2003 Internaltional Joint Conference on Artificial Intelligence (IJCAI)*.

Hse, H.; Shilman, M.; Newton, A. R.; and Landay, J. 1999. Sketch-based user interfaces for collaborative object-oriented modeling. Berkley CS260 Class Project.

Jacob, R. J. K.; Deligiannidis, L.; and Morrison, S. 1999. A software model and specification language for non-WIMP= user interfaces. *ACM Transactions on Computer-Human Interaction* 6(1):1–46.

Lecolinet, E. 1998. Designing guis by sketch drawing and visual programming. In *Proceedings of the International Conference on Advanced Visual Interfaces (AVI 1998).*, 274–276. AVI.

Lin, J.; Newman, M. W.; Hong, J. I.; and Landay, J. 2000. Denim: Finding a tighter fit between tools and practice for web site design. In *CHI Letters: Human Factors in Computing Systems, CHI 2000*, 510–517.

Long, A. C. 2001. *Quill: a Gesture Design Tool for Pen-based User Interfaces*. Eecs department, computer science division, U.C. Berkeley, Berkeley, California.

Mahoney, J. V., and Fromherz, M. P. J. 2002. Handling ambiguity in constraint-based recognition of stick figure sketches. *SPIE Document Recognition and Retrieval IX Conf., San Jose, CA*.

Mahoney, J. V., and Frommerz, M. P. J. 2002. Three main concerns in sketch recognition and an approach to addressing them. In *Sketch Understanding, Papers from the 2002 AAAI Spring Symposium*, 105–112. Stanford, California: AAAI Press.

Pittman, J.; Smith, I.; Cohen, P.; Oviatt, S.; and Yang, T. 1996. Quickset: A multimodal interface for military simulations. *Proceedings of the 6th Conference on Computer-Generated Forces and Behavioral Representation* 217–224.

Rubine, D. 1991. Specifying gestures by example. In *Computer Graphics*, volume 25(4), 329–337.

Sezgin, T. M. 2001. Feature point detection and curve approximation for early processing in sketch recognition. Master's thesis, Massachusetts Institute of Technology.

Sezgin, T. M. 2003. Recognition efficiency issues for freehand sketches. *Proceedings of the 3nd Annual MIT Student Oxygen Workshop*.

Stiny, G., and Gips, J. 1972. Shape grammars and the generative specification of painting and sculpture. In Freiman, C. V., ed., *Information Processing 71*. North-Holland. 1460–1465.

Veselova, O. 2003. Perceptually based learning of shape descriptions. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA.

Zue, and Glass. 2000. Conversational interfaces: Advances and challenges. *Proc IEEE* 1166–1180.