LADDER: A Perceptually-based Language to Simplify Sketch Recognition User Interface Development

by

Tracy Anne Hammond

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

January 2007

© Massachusetts Institute of Technology 2007. All rights reserved.

Author . . . .
Department of Electrical Engineering and Computer Science
January 2007

Certified by . . . .
Randall Davis
Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Students

LADDER: A Perceptually-based Language to Simplify Sketch Recognition User Interface Development

by

Tracy Anne Hammond

Submitted to the Department of Electrical Engineering and Computer Science
on January 2007, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

Diagrammatic sketching is a natural modality of human-computer interaction that can be used for a variety of tasks, for example, conceptual design. Sketch recognition systems are currently being developed for many domains. However, they require signal-processing expertise if they are to handle the intricacies of each domain, and they are time-consuming to build.

Our goal is to enable user interface designers and domain experts who may not have expertise in sketch recognition to be able to build these sketch systems. We created and implemented a new framework (*FLUID* - **f**aci**li**tating **u**ser **i**nterface **d**evelopment) in which developers can specify a domain description indicating how domain shapes are to be recognized, displayed, and edited. This description is then automatically transformed into a sketch recognition user interface for that domain. *LADDER*, a language using a perceptual vocabulary based on Gestalt principles, was developed to describe how to recognize, display, and edit domain shapes. A translator and a customizable recognition system (*GUILD* - a **g**enerator of **u**ser **i**nterfaces using **l**adder **d**escriptions) are combined with a domain description to automatically create a domain specific recognition system. With this new technology, by writing a domain description, developers are able to create a new sketch interface for a domain, greatly reducing the time and expertise for the task

Continuing in pursuit of our goal to facilitate UI development, we noted that 1) human generated descriptions contained syntactic and conceptual errors, and that 2) it is more natural for a user to specify a shape by drawing it than by editing text. However, computer generated descriptions from a single drawn example are also flawed, as one cannot express all allowable variations in a single example.

In response, we created a modification of the traditional model of active learning in which the system selectively generates its own near-miss examples and uses the human teacher as a source of labels. System generated near-misses offer a number of advantages. Human generated examples are tedious to create and may not expose problems in the current concept. It seems most effective for the near-miss examples to be generated by whichever learning participant (teacher or student) knows better where the deficiencies lie; this will allow the concepts to be more quickly and effectively refined. When working in a closed domain such as this one, the computer learner knows exactly which conceptual uncertainties remain, and which hypotheses need to be tested and confirmed. The system uses these labeled examples to automatically build a *LADDER* shape description, using a modification of the version spaces algorithm that handles interrelated constraints, and which also has the ability to learn negative and disjunctive constraints.

Thesis Supervisor: Randall Davis
Title: Professor of Computer Science and Engineering

Acknowledgments

"And what has my dear rabbit been thinking,
safe and protected in her lair,
while her owl was out in the world
suffering the slings and arrows et cetera."

"Father I've climbed the tree beyond its leaves."

- Alice in Bed, by Susan Sontag

I welcome this opportunity to thank the many people who contributed to the successful completion of this project.

First, I wish to thank the members of my dissertation committee. My academic advisor, Dr. Randall Davis, proved me with the necessary guidance from the onset of this topic. Through his gentle concern and challenging line of philosophical thinking, he helped me to realize simplicity from the seemingly impossible. I want to thank Dr. Rob Miller and Dr. Stephanie Seneff for their willingness to take on the sponsorship of this dissertation. Through their rigor and dedication to excellence, they have helped shape my study, leaving me with a final project of which I am proud. I am grateful to Dr. Patrick Winston for his guidance on learning with near-misses.

I would like to thank my academic advisors, Dr. Leslie Pack Kaelbling and Dr. Steve Ward for their advice and encouragement through the process. Marilyn Pierce has been helpful throughout my time here, but she has been a saint through the submission process.

The members of the design rationale group (including, among others, Randall Davis, Michael Oltmans, Aaron Adler, Christine Alvarado, Jacob Eisenstein, Mark Foltz, Metin Sezgin, Olya Veselova, Sonya Cates, Tom Ouyang, Rebecca Hitchcock, Dana Scott, Zardosht Kasheff, Kalani Oshiro, and Oskar Bruening) have provided me with a sounding board to discuss many of these ideas, and I thank them for that.

I want to thank my co-authors who have worked with me to further my research, including Randall Davis, Howard Shrobe, Krzystof Gajos, Christine Alvarado, Michael Oltmans, Metin Sezgin, Olya Veselova, Aaron Adler, Rebecca Hitchcock, Jan Hammond, Dana Scott, Zardosht Kasheff, Kalani Oshiro, and Raghavan Parthasarthy.

I also appreciate the many people who have reviewed this document and given me helpful feedback, including Randall Davis, Jan Hammond, Raghavan Parthasarthy, Aaron Adler, Rob Miller, Stephanie Seneff, and Jacob Eisenstein, who gets credit for the creation of the name *LADDER*.

On a more personal note, there were many people who befriended me and supported me during this time. I would like to thank my close friends at MIT for late night chats and for keeping me smiling throughout the process, including David Craft, Michael Oltmans, Christine Alvarado, Aaron Adler, Jacob Eisenstein, Metin Sezgin, Louis-Philippe Morency, Mike McGeachie, Olya Veselova, Michal Karczmarek, and Saul Griffith.

David Craft, Michael Oltmans, Aaron Adler, and Johan Anderson, I would like to make special thanks to both of you; I owe you more than I can ever repay. You were always there for me, above and beyond what was ever expected. I will appreciate you always.

Throughout the years I have had many housemates (including David Craft, Shannon Senn, Jody Fanto, Tibaut Lamadon, Lars Vojt, Brooke Cowan, Matt Marjanovic, Victor, and Louis-Philippe Morency) who have lived with me during the my growth as a doctoral student and a person. I thank them for their support, flexibility, and understanding. I would like to give a special thank you to David, Shannon, and Louis-Philippe who provided me especially meaningful friendships in my life.

Thank you, Chris Zebo, for dancing our way to found the MIT salsa club. I couldn't have done it without you. Chris, also thank you for your friendship and guidance through this changing time in my life. (And thank to salsa for keeping me sane. :-) Swing dancing has become a new adventure for me in my life, and I am happy to have Adam Siegel and Robert Bonazoli as new friends. Thank you, Ion Freeman, for being Apollo to my Mercury in the Gilbert and Sullivan play Thespis.

I would like to thank the other close friends in my life who have stayed with me for many years, and continue to stay with me as I move into a new chapter of my life. Julie Insignares, Claire Laporte, Beth Zick, and Jenny Drake, you are incredibly important to me, and I feel blessed every day to know that I have you in my world.

My family has been with me always, close to my heart, and always supportive. I treasure my aunts (Rhonda, Barbara, Sherry, Nanci, and Maggie), my uncles (Scott, David, Tom, and Rob), my cousins (Shannon, Tommy, Jayne, Michael, Scotia, Travis, Kimberly, and Jenny), my brother Madhavan, my sister Jen, my nephew Connor, my niece Rachel and my Indian family.

I thank my dad, ‘Captain Jack,’ for always reminding me what my priorities are. He retired at age 49 to live his life long dream, living on a boat, sailing and traveling around the world. I am very proud of him for living his life the way he always wanted.

I thank my dad, ‘Professor Raghavan,’ for his infinite wisdom and advice throughout this process. I also would like to thank his countless patience, love, and support during times of frustration.

And most of all, I thank Jan Hammond, my mother, my editor, my teacher, my consultant, my therapist, my cook, my cleaner, my driver, my all. I am so lucky to have her as a mother.

I dedicate this document to her. To her, and to the two sleeping Newfoundlands at my feet, without I have no idea how I survived so long in this world.

dog in the street

stretching

in a nice long yawn –

pure envy

I do likewise

- Takuboku: Poems to Eat

There is something in animals which soothes him,

at least in the ones that leave him speechless with awe.

- Elias Canetti: The Agony of the Flies

And, at this point, I begin a new chapter of my life. I know there is much more to learn, and I eagerly await the new challenges, lessons, and hopes that precede me.

Now the wonderful world is born.
In an instant it dies.
In a breath it is renewed.

From the slowness of our eye
And the quickness of God's hand
We believe in the world.

- The Mahabharata

Contents

List of Figures

List of Tables

Chapter 1

Overview

1.1 Motivation

As computers become an integral part of our lives, it becomes increasingly important to make working with them easier and more natural. Our vision is to make human-computer interaction as easy and as natural as human-human interaction. As part of this vision, it is imperative that computers understand forms of human-human interaction, such as sketching. Computers should be able to understand the information encoded in diagrams drawn by and for scientists and engineers. A mechanical engineer, for example, can use a hand-sketched diagram to depict his design to another engineer. Sketching is a natural modality of human-computer interaction for a variety of tasks, including for example, conceptual design [60] [135] [134].

Paper sketches offer users the freedom to draw as they would naturally; for instance, users can draw objects with any number of strokes, and strokes may be drawn in any order. However, because paper sketches are static and uninterpreted, they lack computer editing features, requiring users to completely erase and redraw objects in order to move them. In an attempt to combine the freedom provided by a paper sketch with the powerful editing and processing capabilities of an interpreted diagram, sketch recognition systems have been developed for many domains, including Java GUI creation [41], UML class diagrams [99] [143], and mechanical engineering [6]. Sketch interfaces 1) interact more naturally than traditional mouse-and-palette tools by allowing users to hand-sketch diagrams, 2) can connect to a back-end system (such as a CAD tool) to offer real-time design advice, 3) recognize the shape as a whole to allow for more powerful editing, 4) beautify diagrams, removing mess and clutter, and thereby 5) notify the sketcher that the shapes have been recognized correctly.

Sketch recognition interfaces provide a number of benefits, as described above, but they can be quite time-consuming to build and require signal-processing expertise, if they are to handle the intricacies of each domain. We want to enable domain user interface designers, who need not be experts in sketch recognition at the signal level, to be able to build a sketch recognition system for each of their domains. This thesis describes a set of ideas and techniques to enable developers to build sketch recognition systems.

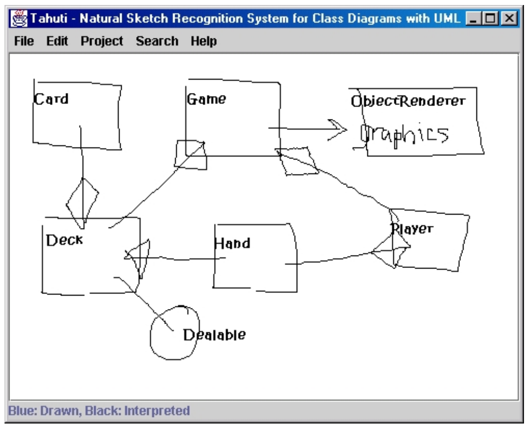
1.2 Natural Sketch Recognition

Previous sketch systems required users to learn a particular stylized way of drawing, and used a feature-based recognition algorithm, such as a Rubine [185] or a Graffiti^{TM}-type [192] algorithm. What these algorithms lose in natural interaction by requiring the sketcher to draw in a particular style, they gain in speed and accuracy. Rather than recognizing shapes, the algorithm recognizes sketched gestures, where each gesture represents a single shape. These sketched gestures focus more on how something was drawn than on how the drawn object looks. These recognition algorithms require that each gesture be drawn in a single stroke in the same manner (i.e., same underlying stylistic features–stroke direction, speed, etc.) each time. Each gesture is recognized based on a number of features of that stroke, such as the initial angle of the stroke, end angle, speed, number of crosses, etc. Because of these requirements, the gesture representing the shape may look different from the shape itself. For example, it would be impossible to draw a cross with a single stroke, so it may, instead, be represented by a ribbon-like gesture. Also, even if a sketcher draws a shape that looks the same as the required gesture, it may not be recognized because it does not have the same underlying stylistic features.

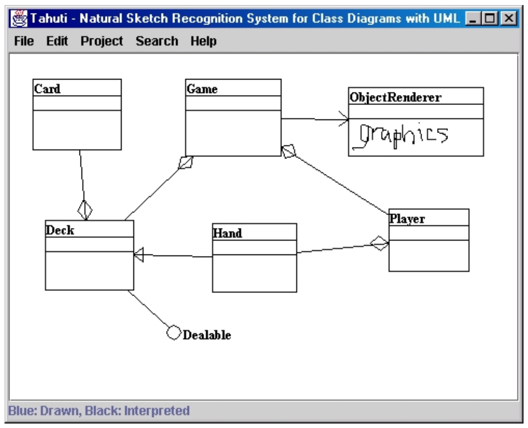
Our goal is to build sketch recognition systems that allow sketchers to draw as they would naturally–that is, without having to learn a new set of stylized symbols. As long as the shape looks like the final shape, the shape should be recognized, independent of the number, direction, or order of the strokes drawn.

To allow for natural drawing in our sketch recognition systems, shapes are described and recognized in terms of the subshapes that make up the shape and the geometric relationships (constraints) between the subshapes. Strokes are first broken down into a collection of primitive shapes, including lines, ellipses, arcs, spirals, points, and curves, using techniques from Sezgin [194]. A higher-level shape is then recognized by searching for possible subshapes and testing that the appropriate geometric constraints hold. The geometric constraints confirm topology, orientation, angles, relative size, and relative location.

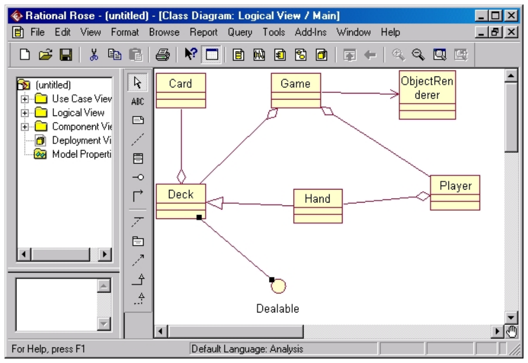
To demonstrate that recognition could be performed using a shape-based model, we built *Tahuti* [99], a system for recognizing UML class diagrams [37] using geometric constraints. *Tahuti* recognizes a general class (represented by a rectangle), an interface class (represented by an ellipse), an inheritance association (represented by an arrow with a triangle-shaped head), an aggregation association (represented by an arrow with a diamond-shaped head), a dependency association (represented by an arrow or line), and an interface association (represented by a line that connects a class to an interface). It also accept keyboard text as part of these objects. The recognized sketches are sent to Rational Rose^{TM}, a CASE (computer-automated software engineering) tool that generates stub code. Figure 1-1 shows the hand-drawn UML class diagram in Tahuti (Figure 1-1(a)), the recognized shapes in Tahuti (Fig-


Tahuti - Natural Sketch Recognition System for Class Diagrams with UML
File Edit Project Search Help
Card
Game
ObjectRenderer
Graphics
Deck
Hand
Player
Dealable
Blue: Drawn, Black: Interpreted

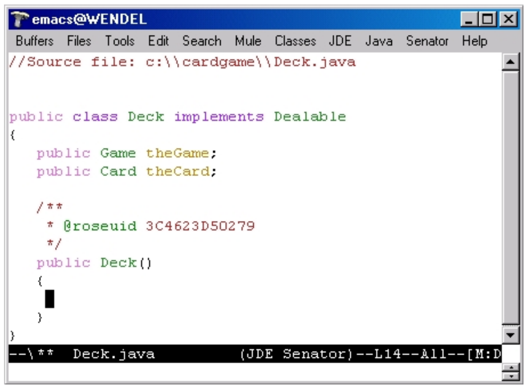
(a) Hand-drawn UML class diagram in Tahuti


Tahuti - Natural Sketch Recognition System for Class Diagrams with UML
File Edit Project Search Help
Card
Game
ObjectRenderer
Graphics
Deck
Hand
Player
Dealable
Blue: Drawn, Black: Interpreted

(b) Recognized UML class diagram in Tahuti


Rational Rose - [untitled] - [Class Diagram: Logical View / Main]
File Edit View Format Browse Report Query Tools Add-Ins Window Help
(untitled)
Use Case View
Logical View
Component View
Deployment View
Model Properties
Card
Game
ObjectRenderer
Deck
Hand
Player
Dealable
For Help, press F1
Default Language: Analysis

(c) Diagram sent to Rational Rose^{TM}


emacs@WENDEL
Buffers Files Tools Edit Search Mule Classes JDE Java Senator Help

```
//Source file: c:\\cardgame\\Deck.java


public class Deck implements Dealable
{
   public Game theGame;
   public Card theCard;

   /**
    * @roseuid 3C4623D50279
    */
   public Deck()
   {

   }
}
--\**  Deck.java          (JDE Senator)--L14--All--[M:D
```

(d) Stub code generated by Rational Rose^{TM}

Figure 1-1: A hand-drawn UML class diagram in Tahuti, the recognized shapes in Tahuti, the recognized shapes sent to Rational Rose^{TM}, and the generated stub code produced by Rational Rose^{TM}.

ure 1-1(b)), the recognized shapes sent to Rational Rose^{TM}(Figure 1-1(c)), and the generated stub code produced by Rational Rose^{TM}(Figure 1-1(d)). Class names come from typed input.

Sketch recognition systems often prove useful in many unexpected ways. Tahuti was used to automatically index video documentation of software design meetings [105]. Collaborative software design meetings often involve the creation of UML software diagrams. What usually gets saved and recorded after those meetings are the final designs and the explanations of the mechanisms. What gets omitted, however,

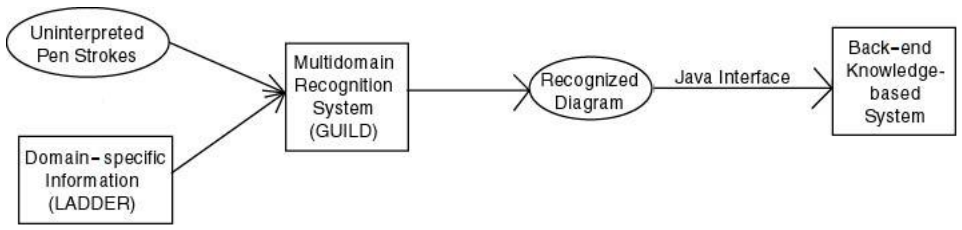
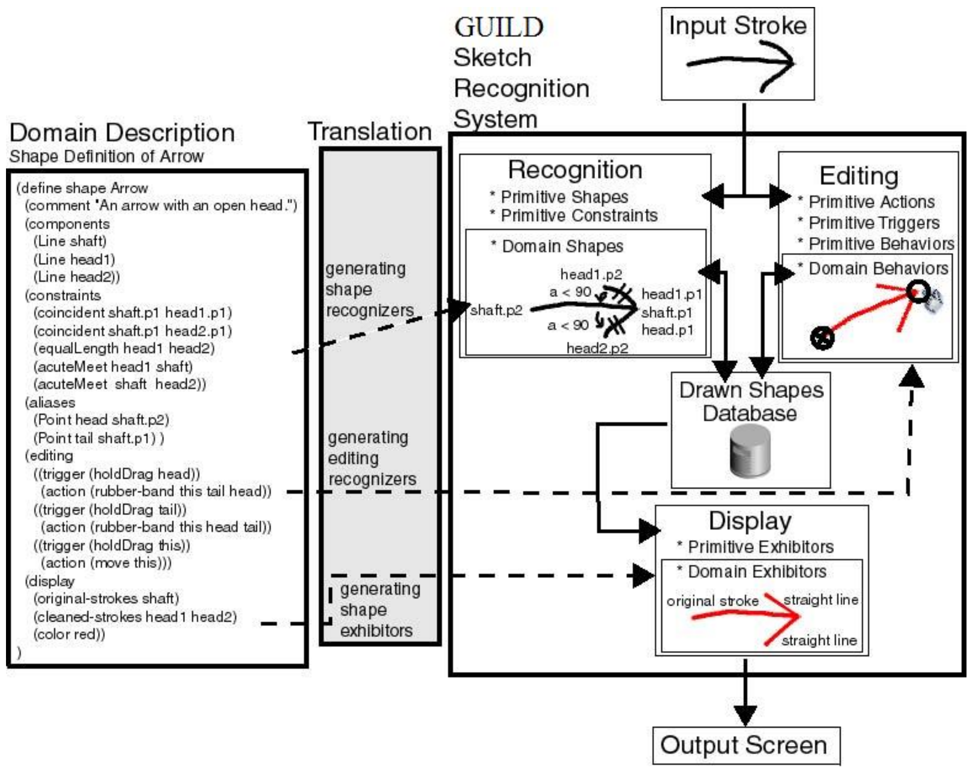

Uninterpreted Pen Strokes
Domain-specific Information (LADDER)
Multidomain Recognition System (GUILD)
Recognized Diagram
Java Interface
Back-end Knowledge-based System

Figure 1-2: The *FLUID* framework.

is the design rationale, or the reasons why those particular solutions were employed. Software meetings can be videotaped so as to unobtrusively capture any design rationales provided at the meeting. However, the videotaped information is in a cumbersome un-indexed format. If a developer wanted to determine the nature of the discussion in the room while a particular class was being developed in order to understand the design rationale behind its creation, he or she might have to watch the entire video. To help to mitigate this problem, Tahuti time-stamps the drawing and editing events, indexing the video, allowing easy access to the portion of the videotaped software meeting during which a class was created.

1.3 The *FLUID* Framework: *F*aci*l*itating *UI* *D*evelopment

Sketch recognition systems are useful, but they take a long time to build, and system designers have to be experts in sketch recognition at the signal level. In order to build a good sketch recognition system, a designer also needs to be an expert in the domain itself. Rarely is the domain expert also an expert in sketch recognition. This research aims not only to decrease the time necessary to build a new sketch recognition system, but also to reduce the effort and the amount of signal processing knowledge that is necessary in order to create a new system. We do this by abstracting away the signal processing, pattern recognition, and algorithmic details of sketch recognition.



```
(define shape Arrow
  (comment "An arrow with an open head.")
  (components
    (Line shaft)
    (Line head1)
    (Line head2))
  (constraints
    (coincident shaft.p1 head1.p1)
    (coincident shaft.p1 head2.p1)
    (equalLength head1 head2)
    (acuteMeet head1 shaft)
    (acuteMeet  shaft  head2))
  (aliases
    (Point head shaft.p2)
    (Point tail shaft.p1) )
  (editing
    ((trigger (holdDrag head))
      (action (rubber-band this tail head))
    ((trigger (holdDrag tail))
      (action (rubber-band this head tail))
    ((trigger (holdDrag this))
      (action (move this)))
  (display
    (original-strokes shaft)
    (cleaned-strokes head1 head2)
    (color red))
)
```

Figure 1-3: The domain description is translated into recognizers, exhibitors, and editors for each shape in the domain.

1.3.1 Framework

Rather than build a separate recognition system for each domain, it should be possible to build a single, domain-independent recognition system that can be customized for each domain. In this approach, building a sketch recognition system for a new domain requires only writing a domain description, which describes how shapes are drawn, displayed, and edited. This description is then transformed for use in the domain-independent recognition system. The inspiration for such a framework stems from work in speech recognition and compiler compilers, which have used this approach with some success [217] [54] [211] [5].

The *FLUID* framework (**f**aci**l**itating **u**ser **i**nterface **d**evelopment) for automatic sketch building is shown in Figure 1-2. To build a sketch interface for a new domain, a developer writes 1) a *LADDER* domain description describing how each shape is drawn, displayed, and edited in the domain and 2) a Java interface to an existing back-end system (such as a CAD tool). The *GUILD* recognition system translates the domain description into shape recognizers, editors, and exhibitors, as shown in Figure 1-3, and functions as a recognition system for that domain, connecting to the back-end system.

1.4 *LADDER*, a Perceptual Language for Describing Shapes

In order to generate a sketch interface for a particular domain, the system needs domain-specific information, indicating what shapes are in the domain and how each shape in the domain is to be recognized, displayed, and edited. Domain information should provide a high level of abstraction to reduce the effort and the amount of sketch recognition knowledge that is needed by the developer. The domain information should be accessible, understandable, intuitive, and easy for the developer to specify. We argue that if domain information is more understandable, the developer is less likely to introduce errors caused by confusion.

A shape description needs to be able to describe a generalized instance of the shape, describing all acceptable variations, so that the recognition system can properly recognize all allowable variations. A shape description should not include stylized mannerisms (such as the number, order, direction, or speed of the strokes used) that would not be present in other sketchers' drawings of a shape, as it would require all sketchers to draw in the same stylistic manner as the developer in order for their sketches to be recognized. Thus, we have chosen to describe shapes according to their user-independent visual properties.

We developed *LADDER*, a perceptual language for describing shapes, for use by developers to specify the necessary domain information. The language consists of predefined primitive shapes, constraints, editing behaviors, and display methods, as well as a syntax for combining primitives to create more complex shapes in a domain specification. Shape descriptions primarily concern shape, but may include information such as stroke order or stroke direction, if that information would prove useful to the recognition process. The specification of editing behavior allows the system to determine when a pen gesture is intended to indicate editing rather than a stroke, and what to do in response. Display information indicates what to display after strokes are recognized.

The difficulty in creating such a language involves ensuring that the language is broad enough to support a wide range of domains, yet narrow enough to remain comprehensible and intuitive in terms of vocabulary. To achieve sufficient broadness, *LADDER* was used to describe several hundred shapes in a variety of domains. Figure 1-4 shows a sample of the shapes described. To achieve sufficient narrowness, we chose to include only perceptually-important constraints. We argue that a language with fewer constraints will be more comprehensible, as the developer can more easily remember which constraints are available in the domain.

To ensure that the language is intuitive, we examined literature to determine what shape constraints are perceptually-important. Literature on Gestalt principles proved valuable, describing which visual constraints are most perceptually important to people [210]. We confirmed that people are better at identifying horizontal or vertical angles than diagonal angles. We also confirmed that these principles agreed with how people naturally describe shapes by asking thirty-five users to describe approximately thirty shapes each, both verbally and textually.

LADDER takes advantage of these human perceptual differences to make the language more intuitive by including constraints that model the Gestalt principles. By modeling the language after perceptually-important constraints, we can simplify it by omitting constraints that describe differences that are perceptually unimportant. This ensures that the number of constraints is kept small, making the language narrow enough to remain comprehensible and, thus, easier to find the appropriate constraints so as to describe the shapes in question.

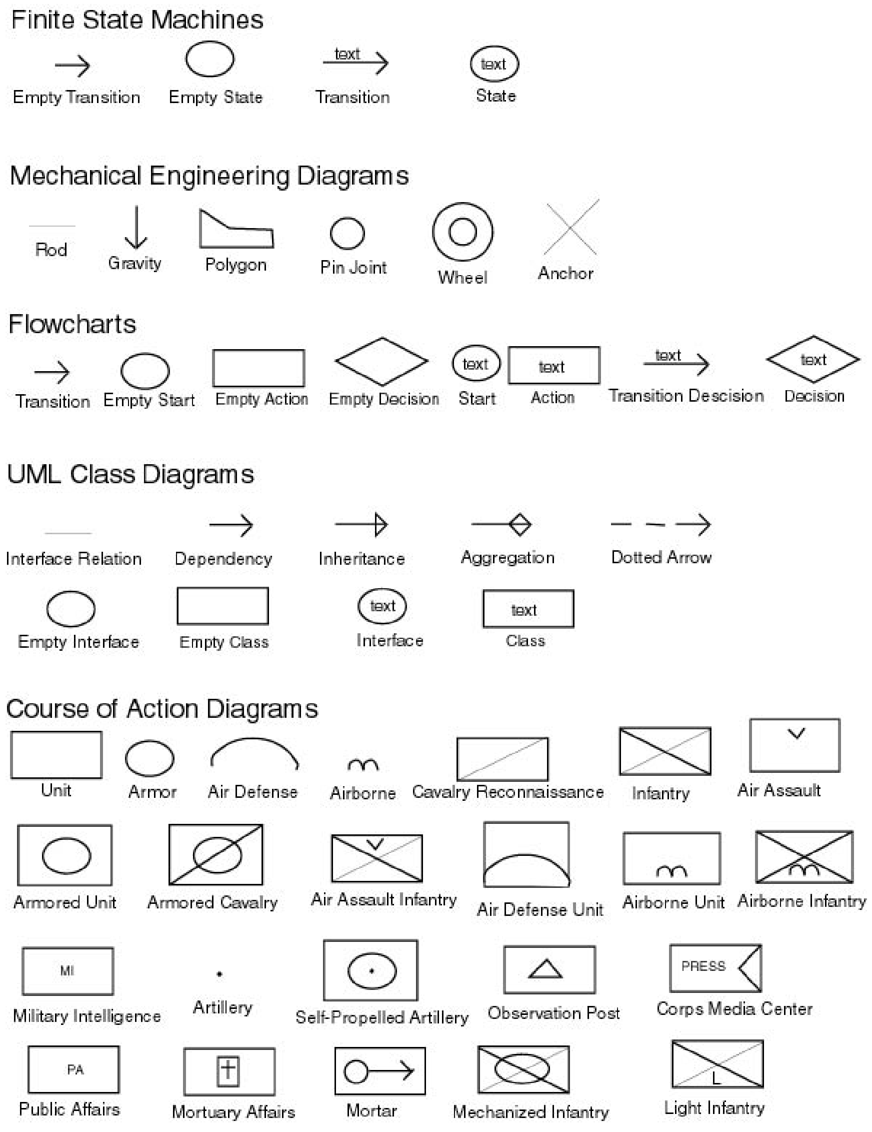


Figure 1-4: A wide variety of shapes have been described using the language.

The language also has a number of higher-level features that simplify the task of creating a domain description. Shapes can be built hierarchically. Shapes can extend abstract shapes, which describe shared shape properties, making it unnecessary for the application designer to define these properties numerous times. As an example, several shapes may share the same editing properties. Shapes with a variable number of components, such as polylines or polygons (which have a variable number of lines), can be described by specifying the minimum and maximum number of components (e.g., lines) allowed. Contextual information from neighboring shapes also can be used to improve recognition by defining shape groups; for instance, contextual information can distinguish a pin joint from a circular body in mechanical engineering. Shape group information also can be used to perform chain reaction editing, such as having the movement of one shape cause the movement of another.

1.5 Recognition System

This research also includes the creation of the *GUILD* (Generator of User Interfaces from a *LADDER* Description) system that automatically creates a sketch recognition system from a *LADDER* domain description. The internal customizable recognition system of *GUILD* takes the translated recognizers, exhibitors, and editors, and acts as a sketch recognition system for the described domain.

Because allowing sketchers to draw as they would naturally is important in creating a usable sketch recognition system, we built the customizable sketch recognition system to do recognition based on geometric shape properties. Each shape description specifies a number of constraints that must be true for that shape, and the system performs recognition based on those constraints.

Recognition consists of two stages: stroke processing and shape recognition. During stroke processing, each stroke is broken down into a collection of primitive shapes, including line, arc, circle, ellipse, curve, point, and spiral. During shape recognition, more complicated shapes are recognized by the identifying subshapes that make up these higher-level shapes, and then by confirming the relationships (constraints) between the subshapes. If a stroke or shape has multiple interpretations, all interpretations are added to the pool of recognized shapes, but a single interpretation is chosen for display. The system chooses to display the interpretation that is composed of the largest number of primitive shapes or the first found interpretation, in the case of interpretations composed of the same number of primitive shapes.

This research also includes a new fast recognition algorithm based on indexing.

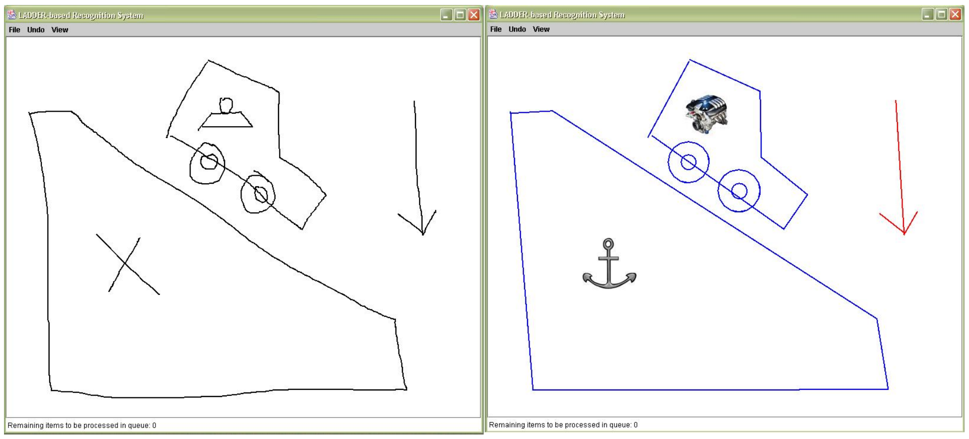

LADDER-based Recognition System
File Undo View
Remaining items to be processed in queue: 0

Figure 1-5: A screenshot depicting the drawn and interpreted version of a mechanical engineering diagram being drawn. This picture depicts a car on a hill. The motor, just having been drawn, is about to be dragged so that it is placed over the front wheel, to depict that the force should be applied to the front wheel ("front wheel drive").

This algorithm takes advantage of the perceptually-based constraints in *LADDER* to allow shapes to be drawn in an interspersed manner, but still be recognized in real-time.

1.5.1 Connecting to Existing Systems

To allow developers of a sketch system to connect to an existing knowledge system, such as a CAD or CASE tool, a connection API was created. With this API, the domain-specific recognition system can connect to a back-end system, providing additional functionality, e.g., checking the diagram for inconsistencies, running the diagram to see whether it works as the sketcher intended, etc. So far, we have created systems that connect to Rational Rose$^{\mathrm{TM}}$(for UML class diagrams as in Figure 1-1(c)), Working Model (mechanical engineering simulations), Spice (for electrical circuit analysis), as well as our own systems (such as when video-taping and indexing software meetings).

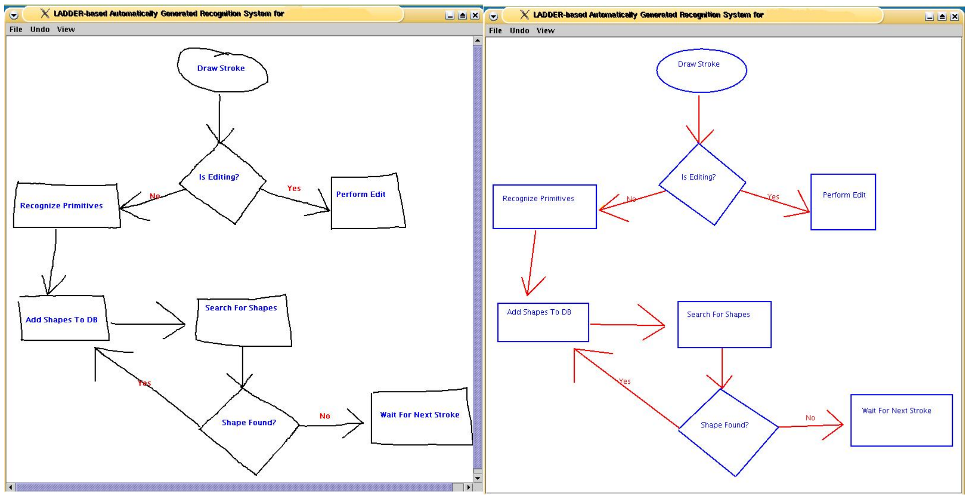


Figure 1-6: A drawn and interpreted flow chart diagram.

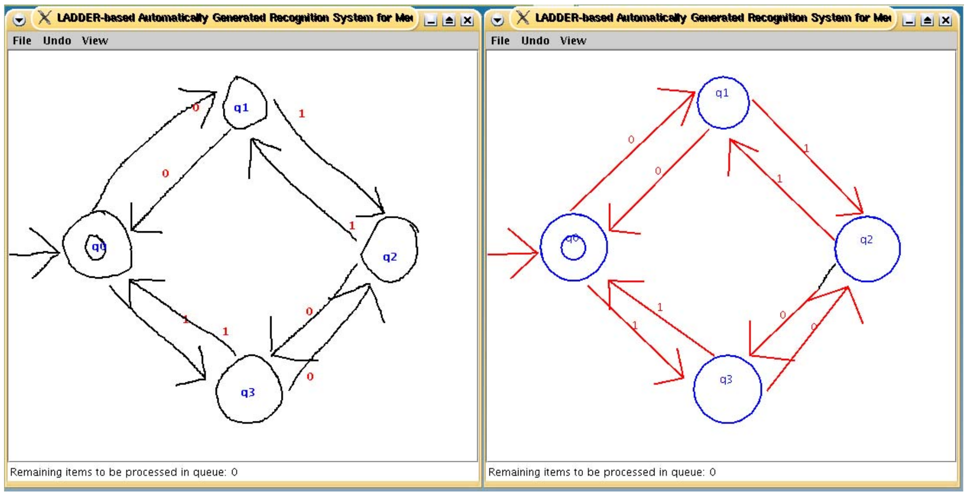

LADDER-based Automatically Generated Recognition System for Me
File Undo View
q0
q1
q2
q3
0
1
Remaining items to be processed in queue: 0

Figure 1-7: The drawn and interpreted diagram of a finite state machine.

1.6 Results: Automatically-Generated Sketch Systems

Several domains have been described using *LADDER*, and recognition systems have been generated for them. Figure 1-5 shows a drawn and interpreted mechanical engineering diagram. Figure 1-6 shows a drawn and interpreted flow chart diagram. Figure 1-7 shows the drawn and interpreted diagrams of a finite state machine.

1.7 Generating Shapes

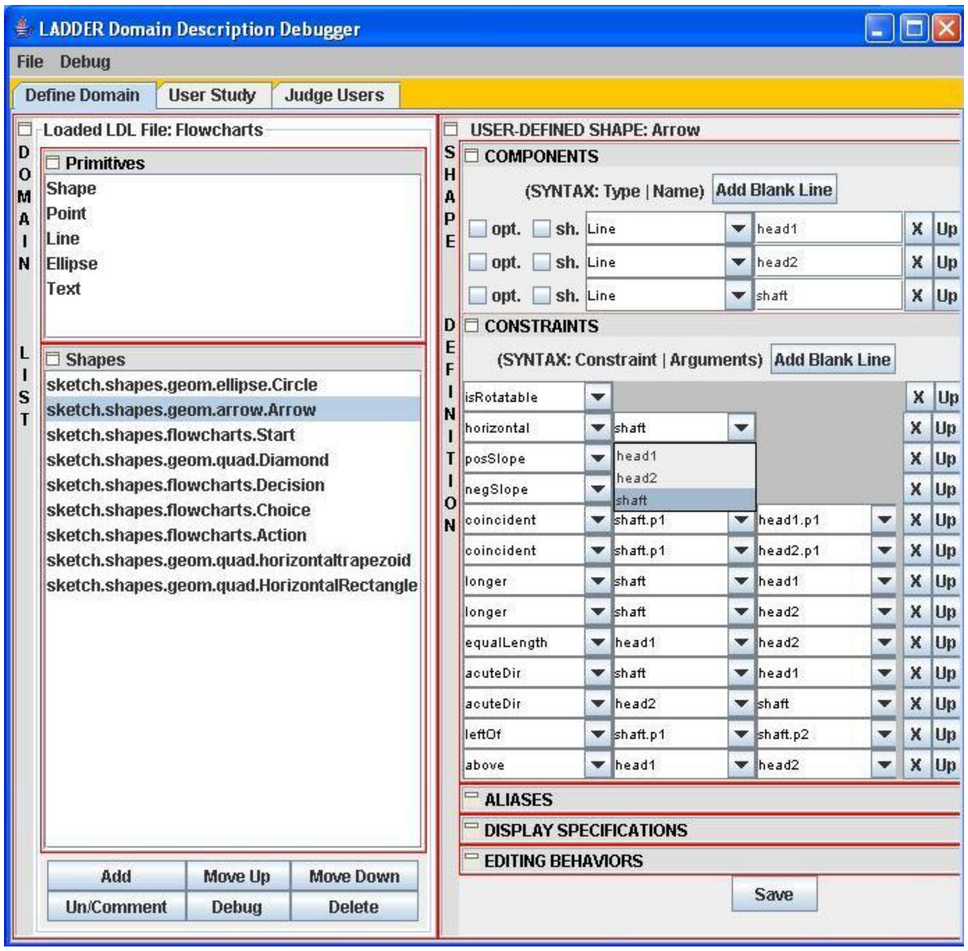
Given a *LADDER* description, the system can generate a shape that agrees with that description. The system uses MATLAB to solve all of the constraints in the description. This is helpful when users want to beautify their shapes and display the ideal versions of their shapes in the recognition system. The system also uses this technique to automatically generate near-miss example shapes.

1.8 System Generated Description from Single Hand-drawn Example

Modeling *LADDER* after Gestalt principles has made the language more intuitive. However, human-generated descriptions contain conceptual and syntactical errors. To help the developer fix syntax errors, we built a GUI to constrain input and to notify the user of syntactical faults. Figure 1-8 shows a screen shot.

While this GUI prevents syntactic errors, conceptual errors, such as a missing constraint, still remain. Developers need to be very logical in order to create correct shape descriptions. In addition, developers may find it much more natural to draw shapes than to describe them textually.

To solve these problems, this research includes the developement a system to automatically generate a best-guess shape description from a single drawn example, based on ideas from Veselova [210]. The system takes a single hand-drawn positive example of the shape and creates a list of all *LADDER* constraints true for that example shape. We know that the correct description of the shape is a generalization of this list. The difficulty is choosing the appropriate generalization of this list. If we were to keep all of these constraints in our best-guess shape description, we would recognize only our very specific example of the shape, producing false negatives. If we make our description too general, it will allow too many variations, creating false



Loaded LDL File: Flowcharts

Primitives

- Shape
- Point
- Line
- Ellipse
- Text

Shapes

- sketch.shapes.geom.ellipse.Circle
- sketch.shapes.geom.arrow.Arrow
- sketch.shapes.flowcharts.Start
- sketch.shapes.geom.quad.Diamond
- sketch.shapes.flowcharts.Decision
- sketch.shapes.flowcharts.Choice
- sketch.shapes.flowcharts.Action
- sketch.shapes.geom.quad.horizontaltrapezoid
- sketch.shapes.geom.quad.HorizontalRectangle

USER-DEFINED SHAPE: Arrow

COMPONENTS

(SYNTAX: Type | Name)

opt.	sh.	Line	head1
opt.	sh.	Line	head2
opt.	sh.	Line	shaft

CONSTRAINTS

(SYNTAX: Constraint | Arguments)

isRotatable		
horizontal	shaft	
posSlope	head1	
negSlope	head2	
	shaft	
coincident	shaft.p1	head1.p1
coincident	shaft.p1	head2.p1
longer	shaft	head1
longer	shaft	head2
equalLength	head1	head2
acuteDir	shaft	head1
acuteDir	head2	shaft
leftOf	shaft.p1	shaft.p2
above	head1	head2

ALIASES

DISPLAY SPECIFICATIONS

EDITING BEHAVIORS

Figure 1-8: A GUI for hand entry of shape descriptions.

positives.

1.9 Active Learning with System Generated Near-miss Examples to Refine Concepts

This best-guess approximation is a plausible one, but, even with the perceptual rules, it is sometimes impossible for the computer to know exactly what variations are possible in a shape. Thus, the generated description, while syntactically correct, may still have conceptual errors: System-generated descriptions may be over-constrained, and human-generated descriptions may be both over- and under-constrained (although in

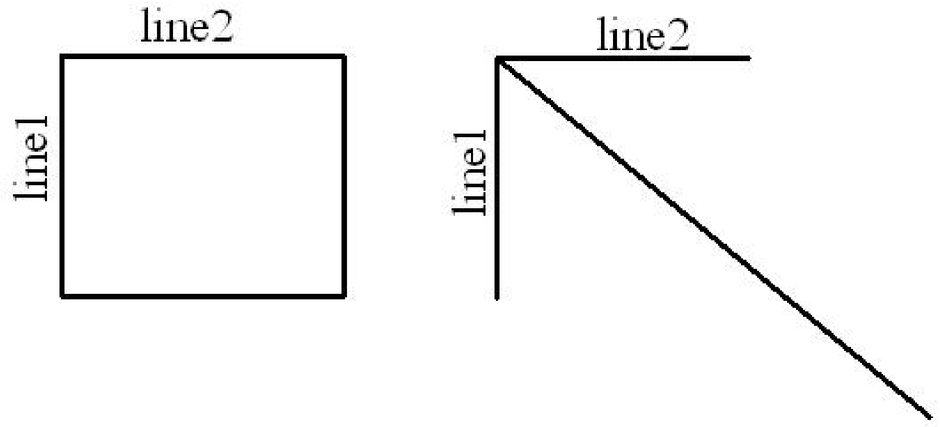


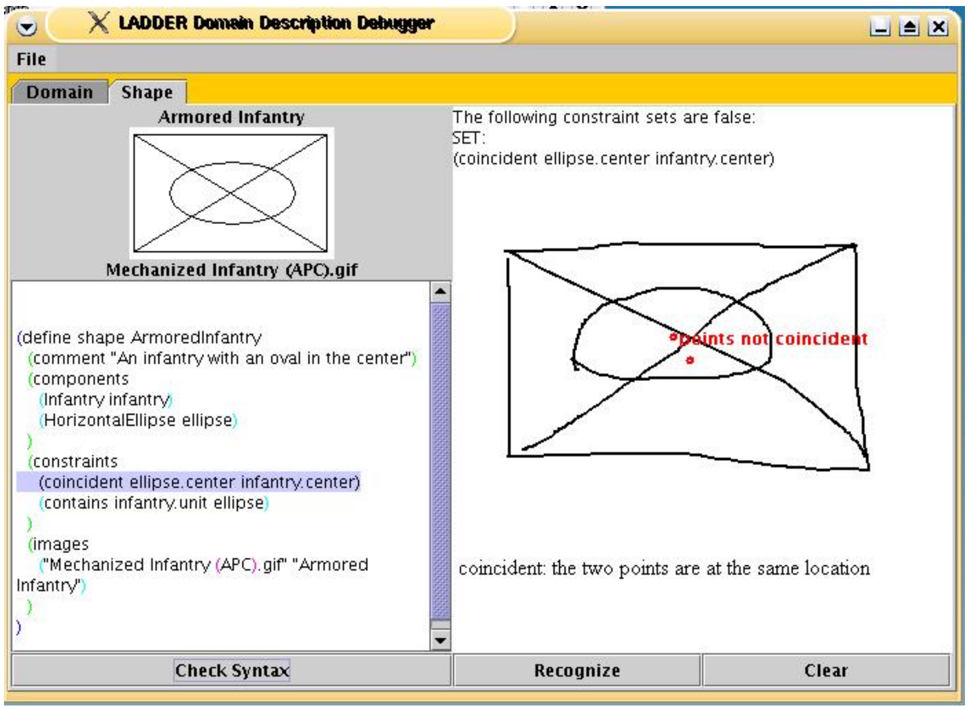
Figure 1-9: Lines 1 and 2 are identical in the square and in the arrow. However, the constraint PERPENDICULAR LINE1 LINE2 should be included in a square description, but not in an arrow description.

practice, they tend to be more under-constrained). Figure 1-9 shows the difficulty of automatically generating a perfect description; the components LINE1 and LINE2 look the same in both the square and the arrow. The constraint PERPENDICULAR LINE1 LINE2 is true for both shapes, and any algorithm that would include the constraint for one shape would include it for the other. However, if the constraint is missing from the square definition, the square definition will be incorrect, as it is under-constrained, but, if the constraint is included in the arrow definition, it will be over-constrained and incorrect.

1.9.1 User-generated Near-miss Examples

Thus, we need a way to test and refine the initial shape description. One way to do this is to build a system that allows users to draw several near-miss examples of the shape, and then modifies the description, based on these examples [214]. We built a system to do that; a screen shot is shown in Figure 1-10.

This system did learn descriptions from user-generated examples. Unfortunately, users proved to be poor at generating sufficiently informative examples. And, there is no guarantee that the user will ever draw the shape in a way that exposes the bugs



```
The following constraint sets are false:
SET:
(coincident ellipse.center infantry.center)
```

```
(define shape ArmoredInfantry
  (comment "An infantry with an oval in the center")
  (components
    (Infantry infantry)
    (HorizontalEllipse ellipse)
  )
  (constraints
    (coincident ellipse.center infantry.center)
    (contains infantry.unit ellipse)
  )
  (images
    ("Mechanized Infantry (APC).gif" "Armored
Infantry")
  )
)
```

coincident: the two points are at the same location

Figure 1-10: A screenshot of a system built to learn descriptions from user-generated near-miss examples.

in the description.

1.9.2 System-generated Near-miss Examples

To solve this problem, this research included the development an algorithm using a novel form of active learning [51] that automatically generates its own suspected near-miss examples, which are then classified as positive or negative by the developer. The algorithm is a modification of the traditional model of machine learning of concepts, in which a teacher supplies labeled examples (and non-examples) of the concept (e.g., "This is an arrow" "This is not an arrow"), and the system constantly updates its evolving version of the concept. Instead, in our model the system selectively generates its own (near-miss) examples and uses the human teacher as a source of labels. The system generates these examples to test whether components of its current concept description are necessary to the concept, or merely happened to be true of the initial example. (For example, is it necessary for both lines in the head of an arrow to be

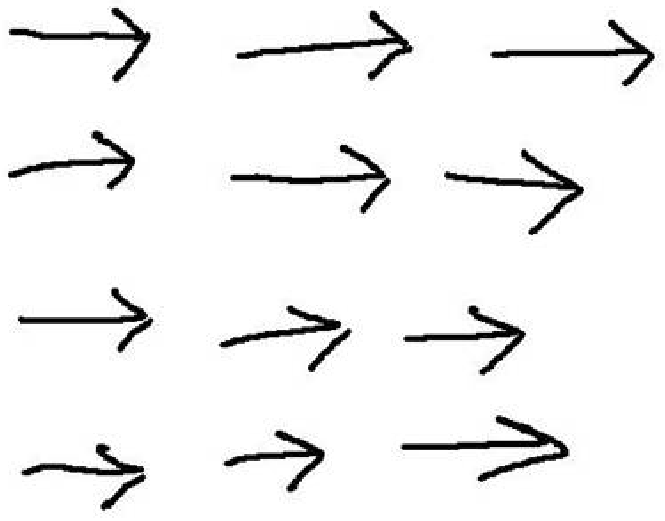


Figure 1-11: User-generated arrows.

the same length, or was this accidental in the original example?)

System-generated near-misses offer a number of advantages. They work even when the teacher does not know the complete concept description in advance; e.g., the teacher might not previously have thought about whether an arrow-like figure with unequal head lines is still an arrow. Also, the system can be an efficient learner simply by virtue of its ability to keep careful track of which parts of the concept description have been verified as necessary and which are yet to be tested, thereby generating only informative examples.

The result is a system that behaves somewhat like a persistent, literal-minded, but intelligent student who wants to get all of the details right and does so by asking, "And would this be an example? How about this one? And this one...?" When learning a concept, while working within a fixed vocabulary and rule-set, the computer learner knows exactly where its uncertainties lie in terms of the concept, and which hypotheses need to be tested and confirmed.

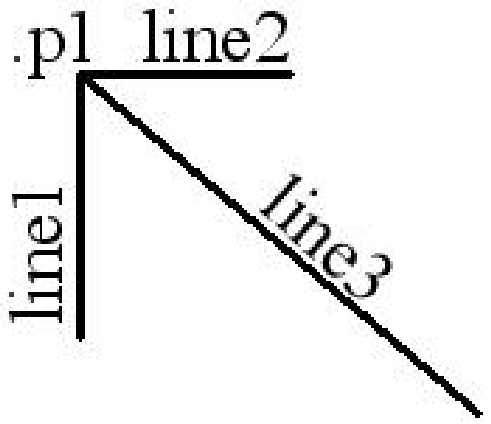
Active learning is a dialogue between a teacher and a student. The goal of the dialogue is to teach the student a concept that is known by the teacher. Examples are selected (by either the teacher or the student), and the teacher labels these examples. Our learning model is based on the principle that the learning participant (either the teacher or the student) who knows better which information is lacking in the student's formation of the concept should generate the near-miss examples.

In human-human (teacher-student) learning, humans are poor at knowing what they do not know; initially, the human teacher knows better what information the student is missing in his concept and provides the near-miss examples. However, as the student begins to learn the concept, at some point there is a transfer of knowledge to the student, and, as the student begins to understand the concept, he knows what information still needs to be confirmed. At this point, the student begins to generate his own near-miss examples, confirming and removing uncertainties, and saying such things as, "Oh, I think I get it. So, is this an example? What about this one? Yes, that makes sense; now, I understand."

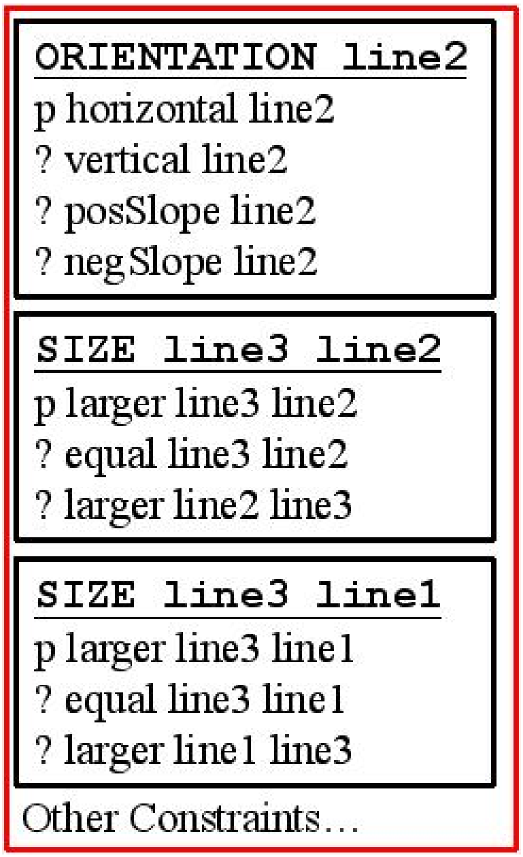
In our task of learning structural shape descriptions, the human developer is the teacher, and the computer is the student. Our situation is different from the human-human learning environment in that this task involves learning structural shape descriptions in a fixed domain with a fixed vocabulary and syntax with a limited number of options. Because the vocabulary is limited, the computer student can easily keep track of all existing possibilities and know exactly what information is still necessary to confirm the current shape concept. Conversely, a human teacher is not good at keeping all possible uncertainties in her mind at one time. In this case, the computer-student is better able to provide informative near-miss examples, allowing the computer-student to more quickly and effectively refine its concept.

1.10 Concept Learning Algorithm

The system uses these labeled examples to automatically learn a shape concept (in the form of a *LADDER* shape description) using a modification of the version spaces



(a) Initial positive example of an arrow.



ORIENTATION line2
p horizontal line2
? vertical line2
? posSlope line2
? negSlope line2

SIZE line3 line2
p larger line3 line2
? equal line3 line2
? larger line2 line3

SIZE line3 line1
p larger line3 line1
? equal line3 line1
? larger line1 line3

Other Constraints…

(b) Arrow concept after initial positive example.

Figure 1-12: Initial example shape and shape concept.

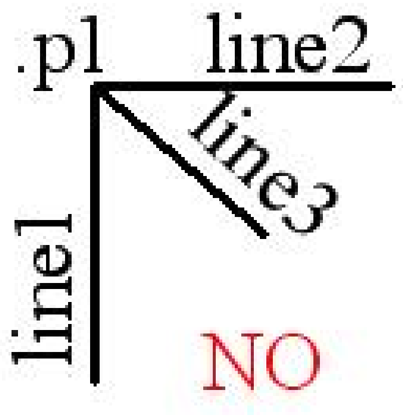


Figure 1-13: Negative arrow example

algorithm [163]. Our modification, which is based on mutually-exclusive perceptually-important constraints, better suits our domain in that it can learn shape descriptions with negative and disjunctive constraints.

Figure 1-12 shows a hand-drawn example of a shape and the initial concept generated for it. As more positive and negative examples are labeled, the system continues to update its concept space using the algorithm. Because constraints are often interrelated in our domain of structural shape descriptions, it may be difficult to tell which constraint causes a negative example. For example, the user may state that the arrow-like example in Figure 1-13 is not an arrow. But if the system does not yet know yet if the two head lines of the arrow must be the same size, the system does not know if the SHAFT of the arrow must not be shorter than HEAD1, HEAD2, or both.

If all shape permutations are generated, it is possible to eventually converge to a single concept using this algorithm. However, we do not want to produce all of the permutations because that would mean the user would have to label several hundred examples. Ideally, we would like to prevent negative examples from causing branching (which may require many more examples to prune down again to a single concept). However, this is difficult to do, as many of the constraints are interrelated in our domain. Thus, we have developed a heuristic called the *purple cow heuristic* to prune the space of possibilities.

The *purple cow heuristic* works as follows: “I have never seen a purple cow, so I am going to assume one does not exist.” This heuristic, applied to the structural shape domain, is reworded as “I have never seen this constraint exist in a positive example shape, thus, I am going to assume it will never exist in a positive example shape.”

This heuristic works particularly well in the structural shape description domain, as many of the constraints are interrelated, and the system may not have produced an example which changed only a single constraint to confirm that it caused the negative example.

1.11 Lessons Learned from Near-miss Use

Ten users employed the near-miss generation system to produce shape descriptions. On many occasions, the system worked well so as to combine the near-miss generation and the concept learning algorithm to determine the final shape, needing only 20 shapes to be able to fully reduce the space of 4773 shapes. However, in other cases, the system did not work as well due to the lack of certainty about the possibility of existence of a shape with a particular set of constraints and due to the slowness of the shape generation algorithm.

1.12 Future Work

This research has several future implications in research. In particular, by simplifying the creation of sketch interfaces, teachers may be able create their own sketch interfaces for use in their classroom with hopes to improve classroom learning and pedagogy. Also, by combining this research with multiple forms of context and feature-based recognition techniques, a larger class of shapes may be identifiable.

1.13 Principle Themes of this Document

Some of the general principles explored in this thesis are the following:

1. Abstracting away the signal processing details in a user interface will enable user interface and domain experts to create more sketch systems.
2. A perception-based UI development language modeled on how humans react is natural and easy for humans to understand.
3. A recognition system based on human perception will align itself to how humans recognize something, and, thus, be accurate according to humans.
4. If a UI language is easy to use, better UI systems will be developed.
5. A UI development language should abstract all details except those that are domain specific.
6. In this thesis, domain-specific information includes what shapes can be observed in a domain, how those shapes are to be recognized, what should happen once those shapes are recognized, and how those shapes can be modified.
7. By including only perceptually-important elements in the domain, the language can be simplified, making it easy to use.
8. Users make syntactical and conceptual mistakes when generating shape descriptions. Conceptual mistakes include over- and under-constrained descriptions.
9. Users are poor at coming up with informative positive and negative examples for the same reason that they forget constraints in typed shape descriptions and that they are bad at generating good test cases.

10. The learning participant that generates the near-miss examples should be the learning participant that knows where the uncertainties in a concept lie.

Chapter 2

Previous Work

2.1 Motivation for Sketch-based User Interfaces

Sketching is a natural interface for many domains. For instance, software design diagrams (UML, flow charts), course of action diagrams, finite state machines, music notation, and mechanical engineering diagrams are often drawn by hand on paper. Currently, input of these diagrams into a computer is done using CAD or CASE software that can be clumsy and nonintuitive; thus, these designs are input into the computer only when necessary. The ideal or most natural input of these diagrams would be as they were first completed, through hand-drawn sketching. Recognizing these sketches can be difficult and proves to be an interesting AI problem.

Sketch recognition systems, in which users draw directly on a Tablet PC (as opposed to off-line sketches that are later interpreted [169]), have been developed for domains such as mechanical engineering [6] [141] [200] [10] [11] [166] [131], UML class diagrams [99], [56] [118] [143], [105], webpage design [147], 3D drawings [108] [181] [120] [149] [216], calendar notation [138], architecture [91] [59], GUI design [41] [145], powerpoint slides [117], virtual reality [58], animation [72], stick figures [156] [155], course of action diagrams [179] [71] [80], mathematical expressions [159] [111], music notation [81] [32], and even dance notation [90].

Sketch interfaces provide a number of benefits, including their ability to 1) interact more naturally than a traditional mouse-and-palette tool by allowing users to hand sketch the diagram, 2) connect to a back-end system (such as a CAD tool) to offer real-time design advice, 3) recognize the shape as a whole to provide powerful editing capabilities, 4) beautify diagrams, removing mess and clutter, and thereby 5) notify the sketcher that the shapes have been recognized correctly.

However, sketch recognition systems can be quite time-consuming to build and require signal-processing expertise if they are to handle the intricacies of each domain. This researcher wants to enable user interface designers and experts in the domain itself, who need not be experts in sketch recognition at the signal level, to be able to build a sketch recognition system for their domain. This thesis describes a set of ideas and techniques to enable developers to build sketch recognition systems.

The focus of the sketch-based systems described in this document is to recognize domains that consist of a set of iconic shapes that can be described geometrically and compositionally from lines, ellipses, arcs, curves, points, and spirals. The system does not handle artistic freeform sketches.

2.2 History of Sketch Interfaces

2.2.1 The Birth and Death of Sketching Interfaces

Sketching interfaces have been around for a long time. Ivan Sutherland created the Sketchpad system in 1963 on the TX-2 computer at MIT [206]. (See Figure 2-1.) His system has been called the first computer graphics application. The system, created before the invention of the computer mouse, provided the user with a light pen as an input device. A user could create a complicated two-dimensional graphical scene through a series of editing commands and primitive graphical commands.

The light pen was used in conjunction with keyboard input to allow users to create simple graphical primitives, such as lines and circles, and editing commands, such as "Copy". The keyboard could be used to place additional constraints on the geometry and shapes in the scene. By defining appropriate constraints, users could develop structures such as complicated mechanical linkages and then move them about in real-time.

The Sketchpad system was based on vector graphics. Raster graphics, despite its inability to produce the smooth continuous lines available with vector graphics, proved to have many advantages over vector graphics [74]. Computers based on raster graphics had a much lower cost. Raster graphics also provided the ability to display an area filled with solid colors or patterns. Most importantly, the refresh process for raster graphics is independent of the complexity of the scene (where complexity is based on the number of objects in the scene) and, thus, because of the high refresh rates available, any scene can be refreshed flicker free.

Vector graphics and its light pen were quickly superseded by raster graphics and the ubiquitous mouse. Pen-based interfaces disappeared from mainstream computer interfaces for many years, with the mouse being the most common input device for graphical applications. Despite the many advantages of a mouse, a mouse is very difficult to use for sketching. It does not have the natural feel of a pen, nor does it

Figure 2-1: Ivan Sutherland and the Sketchpad system.

provide a pen's accuracy. Because the mouse was found to be difficult to sketch with, computer automated design (CAD) systems were based on a mouse-and-palette user interface, rather than a sketching interface.

2.2.2 The Rebirth of Sketching Interfaces

In the last decade, pen-based interfaces have regained popularity. PDAs (Personal Digital Assistants) [192] such as the Palm Pilot [177] [69] and the iPAQ Pocket PC [199] entered the market. PDAs come with a stylus and a screen that can be sketched on. With the influx of PDAs, there has been a growth of Graffiti^{TM}-type interfaces. Companies such as Wacom [191] have created sketching tablets that use a stylus as if it were a mouse for a desktop computer. Companies such as Mimio [52] have created electronic whiteboards, which consist of a regular whiteboard, a projector projecting the drawn contents, and special markers acting as cordless mice. Tablet PCs [160] now allow users to sketch directly onto their laptop screens using a Wacom pen [191] [161].

Sketch-based interfaces are useful for a number of reasons. PDAs use a Graffiti^{TM}-type interface to allow users to hand write their notes. PDAs are built to fit easily in a pants pocket, but still provide the computer power and ease of use of a computer-based organizer. Because of the small size of PDAs, a traditional keyboard is not practical. Handwriting recognition allows the pen to be used in place of a keyboard.

Many things are much more naturally input with a pen or sketch-based interface than with a keyboard or mouse. The clunky-feeling of the mouse lacks the precision of a pen-based stylus. For many people, drawing architectural sketches or mechanical engineering designs would be very difficult without a pen, and many of the CAD systems lack the natural feel and spontaneity of freehand sketching. Because of the lack of free drawing available in a CAD system, many designers first sketch a freehand diagram of their design before entering it into a CAD program.

Most importantly, sketch-based interfaces are useful because many people prefer to sketch. When given the option between sketching a design or using a mouse-and-palette tool, users will generally choose to sketch the design [116]. Hse et. al. performed a Wizard of Oz [55] experiment comparing the two design methods. During the experiment, not only did users say they preferred a sketch-based interface, they also requested more sketching flexibility, such as the ability to draw with multiple strokes.

2.2.3 Sketch-based Applications

A myriad of applications with sketch-based user interfaces have been created for use with pen-based input devices. Many sketching applications are based on a list of domain symbols or icons; the user interacts with the system by drawing symbols in the domain.

Originally, the objects in these sketches were recognized using trained gesture recognition. Rubine [185] was one of the first to implement trained gesture recognition. The Rubine recognition engine recognizes objects statistically with the use of a linear discriminator, which processes a single stroke and determines certain features of it. The Rubine system does not break down the stroke into line segments or curves, requiring the sketcher to draw each object with a single stroke. Sketchers must learn a particular stylized way of drawing. Graffiti^{TM}[192], a language for hand writing text used on Palm Pilots and other PDAs, uses a recognition algorithm similar to the Rubine algorithm.

Labanotation, a system for recording and analyzing human movement, was first published by Rudolf Laban in 1928 [38]. His analysis of movement is based on spatial, anatomical, and dynamic principles. The LabanPad contains a handwriting recognition algorithm that is based on Labanotation. As the user writes down Labanotation symbols using a pen, they are analyzed, tokenized, and redisplayed [90].

Landay [141] [142] created SILK (Sketching Interfaces Like Krazy), a tool that allows users to sketch interactive user interfaces. SILK was one of the first systems that recognized a sketch and allowed interactive use of the sketch without replacing the strokes with cleaned-up strokes, allowing the user to view and modify her original strokes. SILK and many other systems were based on the Rubine recognition engine.

Denim, also by Landay [147] [168], recognizes boxes and two link types to allow users to sketch and design web pages. In Denim, the link types are differentiated not by the geometrical properties of the drawn links, but by what the links connect. Other informal sketching tools to encourage brainstorming, but that do not recognize drawn objects, have been developed [110] [79] [113] [140].

Ligature [75], also based on link connections, is a sketch-based system that connects to Metaglue [48], the multi-agent software system for the Intelligent Room, to configure hardware connections in the MIT AI Lab Intelligent Room.

Several other sketch recognition systems in other domains have also been developed. Early systems attempted to recognize draft drawings of mechanical engineering diagrams [85]. SketchIt [201] is a sketch-based user interface for mechanical engineering designs, recognizing hand-drawn (as opposed to drafted) diagrams.

JavaSketchIt [41] is a sketch-based tool for GUI design in Java. Kullberg [138] presents a way of annotating one's daily calendar by using hand-drawn sketching. SketchVR is a tool to recognize virtual reality architecture sketches [58] [59].

Course of Action Diagrams are used by the military to plan and depict battles, depicting unit movements and tasks in a given region [171]. COA diagrams are usually hand-drawn by the military, and these sketches have been successfully recognized by Quickset and other sketch-based tools that enable multiple users to create and control military simulations [179] [71] [70].

Edward Lank et al. built a UML recognition system that uses a distance metric [143] for recognition. Each glyph (square, circle, or line) is classified based on the total stroke length compared to the perimeter of its bounding box (e.g., if the stroke length is approximately equal to the perimeter of the bounding box, it is classified as a square). Similar to other feature-based algorithms, this algorithm can cause many false positives. (For example, the letter M can be detected as a box.) Although the system does allow users to draw somewhat naturally, it does not allow them to edit naturally; they do not sketch edits to their diagrams, but have to use correction dialogue boxes.

Ideogramic UML $^{\mathrm{TM}}$[118] has developed a gesture-based diagramming tool to recognize UML diagrams. The tool is based on a Graffiti$^{\mathrm{TM}}$-like implementation and requires users to draw each gesture in a single stroke, and in the direction and style specified by the user manual. As a consequence, some of the gestures to be drawn only loosely resemble the output glyph. For example, φ is used to indicate an actor, drawn by the system as a stick figure.

Eric Lecolinet [145] has created a system to design GUIs that is based on modal interaction, in which the user is required to select an appropriate drawing tool. His system is quite different from ours in that it does not allow free-hand drawing and, thus, does not perform sketch recognition; rather, it uses gestures for diagram manipulation.

The systems above require that the user draw shapes similar to the developer, but are not intended to be retrained for every user, and simply suffer lower recognition rates when a user does not draw the shape as intended. Other systems try to capture individual user styles by having every user intricately train the system [112].

2.3 Natural Sketch Recognition Based on Geometric Properties

Sketch systems using a feature-based algorithm for recognition, such as a Rubine[185] or Graffiti$^{\text{TM}}$-type[192] algorithms require users to learn a particular stylized way of drawing. What these algorithms lose in natural interaction by requiring the sketcher to draw in a particular style, they gain in speed and accuracy, as these algorithms are fast and effective. Rather than recognizing shapes, the algorithm recognizes sketched gestures, in which each gesture stands for a single shape. These sketched gestures focus more on how something was drawn than on what it looks like. The recognition algorithm requires that each gesture be drawn in a single stroke in the same manner (stroke direction, speed, etc.) each time. Each gesture is recognized based on a number of features of that stroke such as the initial angle, the end angle, the speed, number of crosses, etc. Because of these requirements, the gesture representing the shape may look different from the shape itself (e.g., it would be impossible to draw a cross with a single stroke, so it may, instead, be represented by a ribbon-like gesture). Also, even if a sketcher draws a shape that looks the same as the required gesture, it may not be recognized because it does not have the same underlying features of the stroke (stroke direction, speed, or number of strokes).

Our goal is to build sketch recognition systems that allow sketchers to draw as they would naturally, i.e., without having to learn a new set of stylized symbols. As long as the shape looks like the final shape, the shape should be recognized, independent of the number, direction, or order of the drawn strokes. Apte was one of the first researchers to recognize multi-stroke shapes using geometry. [17]. Two previous systems in our group – Assist [6] and Assistance [174] – took a step in this direction – providing a sketch-based user interface for mechanical engineering design that allowed users to draw shapes with multiple strokes and performed recognition based on shape. Other systems use geometric constraints to recognize shapes in sketch recognition [62] [86] [125].

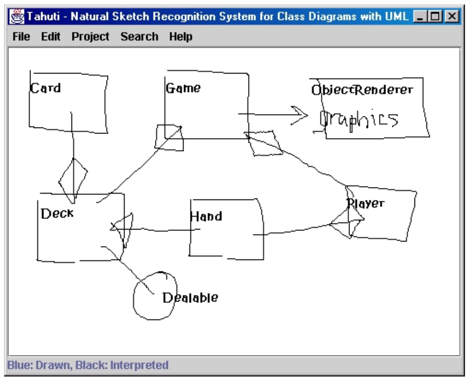
To allow for natural drawing in our sketch recognition systems, shapes are described (and recognized) in terms of the subshapes that make up the shape and the geometric relationships (constraints) between the subshapes. Strokes are first broken into a collection of primitive shapes, including lines, ellipses, arcs, spirals, points, and curves, using techniques from Sezgin [195] [194]. (Current improvements have also integrated techniques from Yu [215] to recognize a larger class of shapes. Future plans include the possible integration of techniques from Cates and Hse [45] [115].) A higher-level shape is then recognized by searching for possible subshapes and testing whether the appropriate geometric constraints hold. The geometric constraints confirm topology, angles between lines, relative sizes, and the like. Recent work of this researcher has created a recognizer that also distinguishes helixes, spirals, and overtraced ellipses. That work is not described in this document.

2.3.1 Tahuti

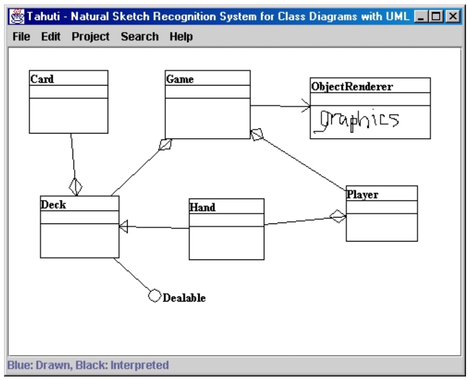
To demonstrate that recognition could be done using a shape-based model, this researcher built Tahuti[1] [99] [105] [99], a system for recognizing UML class diagrams [37] [1] using geometric constraints. To test the usability of Tahuti, this researcher performed a field experiment in which subjects compared Tahuti to a paint program and to Rational Rose^{TM}. Subjects created and edited a UML class diagram, using each method and quantifying the ease of drawing and editing of each method.

Sketching is a natural and integral part of software design, as software developers use sketching to aid in the brainstorming of ideas, visualizing programming organization, and understanding of requirements. UML diagrams are a de facto standard for depicting software applications. Unfortunately, when it comes to coding the system, the drawings are left behind, and the sketched information has to be re-input into the computer using a CASE tool. Traditional CASE (Computer Automated Software Engineering) tools, such as Rational Rose^{TM}, give users powerful editing features and

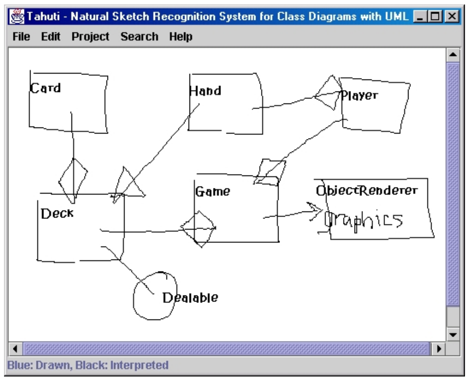
[1]Tahuti, also known as Thoth, is the Egyptian god of wisdom. He always carried a pen and scrolls upon which he recorded all things.


Tahuti - Natural Sketch Recognition System for Class Diagrams with UML
File Edit Project Search Help
Card
Game
ObjectRenderer
Graphics
Deck
Hand
Player
Dealable
Blue: Drawn, Black: Interpreted

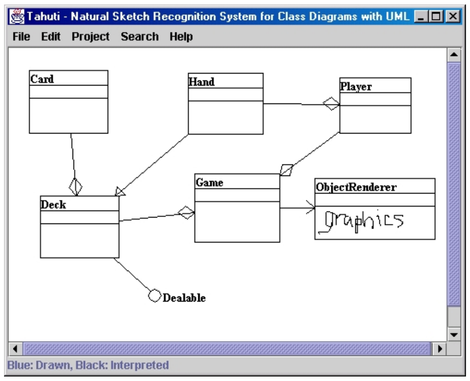
(a) Hand-drawn UML class diagram in Tahuti


Tahuti - Natural Sketch Recognition System for Class Diagrams with UML
File Edit Project Search Help
Card
Game
ObjectRenderer
Graphics
Deck
Hand
Player
Dealable
Blue: Drawn, Black: Interpreted

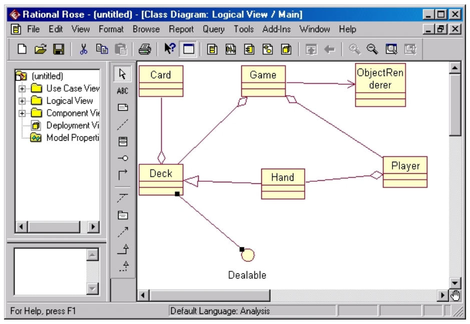
(b) Recognized UML class diagram in Tahuti of Figure 2-2(a)


Tahuti - Natural Sketch Recognition System for Class Diagrams with UML
File Edit Project Search Help
Card
Hand
Player
Game
ObjectRenderer
Graphics
Deck
Dealable
Blue: Drawn, Black: Interpreted

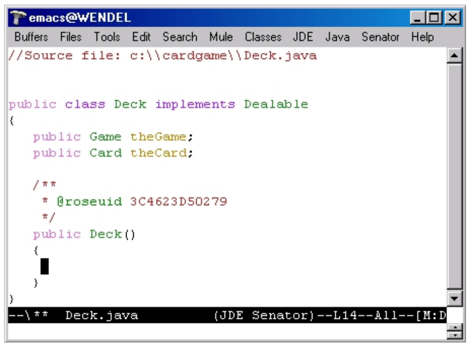
(c) Diagram in Figure 2-2(a) with classes moved


Tahuti - Natural Sketch Recognition System for Class Diagrams with UML
File Edit Project Search Help
Card
Hand
Player
Game
ObjectRenderer
Graphics
Deck
Dealable
Blue: Drawn, Black: Interpreted

(d) Interpreted diagram of Figure 2-2(c)


Rational Rose - (untitled) - [Class Diagram: Logical View / Main]
File Edit View Format Browse Report Query Tools Add-Ins Window Help
(untitled)
Use Case View
Logical View
Component View
Deployment View
Model Properties
Card
Game
ObjectRenderer
Deck
Hand
Player
Dealable
For Help, press F1
Default Language: Analysis

(e) Diagram in Figure 2-2(a) sent to Rational Rose^{TM}


emacs@WENDEL
Buffers Files Tools Edit Search Mule Classes JDE Java Senator Help

```
//Source file: c:\\cardgame\\Deck.java


public class Deck implements Dealable
{
   public Game theGame;
   public Card theCard;

   /**
    * @roseuid 3C4623D50279
    */
   public Deck()
   {
   
   }
}
--\** Deck.java          (JDE Senator)--L14--All--[M:D
```

(f) Stub code generated by Rational Rose^{TM}Figure 2-2(e)

Figure 2-2: A hand-drawn UML class diagram in Tahuti, the recognized shapes in Tahuti, the recognized shapes passed off to Rational Rose^{TM}, and the generated stub code produced by Rational Rose^{TM}.

even allow users to automatically generate skeleton user code. However, these CASE tools give the users very little, if any, flexibility to create a diagram. Users do not have the freedom to sketch their designs and are required to learn a large number of commands before they can use the system with ease.

Thus, this researcher felt that a multi-stroke sketch recognition system for UML Class diagrams would be a way to bridge that gap, allowing users to sketch the diagrams on a tablet or whiteboard in the same way they would on paper. The sketches are then interpreted by the computer, allowing the sketch itself to take an active role in the coding process. This researcher chose to build a sketch system for UML class diagrams because of their central role in describing program structure. And, many of the symbols used in class diagrams are quite similar, and, hence, offer an interesting challenge for sketch recognition.

Domain Knowledge

Class diagrams describe the static structure of an object-oriented software system, rather than how it behaves. Class diagrams consist of (i) general classes, (ii) interface classes, and (iii) associations between two classes. UML uses a rectangle to indicate a general class, while an interface class is represented by a circle or rounded rectangle. There are three types of associations: (i) A dependency association exists if one class calls a method from another class, including the constructor. The dependency relationship is represented by an arrow with an open head. In Figure 2-2, the Game class is dependent on the Graphics class. (ii) A generalization or inheritance association exists if one class is a kind of or extension of another class. The inheritance relationship is represented by an arrow with a triangular head. In Figure 2-2, the Hand class is inherited from the CardDeck class. (iii) An aggregation association exists if one class is part of another. The aggregation relationship is represented by an arrow with a diamond head. In Figure 2-2, the Card class is part of the CardDeck class.

Recognition

Our system differs from graffiti-based approaches in that it allows users to draw an object as they would with pen and paper. The system recognizes objects based on their geometrical properties by examining the line segments' angles, slopes, and other properties, rather than requiring the user to draw the objects in a predefined manner. Recognizing the objects by their geometric properties allows users the freedom to sketch and edit diagrams as they would naturally, while still making it possible to maintain a high level of recognition accuracy.

Tahuti recognizes five shapes: a general class (represented by a rectangle), an interface class (represented by an ellipse), an inheritance association (represented by an arrow with a triangle-shaped head), an aggregation association (represented by an arrow with a diamond-shaped head), a dependency association (represented by an arrow or line), and an interface association (represented by a line, but connecting a class to an interface). If a collection of strokes is not recognized as one of these shapes, the collection is left unrecognized. The system also accepts text input through the keyboard.

In order to recognize the objects created from multiple strokes by their geometrical properties, this researcher created a multi-layer framework of recognition in which strokes are preprocessed, selected, recognized, and then identified.

After each stroke is drawn, rudimentary processing is performed on the stroke, classifying it as an ellipse or a series of line and curve segments. A collection of spatially and temporally close strokes is chosen, and the line segments contained in the collection of strokes are then recognized as either an editing command or a viewable object.

Figure 2-3 shows the stages of the multi-layer recognition framework applied to a drawn UML aggregation association.

Stage 1: Preprocessing – At the most basic level, strokes drawn by the user are

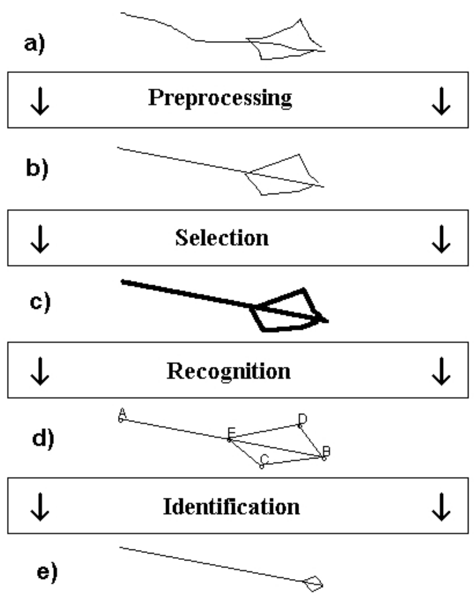


Figure 2-3: Multi-layer framework of recognition used in Tahuti: A UML aggregation association is identified using the multi-layer recognition framework. a) The association was originally drawn using two strokes. b) During the preprocessing stage, the original strokes are processed into line segments. c) The two strokes of the arrow are then selected for recognition. d) Recognition occurs on the two strokes, at which point a UML aggregation association is deemed to be a possible interpretation. e) The collection of strokes is identified as a UML aggregation association.

processed using algorithms for stroke processing developed in our group [194]. The preprocessing stage uses stroke timing data to find possible corners, as users tend to slow down when drawing a corner. A stroke is processed only once, immediately after having been drawn. The stroke is fit to each of the following: 1) an ellipse, 2) a line, 3) a polyline, which is a collection of line segments, and 4) a complex shape, which is a collection of line segments and bezier curves. Along with the original data points, the stroke data structure contains each possible interpretation and its probability of correctness.

Figure 2-3a shows the originally drawn strokes of a UML aggregation association. The diamond-headed arrow was drawn using two strokes. The stroke is processed immediately after it is drawn. The data structure of the strokes will contain a fit for a best fit ellipse, line, polyline, and complex shape. Figure 2-3b shows the polyline interpretation of the strokes.

Stage 2: Selection – After the recently drawn stroke has been preprocessed, the stroke is combined with zero or more unrecognized strokes to form a collection of strokes. This collection of strokes is then sent to the recognizer, where it determines whether the combined strokes form a recognizable object or an editing command.

Ideally, all possible stroke combinations would be tested for possible recognition of a recognizable object or editing command. However, if we allow the system to test for all possible stroke combinations, it would take exponential time, based on the number of strokes to identify an object. While this may be OK for small diagrams, this would be unacceptable for large diagrams, making the system unusable. To reduce the number of stroke collections for recognition, we use spatial and temporal rules to prune off stroke collections.

To ensure that all interactions take polynomial time based on the number of strokes, we limit the number of strokes in a collection to a threshold. Experimentally, we have found that 9 strokes is an acceptable threshold. Since users tend to draw an entire object at one time, completing the drawing of one object before drawing the next, it is generally safe to form stroke collections consisting only of strokes drawn recently. Thus, only the last nine unrecognized strokes can possibly be included in a stroke collection.

All recognizable objects within the UML class diagram domain are connected objects. Thus, we require all strokes within a collection to be within close proximity of other strokes in the collection. Let C be the collection of all of the strokes. Let S be a subset of the strokes. For every subset S, where S is nonempty and $C - S$ is nonempty, we require that the smallest distance between the subsets be less than a threshold, τ. ($SubsetDistance(S, C - S) < \tau$) Experimentally, τ was chosen to be 10 pixels.

$$SubsetDistance(X, Y) = Min(\bigcup_{i=1..n} \bigcup^{j=1..m} D(X_i, Y_j)) \tag{2.1}$$

In the above equation, n and m are the number of line segments in X and Y respectively and X_i represents the ith line segment. D is the distance function computing the distance between two points.

Figure 2-3c shows the two strokes of the UML aggregation association selected. Note that this is not the only collection that would have been created. Assuming that the arrow shaft was drawn first, after the arrow shaft was drawn, a stroke collection would have been formed with only that stroke. Another stroke collection would have been formed with only the stroke of the arrow head. If other unrecognized strokes are present in the diagram, several more stroke collections that include these strokes would be created for recognition testing. After all stroke collections have been created, the recognition stage attempts to identify the stroke collections as possible viewable objects or editing commands.

Stage 3: Recognition – During the recognition stage, all stroke collections are examined to see whether a particular stroke collection could be interpreted as a viewable object or an editing command. An editing command is a collection of strokes indicating deletion or movement of a viewable object. The system currently recognizes eight viewable objects: a general class, an interface class, an inheritance association, an aggregation association, a dependency association, an interface association, and text. The system may also choose to leave strokes unrecognized. The algorithms used in the recognition stage will be described in more detail in the next section.

If more than one interpretation is possible for any stroke collection, the final interpretation is deferred until the identification stage. Figure 2-3e shows the UML aggregation association interpretation chosen by the recognition system. Other stroke collections presented to the recognition stage also have interpretations. For example, the collection of strokes consisting only of the arrow head stroke is recognizable as a general class since it forms a square-like shape. The decision between choosing the general class interpretation and the UML aggregation association is deferred until the identification stage.

Stage 4: Identification – During the identification stage, a final interpretation is chosen, and a collection of strokes is identified as a viewable object or an editing command. All possible interpretations found in the recognition stage from the stroke collections are presented to the identification stage. The identification stage selects the final interpretation, based on the following rules:

Object Movement: An interpretation of object movement has priority over any other possible interpretation. Object movement recognition is interesting in that it is the only interpretation that can be determined while the stroke is still being drawn. If object movement is recognized, the multi-layer recognition framework will be short-circuited, preventing the stroke from being recognized by other algorithms. Immediate recognition is necessary for movement to allow the user to visually move the objects in real-time, rather than having the object move only after the stroke is completed.

Any Interpretation: Any interpretation is preferred to no interpretation, where no interpretation leaves the stroke collection as a collection of unrecognized strokes.

Many Strokes: We prefer to recognize collections with a larger number of strokes, rather than fewer, since our goal is to recognize as much of what the user draws as possible.

Correctness Probability: Each algorithm has a ranking, based on its probability of correctness. The probability of correctness is a combination of both prior and predicted probability. Certain recognizers have a higher level of accuracy than others, creating a prior correctness probability. Predicted probability is calculated during recognition: for example, the ellipse fit predicted probability of correctness is much higher for a perfectly drawn ellipse than for a crooked ellipse. If more than one interpretation is still possible, the interpretation with the highest ranking is then chosen.

After the final interpretation is chosen, the associations are examined to see whether any unconnected associations can be connected to a class. This is done by checking whether an association endpoint lies on or near a general or interface class.

During the recognition stage of the multi-layer recognition framework, stroke collections are tested for possible interpretations. In particular, we present here the recognition algorithms for rectangle, ellipse, arrow, and editing action recognition. Each algorithm is hand-constructed to recognize a particular shape using geometric principles. These shape-based algorithms are not the same as used by the LADDER/GUILD system, but rather form a model and inspired the LADDER-based automatically generated algorithms described later in this document.

Rectangle Recognition: General classes are represented as rectangles in UML class diagrams. To recognize rectangles, we constructed an algorithm that is based on a rectangle's geometrical properties. The algorithm does not require that the class be parallel to the horizontal plane or that it be created from a single stroke or even one stroke per side. The algorithm's inputs are the line segments of the polyline fit of the preprocessed strokes. (See Figure 2-4a-b.) The six steps are:

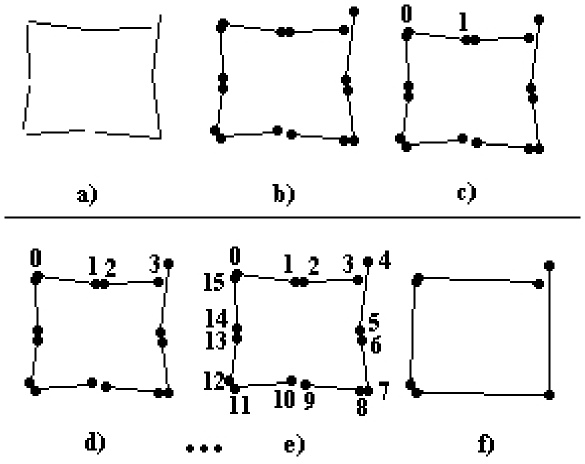


Figure 2-4: Rectangle Recognition Process. a) The stroke is fit to a polyline, and the line segments of the fit are shown here. b) The endpoints of the line segments from a) are specified. c) The endpoints of one line segment have been labeled. d) The endpoints of two line segments have been labeled. e) All line segments have been labeled. f) The new line segments after the joining.

1. Confirm that the preprocessed collection of strokes consists of at least 4 line segments of non-trivial size (> 10 pixels).
2. Order the lines into a circular path by numbering the endpoints one by one:
   (a) Select a line segment to start. Label its first endpoint 0. Label its other endpoint 1. (See Figure 2-4c.)
   (b) Find the closest unlabeled endpoint to the last labeled endpoint n. Label it $n+1$, and label the other endpoint of the segment $n+2$. (See Figure 2-4d-e.)
   (c) Repeat above until all endpoints are labeled.
3. Confirm that first endpoint labeled is relatively close to the last endpoint labeled (i.e., that the distance is $< 1/4$ of the distance between the two points furthest apart).
4. Join lines that have adjacent endpoints with similar slopes. (See Figure 2-4f.)
5. Confirm that there are four lines left.
6. Confirm that every other line is parallel and that adjacent lines are perpendicular.

The above algorithm recognizes rectangles containing any number of strokes. The strokes can be drawn in any order, and the strokes can stop or start anywhere on the side of the rectangle. The algorithm emphasizes that the rectangle be recognized by its geometrical properties rather than the method in which it was drawn. This method allows users to draw as they would naturally, without sacrificing the recognition accuracy.

Ellipse Recognition: Interface classes are represented as ellipses in UML class diagrams. After a stroke has been preprocessed, if the ellipse fit has the highest probability compared to the complex shape, polyline, or line fit, the interface class recognition algorithm accepts the stroke as an interface. An ellipse must be drawn with a single stroke, as we use Sezgin's toolkit for recognition [194]. To recognize multi-stroke ellipses, we could assemble two curves together in a process similar to the rectangle recognizer, but, in practice, this has not been necessary. The single-stroke requirement for the interface class is not a hindrance to the user since circles are almost always drawn with a single stroke.

Arrow Recognition: Here, we present two methods for arrow recognition, geometrical and contextual. The geometrical method is used if the user has drawn an arrow, complete with an arrow head, to specify the association type. The contextual method is employed if the user has drawn only the arrow shaft connecting two classes, letting the application assume the dependency association.

Geometrical Method for Arrow Recognition: Associations are represented by arrows in UML, of which there exist three types: aggregation association with a diamond arrow head, inheritance association with a triangular arrow head, and dependency association with an open arrow head (Figure 2-5). The recognition algorithm uses the polyline fit of the preprocessed strokes. To facilitate recognition of all three types, we identified five feature points (A, B, C, D, E), as labeled in Figure 2-3d and Figure 2-5.

1. Locate the arrow shaft by locating the two points farthest from each other (points A and B).
2. Locate the arrow head ends by locating points farthest from the arrow shaft on either side (points C and D).
3. Let point E be the point on line AB that is twice the distance from B as the intersection point of lines CD and AB.
4. Classify each of the line segments as a part of the arrow shaft, an arrow head section, or unclassified (AB, BC, BD, CD, CE, DE, or unclassified), based on the line's bounding box, slope, and y-intercept.
5. Compute the total length of each of the line segments in each section (AB, BC, BD, CD, CE, DE, or unclassified). A section is said to be filled if the total

length of each of the line segments in each section is greater than half the ideal length of the segment.

6. Confirm that sections AB, BC, and BD are filled.

7. Confirm that the unclassified section accounts for less than 1/4 of the sum total of the length of all of the strokes.

8. Based on the results of the line-segment classification, classify the arrow type, as follows:

   (a) open head: CD, CE, and DE not filled

   (b) diamond head: CE and DE filled

   (c) diamond head: CD not filled and either CE or DE filled

   (d) triangular head: Either CE or DE not filled and CD filled

(The first three steps of the algorithm serve to reduce the search space so the system does not have to try all possible permutations. In *GUILD* we have built an indexing algorithm to solve a similar problem. The following steps attempt to fill the missing segments; a more discrete approach is currently used in *GUILD*: *GUILD* first attempts to merge overlapping lines into a single line, and then attempts to find a single line which will fill the missing segment. The algorithm described above, however, is slightly more flexible in template filling, and it would be interesting to investigate integrating this alternate template-filling method into *GUILD* as future work.)

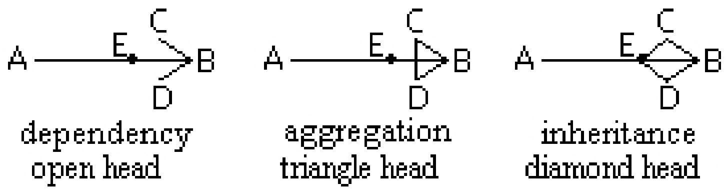


Figure 2-5: Points A, B, C, D, and E, as determined in the arrow recognition algorithm

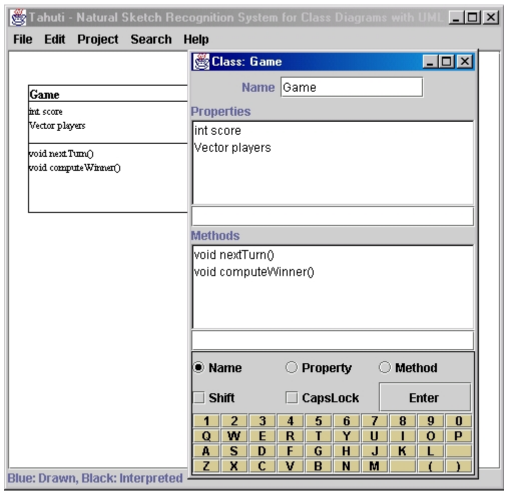
Contextual Method for Arrow Recognition: Contextual information can be used to recognize arrows. If a stroke without a specified arrow head starts in one class and ends in another, the stroke is interpreted as an arrow. The stroke is assumed to be a dependency association, with the first class being dependent on the second, if both classes are general classes. In this case, the dependency arrow head is added to the interpreted diagram. If one class is an interface class, the interpreted diagram replaces the stroke with a straight line connecting the two classes, creating an interface association. The association is attached to the classes and, if an attached class is moved, the association will move in accordance with the moving class. This recognition algorithm is modeled in a *LADDER* description by the use of contextual shapes as part of the *LADDER* shape description.

Text: Text can be handwritten directly onto the class. In Figure 2-2(a), the ObjectRendered class contains the text desciption "graphics." Note that the text is not recognized, but merely identified as text. It is identified using a combination of properties such as size and location. The text must be small in comparison to the class size. The text must lie inside of or near a class. Figure 2-2(c) shows how the identified text describing the ObjectRendered class remains attached to the correct class when the class is moved. In future work, we intend to perform character recognition on the text.

Although we currently do not recognize text, class, property, and method names can be named using a form. Text can be input into the form using the panel keyboard and a mouse or by typing directly with the keyboard. Figure 2-6 shows a picture of the form inputting information for the Game class. Notice that the information on the form is also updated on the diagram.

Editing

The system is non-modal: users can edit or draw without having to give any explicit advance notification. One editing action is moving classes and associations on the



Game
int score
Vector players
void nextTurn()
void computeWinner()

Name: Game

Properties:
int score
Vector players

Methods:
void nextTurn()
void computeWinner()

[✓ Name] Property Method

Blue: Drawn, Black: Interpreted

Figure 2-6: Class names, property names, and method names can be input into the form using the panel keyboard and a mouse or by typing directly with the keyboard.

screen. The system understands a gesture as a move command rather than as a drawing command based on the user's sketching behavior: Users tend to click and hover over a class when moving it. For example, the system interprets a hover longer than .5 seconds as a move command. The move command is signified to the user by a cursor changing to a gripping hand with which the user can move the class.

The user can delete a class or association by scribbling over the shape. We define class deletion lines as being the horizontal, vertical, and diagonal lines passing through the body of a class. Deletion of an interface or a general class is recognized by checking whether the stroke collection crosses a deletion line of the class more than four times. Deletion of a relationship is recognized by checking whether the collection of strokes crosses the arrow more than four times. More than one object can be deleted with a single deletion command.

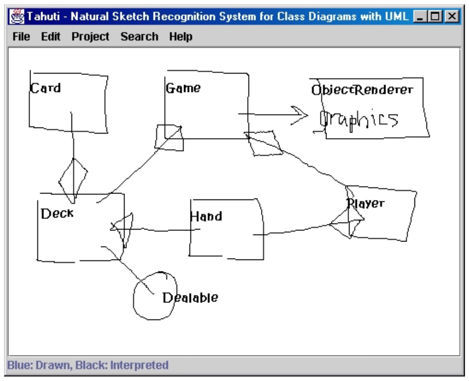
A stroke is recognized as a movement action if the user has clicked and held the cursor over the body of a class or the endpoint of an association with relatively little movement for a period of time greater than a half second. After the stroke is identified as a movement action, the cursor changes to a gripping hand, and any further movement of the cursor will move the object appropriately. Recognition of a stroke as movement of an object must occur during the stroke, rather than after the stroke is completed. In this case, the recognition process is short-circuited, and no other interpretations are attempted.

If an interface or general class is moved, any relationships attached to the class will remain attached, moving in rubber-band format. If a relationship is moved, the endpoint that has been moved will detach from any class that it is currently attached to. Once the relationship is moved and the mouse has been released, the relationship endpoint is examined to see whether it should be reattached to a class or remain unattached. It is possible for a relationship type to change from an aggregation, inheritance, or dependency relationship to an interface relationship, if the arrow is moved from a general class to an interface class, or vice versa.

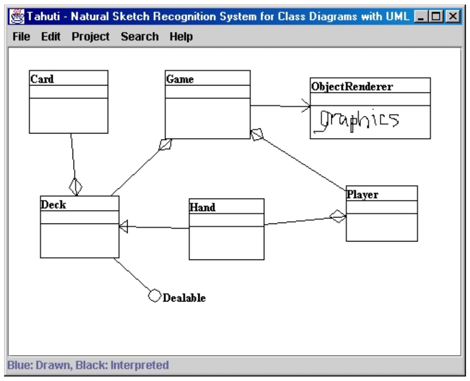
Multi-View System

While sketching, the user can seamlessly switch between two views: the drawn view (Figure 2-7(a)), which displays the users original strokes, or the interpreted view (Figure 2-7(b)), which displays the identified objects. Users can draw and edit in either view. Editing commands operate identically in the two views, with the drawn view allowing users to view and edit their original strokes. When a class is dragged, the strokes of an attached association must be stretched, translated, and rotated in order for it to remain attached and for the strokes to remain faithful to those originally drawn. Figure 2-7(d) shows the results after moving classes in Figure 2-7(b). The drawn view is shown in Figure 2-7(c). Some sketchers become distracted by the sketch recognition process when it replaces their strokes with the interpreted version. The alternate views allow the users to sketch in the manner in which they are more comfortable, thus engendering user-autonomy in sketching.

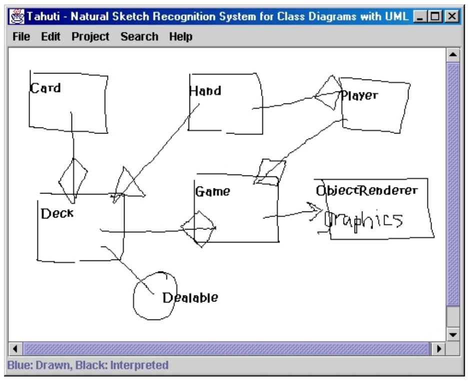
The strokes shown in the drawn view are not the same as those shown in the interpreted view. Several complications arise from this. One complication is that the system now has to keep track of three different sets of stroke data for each stroke drawn. Thus, for each viewable object, the data structure must contain 1) the original


Tahuti - Natural Sketch Recognition System for Class Diagrams with UML
File Edit Project Search Help
Card
Game
ObjectRenderer
Graphics
Deck
Hand
Player
Dealable
Blue: Drawn, Black: Interpreted

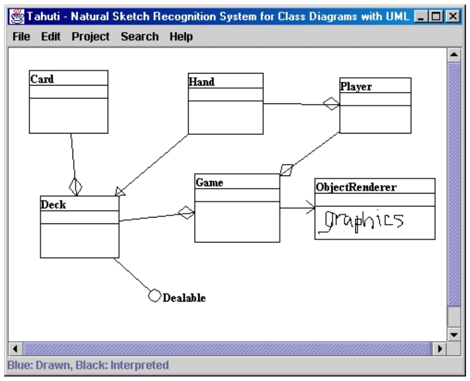
(a) Hand-drawn UML class diagram in Tahuti


Tahuti - Natural Sketch Recognition System for Class Diagrams with UML
File Edit Project Search Help
Card
Game
ObjectRenderer
Graphics
Deck
Hand
Player
Dealable
Blue: Drawn, Black: Interpreted

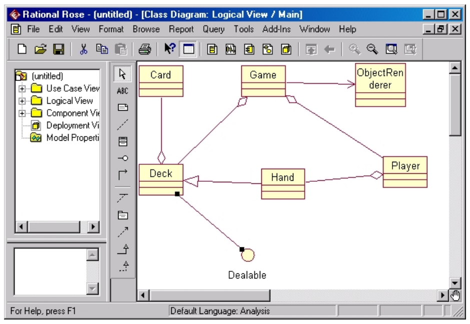
(b) Recognized UML class diagram in Tahuti of Figure 2-2(a)


Tahuti - Natural Sketch Recognition System for Class Diagrams with UML
File Edit Project Search Help
Card
Hand
Player
Game
ObjectRenderer
Graphics
Deck
Dealable
Blue: Drawn, Black: Interpreted

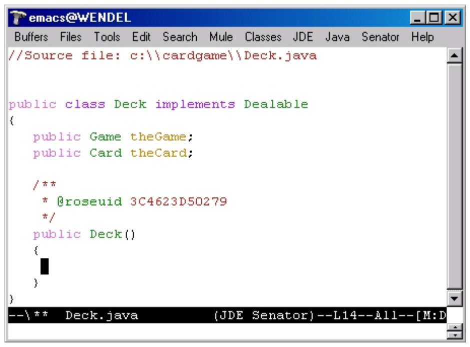
(c) Diagram in Figure 2-2(a) with classes moved


Tahuti - Natural Sketch Recognition System for Class Diagrams with UML
File Edit Project Search Help
Card
Hand
Player
Game
ObjectRenderer
Graphics
Deck
Dealable
Blue: Drawn, Black: Interpreted

(d) Interpreted diagram of Figure 2-2(c)


Rational Rose - (untitled) - [Class Diagram: Logical View / Main]
File Edit View Format Browse Report Query Tools Add-Ins Window Help
(untitled)
Use Case View
Logical View
Component View
Deployment View
Model Properties
Card
Game
ObjectRenderer
Deck
Hand
Player
Dealable
For Help, press F1
Default Language: Analysis

(e) Diagram in Figure 2-2(a) sent to Rational Rose^{TM}


emacs@WENDEL
Buffers Files Tools Edit Search Mule Classes JDE Java Senator Help

```
//Source file: c:\\cardgame\\Deck.java


public class Deck implements Dealable
{
   public Game theGame;
   public Card theCard;

   /**
    * @roseuid 3C4623D50279
    */
   public Deck()
   {

   }
}
--\**  Deck.java          (JDE Senator)--L14--All--[M:D
```

(f) Stub code generated by Rational Rose^{TM}Figure 2-2(e)

Figure 2-7: A hand-drawn UML class diagram in Tahuti, the recognized shapes in Tahuti, the recognized shapes passed off to Rational Rose^{TM}, and the generated stub code produced by Rational Rose^{TM}.

strokes, 2) the uninterpreted strokes (the strokes viewable in the drawn view), and 3) the interpreted strokes (the strokes viewable in the interpreted view). The uninterpreted strokes are not the same as the originally drawn strokes since the object may have been moved, causing the viewable strokes to have been stretched, translated, or rotated. After movement of an object, the uninterpreted strokes are recalculated based on the original strokes, rather than the current uninterpreted strokes to ensure that there is no loss of accuracy.

Since the originally drawn strokes and the viewable strokes in the interpreted view are different, the recognition algorithms must take into account the current view. For example, when deleting an association in the interpreted view, the line or arrow shaft representing the relationship must be crossed. However, in the drawn view, the stretched, scaled, or rotated original strokes representing the relationship must be crossed.

Rational Rose^{TM} Diagrams and Code Generation

The recognized sketches are then automatically sent to Rational Rose^{TM}, a CASE (computer-automated software engineering) tool which generates stub code. Figure 2-7 shows a hand-drawn UML class diagram in Tahuti (Figure 2-7(a)), the recognized shapes in Tahuti (Figure 2-7(b)), the recognized shapes passed off to Rational Rose^{TM}(Figure 2-7(e)), and the generated stub code produced by Rational Rose^{TM}(Figure 2-7(f)). Notice that the Hand class extends the Deck class, and that the Deck class implements the Dealable interface, as specified in the original sketch. This enables the user to take full advantage of the benefits of a CASE tool, such as the ability to auto-generate code stubs, while still retaining the natural feeling of a sketch tool.

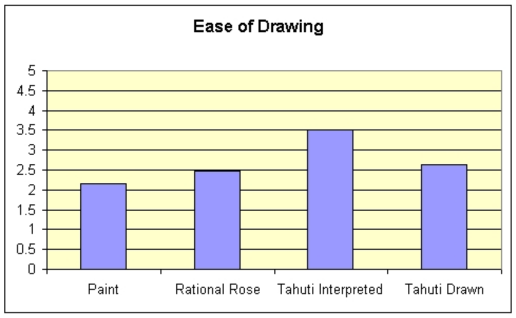


Figure 2-8: Results of user study for ease of drawing. Note that users preferred drawing in the interpreted view of Tahuti.

Experiment

In a preliminary study, six subjects were asked to draw and edit a UML diagram in four different ways: A) using a paint program, B) using Rational Rose^{TM} C) using Tahuti in the interpreted view D) using Tahuti in the drawn view. Subjects were aided in the use of Rational Rose^{TM} if they were unfamiliar with it, but little instruction was given otherwise.

The subjects were asked to rank the four methods on a continuous scale from zero to five (with zero being the hardest, and five being the easiest) both for ease of drawing and for ease of editing. Figure 2-8 displays the results for ease of drawing. Figure 2-9 displays the results for ease of editing. The results reveal that subjects greatly preferred the interpreted sketch interface of Tahuti as compared to the other choices.

At the end of the study, each subject was briefly interviewed. During this time, the subjects stated that they appreciated having the freedom to draw as they would on paper, assisted by the editing intelligence of a computer application. Subjects said that editing was difficult in the paint program because of the large amount of re-sketching required for class movement. Subjects complained that Rational Rose^{TM} was extremely nonintuitive, and that they had difficulty performing the commands that they wished to perform.

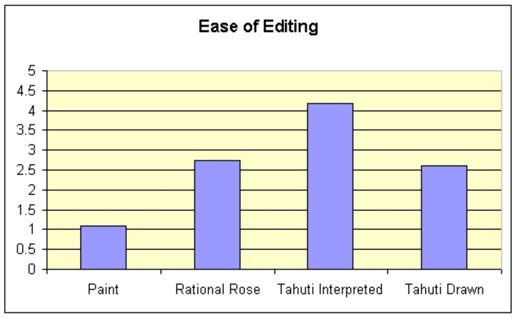


Figure 2-9: Results of user study for ease of editing. Note that users preferred editing in the interpreted view of Tahuti.

Most subjects preferred to work in the interpreted view, rather than the drawn view. The subjects contrasted the domain of UML class diagrams with domains such as Mechanical Engineering and Architecture where a cleaned-up drawing may imply a finished design. They stated that the domain of UML class diagrams is one in which cleaned-up drawings are appreciated since they are created in the design stage, and cleaned-up drawings do not imply solidified ideas. The subjects said that they would prefer to work in the drawn view in a domain such as Mechanical Engineering or Architecture. The subjects predicted that the drawn view would be a valuable feature in any domain, because it would allow them to examine their original strokes, when necessary.

Our experiment suggests that future system enhancements should consider incorporating an ability to recognize multiplicity relationships and modification of recognized objects (e.g., changing a dependency association into an inheritance association by adding a stroke). Further field evidence is, however, necessary before any categorical recommendations can be made in this area. Toward this end, future research should test Tahuti, using larger samples, tighter controls, and varied experimental settings.

Tahuti has been used at Columbia University to teach Object Oriented Programming to a group of 65 students The system was well-received, and it appeared to aid both in the initial program design and in progressive program design, although a formal study was not done. Even simply having a graphical picture of the program seemed to allow the students the ability to maintain a clear picture of the program structure throughout the coding process.

2.3.2 Indexed Software Meetings to Aid in Design Rationale Capture

These sketch recognition systems often prove themselves useful in many unexpected ways. Tahuti was used to automatically index video documentation of software design meetings [105] in an attempt to provide easy access to design rationale.

Defining Design Rationale

Design rationale has been defined in a variety of ways, but all definitions agree that it attempts to determine the *why* behind the design [153] [164] [165]. Design rationale is the externalization and documentation of the reasons behind design decisions, including the design's artifact features. We have chosen the following definition that has been borrowed from Moran and Carroll: Documentation of (a) the reasons for the design of an artifact, (b) the stages or steps of the design process, (c) the history of the design and its context. Louridas and Loucopoulos claim that the design rationale research field includes all research pertaining to the capture, recording, documentation, and effective use of rationale in the development processes. The researchers state that a complete record, which they define to be a video of the entire development process, combined with any materials used and produced, could, in theory, be used to determine the rationale behind the decisions that have been taken. However, they claim that this unformatted data would be unwieldy to search through. Thus, design rationale research has generally encouraged the structuring of design to provide a proposed formalism, using a small set of concepts that are appropriate for representing the deliberations taking place.

A considerable body of effort has been devoted to capturing and indexing design rationale. One part of design rationale is documentation of the design history [153] [164] [165]. While videotaping a design session can capture the design history, retrieval may require watching the entire video. Retrieval can be made simpler by structuring the design process, but this can hold back fast-flowing design meetings [198]. There is an apparent tension between the simplicity of design rationale capture and effectiveness of design rationale retrieval [197]. We hope to bridge this gap by allowing designers to design as they would naturally, and, at the same time, supply them with the tools that understand those designs and allow the designer to use this understanding to help in retrieving appropriate moments of a design meeting.

Collaborative software design meetings often involve the creation of UML software diagrams to design object-oriented software tools by sketching UML-type designs on a white-board. At MIT, this includes building new agent-based technologies for the Intelligent Room. Figure 2-10 shows people designing agents in the Intelligent Room. Traditionally, when new components need to be added to the Intelligent Room's software, a small number of designers gather in the Room and sketch the new design on the whiteboards, while discussing their decisions. What gets recorded after those sessions is the final design and the explanation of the mechanisms employed. What gets omitted are the reasons why those particular solutions got employed.

In response, we have created a system that allows software designers to design agents naturally, using sketch information gained from Tahuti while interpreting software diagrams. The designers can draw UML-type free-hand sketches on a whiteboard, using an electronic marker whose "strokes" are digital ink marks that are projected onto the board, rather than drawn on it. These sketches are recorded and interpreted in real-time to aid in the later retrieval of design history. The system allows the users to design as they would naturally, requiring only that they learn the UML syntax. Information extracted from the diagrams can be used by the system to generate stub code, reducing some of the routine parts of the programming process. The recognition also allows us to flag, label, and timestamp events as they occur,

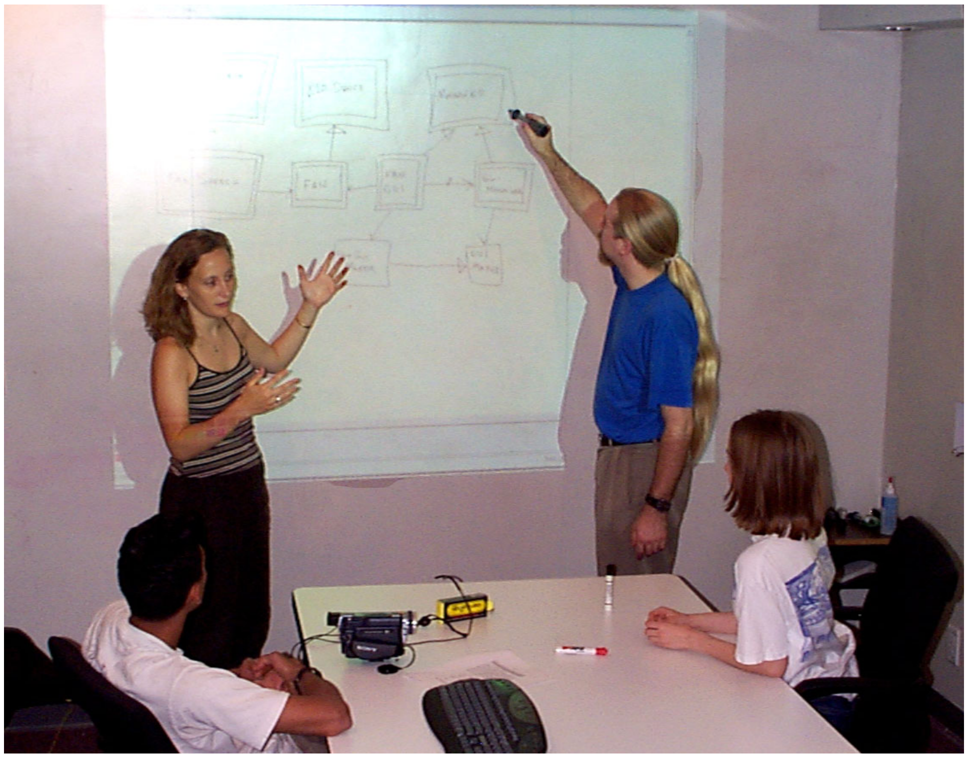


Figure 2-10: People designing agent software in the MIT Intelligent Room.

facilitating retrieval of particular moments of the design history.

Research has been done on indexing audio-visual material [39]. Researchers have attempted to label the video with salient features within the video itself, focusing on the recognition and description of color, texture, shape, spatial location, regions of interest, facial characteristics, and, specifically for motion materials, video segmentation, extraction of representative key frames, scene change detection, extraction of specific objects, and audio keywords. While not much research has been done using sketch recognition to label and index a particular moment in video, a considerable body of work has been done using sketch recognition to find a particular moment in a pre-indexed video [133] [47] [123]. Research has also been used to use sketches to search for static images [31] [14] [47] [133] [61].

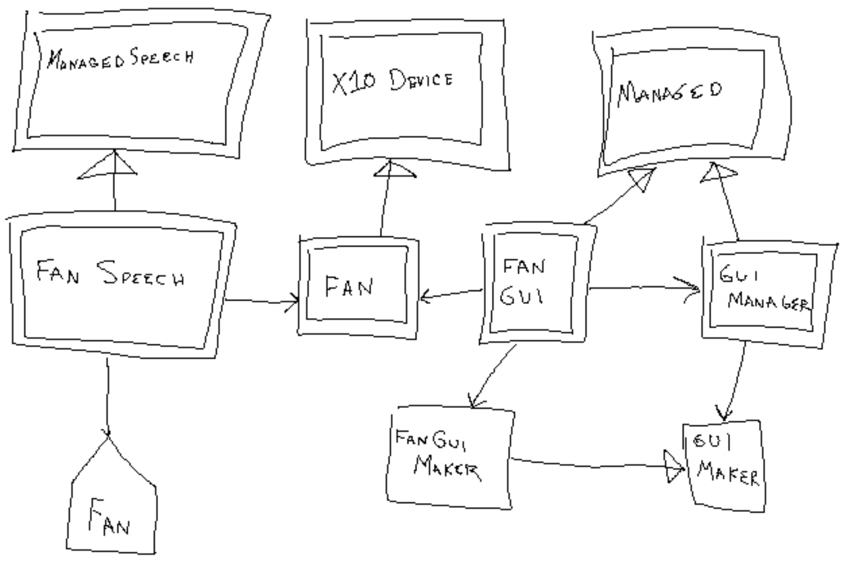


Figure 2-11: A software sketch of a fan agent.

Implementation

UML diagrams have been found to lack simple ways to describe agent-based technologies [170]. We added several symbols for agent-design since many of the applications created in the Intelligent Room [109] of the MIT AI Lab are agent-based. Figure 2-11 shows a software sketch of a fan agent. The recognizable shapes in Tahuti include the components of UML class diagrams, as well as agents (indicated by a double-edged rectangle) and speech grammars (indicated by a triangle) [105]. Bergenti and Poggi [28] have created a CAD system to input UML diagrams for agent-based systems; however, their system requires designers to enter their diagrams using a rigid CAD interface rather than allowing designers to sketch as they would naturally.

We implemented the software meeting indexer as a Metaglue agent since the Metaglue agent architecture provides support for multi-modal interactions through speech, gesture, and graphical user interfaces[48] in the MIT AI Lab's Intelligent Room [109]. The Metaglue agent architecture also provides mechanisms for resource

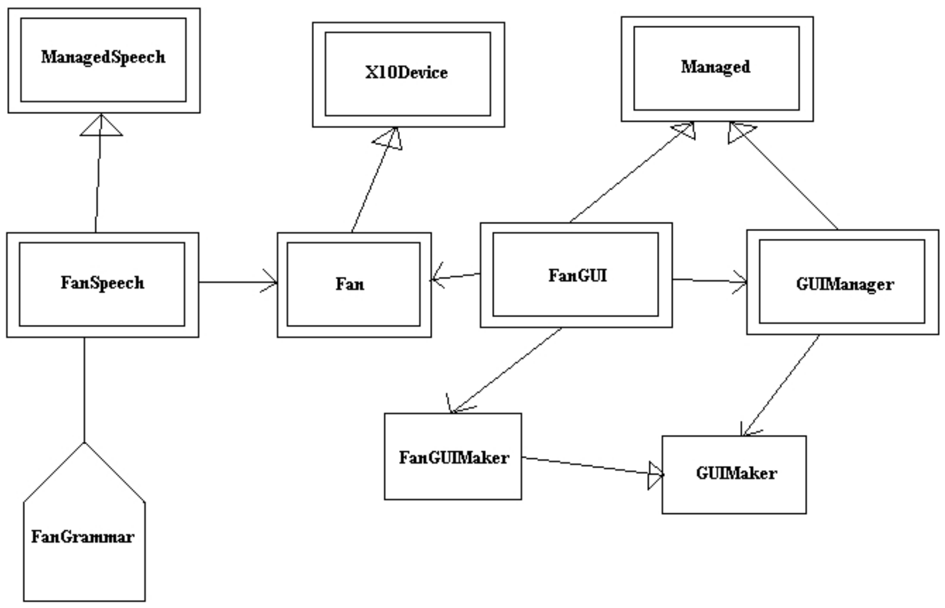


Figure 2-12: The interpretation of Figure 2-11

discovery and management, which allows us to use available video agents or screen capture agents in a Metaglue supported room [84]. The Design Meeting Agent extends the Meeting Management System [172] for capture of non-design information such as the structure of the design meeting [178]. It initializes the Tahuti Agent, which controls the sketch recognition and the timestamping of significant events. It also controls the video and screen capturing agents.

To capture the design meeting history, the Design Meeting Agent requests available audio, video, and screen capture services from the environment and uses them to capture the entire design meeting. However, finding a particular moment of the design history video and audio records can be cumbersome without a proper indexing scheme. To detect, index, and timestamp significant events in the design process, the Tahuti Agent, also started by the Design Meeting Agent, records, recognizes, and understands the UML-type sketches drawn during the meeting. These timestamps can be mapped to particular moments in the captured video and audio, aiding in the retrieval of the captured information. Metaglue, a multi-agent system, provides the computational glue necessary to bind the distributed components of the system together. It also provides necessary tools for seamless multi-modal interaction between the varied agents and the users.

Design Meeting Manager: The Design Meeting Manager extends our earlier Meeting Manager [172]. At startup, it is responsible for obtaining resources necessary for running a basic meeting (a display for keeping track of the agenda, issues, commitments, etc.) and for starting Tahuti, the sketch recognition part of the system. It is also responsible for negotiating with the environment the use of available audio, video, and screen-capture devices. During the meeting, the Design Meeting Manager will keep track of the organizational aspects of the meeting, such as moving through and augmenting the meeting agenda. It also provides a means for querying previous meetings.

Speech Interfaces: Both the meeting manager and the Tahuti Agent can interact with users through speech. The grammar of the Design Meeting Manager contains vocabulary for controlling the flow of the meeting and for querying previous meetings. Tahuti's speech interface allows users to interact with the sketch (e.g. provide feedback in case of misrecognition of drawn shapes) and to query earlier designs (e.g. "What where we talking about when we added this class?").

Meeting Capture Services: There are a number of agents deployed in Metaglue-enabled spaces that can, depending on the availability of hardware and software resources, provide capture services to the Design Meeting Manager. All or some of the following may be present: the video capture agent, audio capture agent, and/or screen display capture agent (using Camtasia^{TM}). Ideally, all of those capabilities would be present.

Tahuti Agent: The Tahuti Agent recognizes UML class diagrams and time-stamps events as they occur; these timestamps are used to index the video of the design meeting. Figure 2-12 shows the interpretation of the diagram in Figure 2-11. The symbols include those described above and two additional symbols: a double-

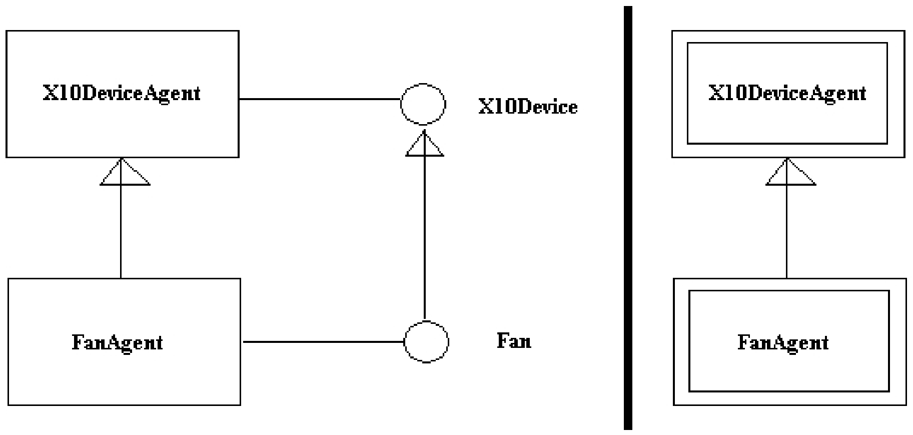


Figure 2-13: Each agent implements an interface with a corresponding name. If an agent inherits from another agent, so do the interfaces of the agents (left figure). In these sketches, the interfaces are assumed and not drawn (right figure).

edged rectangle to denote agents and a triangle (shown in the interpreted view as a triangle with an extended bottom, or a pentagon, to fit more text) to denote grammars for speech-enabled agents. These symbols simplify our diagrams by providing certain syntactic shortcuts. In reality, an agent's implementation is always accompanied by an interface, and the inheritance structure of interfaces usually parallels that of agents (Figure 2-13). In our sketches, we omit the interfaces. The interactions among agents involve a complex pattern of proxy objects and helper classes, which we also omit in our sketches (Figure 2-14). Finally, reliance on a grammar always implies use of a special proxy agent and interaction with Metaglue's speech facilities (Figure 2-15).

Timestamping of Significant Events: All events in the design process are recorded, labeled, and time stamped. A significant event is defined as the addition or deletion of a general class, interface class, agent, grammar, or relationship. Less significant events include the movement of a class, agent, grammar, or association, or the addition, deletion, or editing of text, such as class, method, or property names. During the development process, the designer may also mark a particular event as particularly significant. The designer can then later ask questions such as, "What was the discussion when this class was created?" and the system can show the appropriate

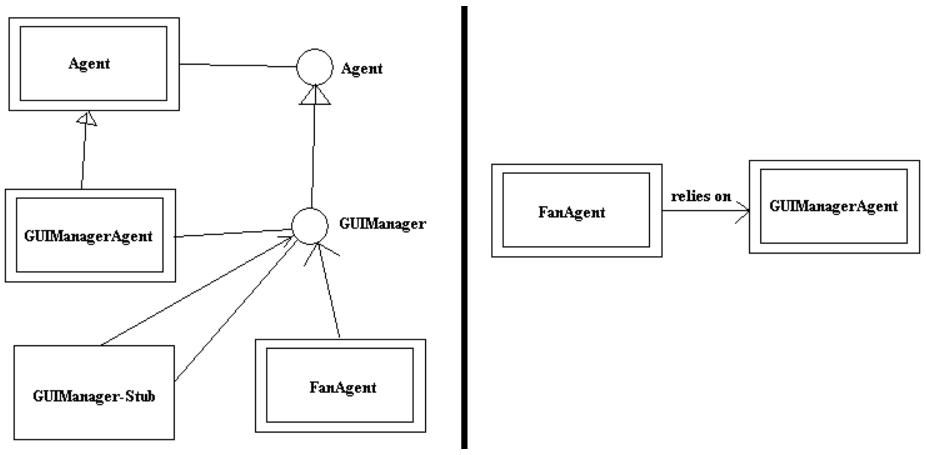


Figure 2-14: The figure on the left displays the actual interaction between the two classes. The figure on the right displays the abstraction for “relies on.”

section of video and screen shots.

Design History: Designers may also want to ask the more general question “How did we design this system?” We would like to present to the designer a visual description of how the scene evolved. We do not want to show the designer all of the significant events. Rather, we want to select a small number of snapshots that, when combined together, can best display the evolution of the design. We want to select the most significant events to show to the user and show the most revealing snapshot related to those significant events. Significant events are each given a rank, represented as a floating-point number. The number before the decimal place is set according to the type of event. For instance, creation of an Agent is given the highest rank of all sketched objects, with a rank of 10. The table below lists the initial rank of each of the possible events. While the numbers themselves are slightly arbitrary, what is important is the relative ordering of the events.

- Final Design: 12
- User Marked Significant Event: 11
- Creation of an Existing Agent: 10

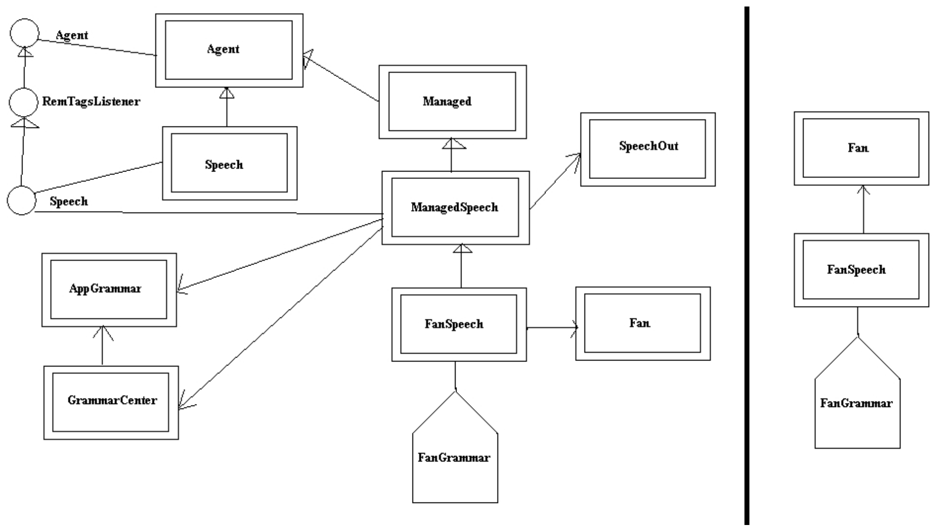


Figure 2-15: Left: the speech grammar along with connected classes. Right: the simpler version with the grammar symbol implying the relationships and agents on the left.

- Creation of an Existing General or Interface Class: 9
- Creation of an Existing Speech Grammar: 8
- Creation of an Existing Association: 7
- Creation of an Existing Unrecognized Stroke: 6
- Text Update: 5
- Movement: 4
- Creation of a Deleted Object: 3
- Deletion of an Object: 2
- Undo/Redo: 1

The logic behind the initial ranking is as follows: The final event is always ranked the highest. The designer selected significant events outrank computer selected significant events. Creation of viewable objects is considered a more significant event than the updating or movement of that object. Creations of objects that no longer exist in the final version are considered to be much less significant than those that remained throughout the entire process.

Within a particular category (e.g., looking only at the Creation of Agent Events), events are again ranked as more or less significant. Events that affect more objects have a higher ranking. The fraction part of the floating-point number is used to assign further ranking. Events specifying the creation of agents, classes, and grammars are further differentiated by the number of associations attached to them. For instance, an agent connected by an association to 4 classes would have a rank of 10.04 (since the number of associations is divided by 100).

A designer may want to see screenshots of the five most significant events to review a brief history of the design process. When the most significant events are chosen, the screenshot associated with the event is not the snapshot of the time of the occurrence of that significant event, but, rather, the snapshot of the moment before the next significant event. The next significant event is defined as the next event greater than or equal to the lowest ranking in the listing of the most significant event. This allows any smaller additions, such as text or movement, to be included in the snapshot. Figure 2-16 shows the ranking of each of the significant events of the diagram. Figure 2-17, Figure 2-18, and Figure 2-19 show the three most significant events of a diagram.

2.4 General Recognition Systems

The work in this document proposes building a general purpose sketch recognition that makes building a sketch system for a particular domain easy and fast [8] [107].

Quill [150] [152] [151] is a tool for designing gestures and gesture sets for pen-based user interfaces that allows designers of a gesture recognition system to sketch the gestures to be recognized. It exposes some information about the recognizer,

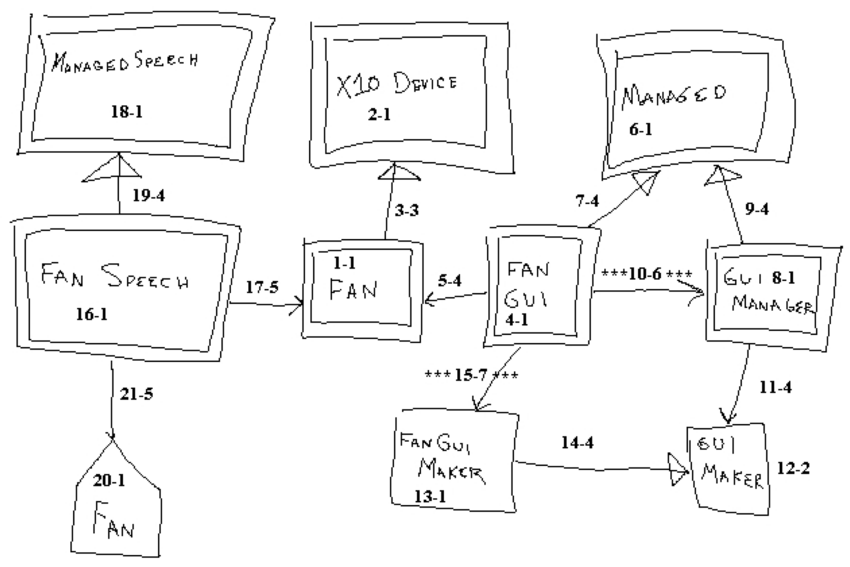


Figure 2-16: Each class, agent, association, and grammar is marked with a number specifying its order drawn, followed by its ranking. Note that the two with the highest rankings are marked with stars.

and provides active advice about how well the gestures will be recognized by the computer, and how well they will be learned and remembered by people. Quill uses a feature-based Graffiti^{TM}-type recognition that focus on the way the shape is drawn (e.g., the number of strokes, as well as stroke speed, curvature, order, and direction, etc.). In order for their strokes to be recognized, users of this system must sketch each gesture in the same way as that of the designer who trained the system, including stroke direction, order, and speed, rather than recognizing the drawn object by its geometrical shape. Our focus is on removing as many sketching restrictions as possible so as to provide a more natural sketching medium in which sketchers are able to draw shapes by using their own individual natural style. 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 [95] [93] [92] created a domain-independent

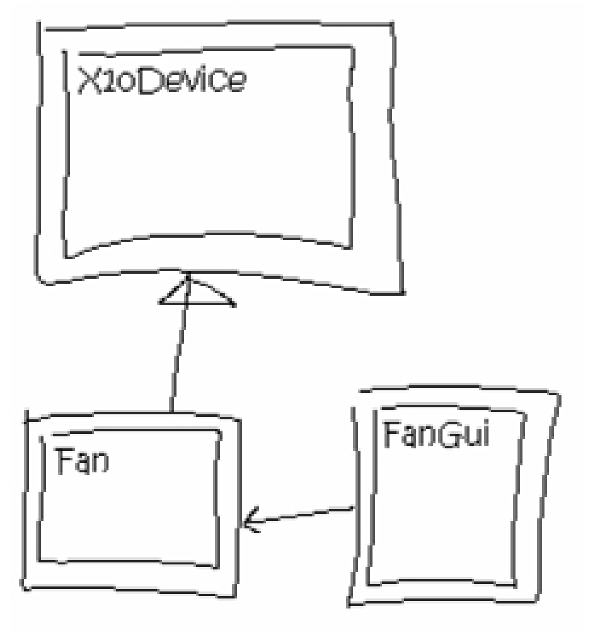


Figure 2-17: Significant Design Event 1, the screen shot significant event 4 (which includes significant event 5)

sketch recognition system that allowed users to define shapes by drawing them. The shape is then described by the shapes from which it is composed and the constraints between them. The ECNP does not handle ambiguity, nor can it describe non-shape information, such as editing behavior. The low level recognizers have stroke order and direction requirements and need to be trained a multitude of times, rather than recognizing based on shape. The project does not appear to have been pursued beyond its initial stage. For instance, the system built did not allow users to do anything beyond defining new shapes. It did not provide any mechanism for defining how a shape was to be displayed or edited once recognized. Neither did it provide any method for altering the shape definitions by hand if the description it generated was too specific or too general. Our system improves on this system in that we allow users to specify all parts of a sketch domain, including how shapes are displayed and edited in the domain.

Jacob [122] has created a software model and language for describing and programming fine-grained aspects of interaction in a non-WIMP user interface, such as

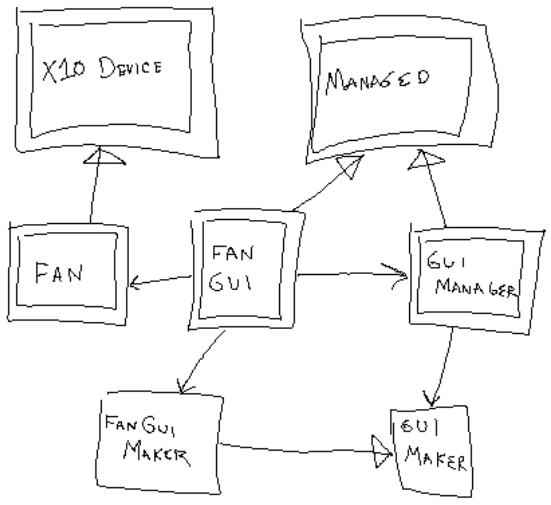


Figure 2-18: Significant Design Event 2, after significant event 8 (which includes significant event 15)

a virtual environment. The language is still close to the signal-processing level and requires users to do a significant amount of coding to define new interactions, and, in the domain of sketching, it does not provide a significant improvement to coding the domain-dependent recognition system from scratch.

Lank [144] has created a framework for developing new sketch systems. The system simplifies application development, but the user still has to write and 5,500 lines of code to produce a UML sketch recognition system, and have an expertise in sketch recognition.

Systems have been built to automatically build vision recognition system. Ikeuchi and Kanade have worked on systems to automatically compile object and sensor models in to a visual recognition strategy for recognizing and locating and object in three-dimensional space from visual data [121]. Their object model is (similar to the one described in this document) is also based on geometric properties. As light and sensor characteristics also play a role in vision, they also model photometric and sensor properties.

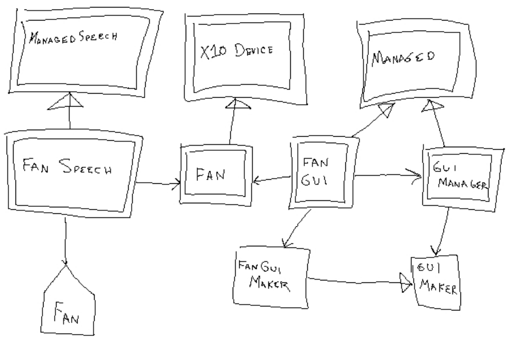


Figure 2-19: Significant Design Event 3, the final diagram

Several frameworks have been developed for simplifying the development of multi-modal user interfaces [73].

2.5 Handling Ambiguity in Recognition

GUILD handles ambiguity by sending all possible interpretations to the recognition system. Other systems handle ambiguities in other ways. SketchRead [12] handles ambiguity by asking the system for another interpretation when the first interpretation does not work. Mankoff et. al. [158] have developed a user-interface framework that handles ambiguity through the use of mediators.

2.6 Previous Shape Languages

Shape definition languages, such as shape grammars, have been around for a long time [204]. Shape grammars are studied widely within the field of architecture, and many systems are continuing to be built using shape grammars [87]. However, shape grammars have 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 [83]. These shape definition languages are not intended for use with an on-line system and do not provide ways to specify how to display or edit a shape. 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.

GUILD is based on template filling of a shape's structural description. These structural descriptions are often represented in relational graphs. Lee performed recognition using attribute relational graphs [146]. Lee's attribute language differs from ours in that ours is more topological or geometrical, whereas theirs is more quantitative, requiring specific details of the shape's position. Keating and Mason also performed 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 [136]. 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 [43].

Within the field of sketch recognition, there have been other attempts to create languages for sketch recognition. Mahoney [157] uses a language to model and recognize stick figures.The language currently is not hierarchical, making large objects cumbersome to describe. Saund developed a symbolic method for constructing higher level primitive geometric shapes, such as curved contours, corners, and bars. Bimber, Encarnacao, and Stork created a multi-layer architecture for sketch recognition [30] of three-dimensional sketches. Their system recognizes objects created by multiple strokes with the use of a simple BNF-grammar to define the sketch language. However, due to the nature of their domain, the system requires users to learn drawing commands before using the system, rather than giving users the freedom to draw as they would naturally. This language allows a programmer to specify only shape information, and it lacks the ability to specify other helpful domain information such as stroke order or direction and editing behavior, display, or shape interaction information.

Caetano et al. [42] use Fuzzy Relational Grammars to describe shape recognition. This allows them to combine fuzzy logic and spacial relation syntax in a single unified formalism [129] [76] [77]. In their grammar, they assign attributes to objects, such as "VERY THIN", which are quantified using fuzzy grammar. An example production rule for lines is: "IF Stroke IS VERY THIN THEN Shape IS Line". These languages lack the ability to describe editing, display, or shape group information.

Myers et. al. [167] designed a large system and language for designing user interfaces. The language allows the user to define graphical objects, interactions, editing operations, and gestures to be recognized. The system, however, uses the Rubine engine to define and recognize gestures, thus each shape must be drawn in a single stroke and style.

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 the user to specify concrete values, using either a range or a gaussian [196]. We find it difficult to describe some shapes using this technique as the language requires the provision of 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 < SHAPE2.WEST).RANGE(0, 100)) Shilman's work also lacks the ability to describe editing and display.

Egenhofer [61] has used a geometric-based recognition system similar to our to search for images on the web. The user's sketch is processed geometrically, and each item in the drawing is compared to the other items in the drawing, comparing the relative cardinal directions (NW, N, NE, W, 0, E, SW, S, SE), topology (disjoint, meet, overlap, contains, covers, inside, coveredBy, equal), and a metric refinement to determine the amount of area or border intersection. The system then retrieves the pictures in the image library (which are all preprocessed in a similar way) that best match the drawn sketch according to those measures. They have implemented a constraint relaxation technique to find imperfect matches.

2.7 Human Perception

The *LADDER* constraints are based on geometric perceptual principles. Veselova has made a significant research progress in this area by automatically generating shape descriptions using perceptual rules [208] [209] [210]. David has developed a method for recognizing deformable shapes based on perception [57]. Sarkar develops metrics for quantifying Gestalt properties of similarity between arcs and lines [186].

Saund et. al. has made significant progress in using perception to aid in recognition, editing, and object grouping [190] [189] [187] [188].

This research attempts to use some of the ground work done using perception in sketch recognition to automatically generate recognizers based on similar principles.

2.8 Sketch Beautification

Sketch beautification and display is an important part of the sketch recognition user interface. Not only does it clean up the diagram, it also acts to notify the user that an object has been identified [119]. It may also be used to supply additional helpful information to the user during the design process (as in the UML sketch system described in Sections 6.2 and B.2). Arvo and Novin discuss integrating beautification into the drawing process, by morphing the drawn shape into the beautified shape while the user is drawing it [19] [20].

2.9 Recognition Using Indexing

Part of this document describes a recognition algorithm that uses indexing to speed-up recognition search and retrieval. Indexing has been used in this form in many other fields. For example, it has been used in the closely related field of vision object recognition [203] [202] [16] [139] [26] [44] [53] [173] [180]. Indexing sketches has also been done to search for images [31] [133]. This researcher has previously used recognized sketches to index software design meetings as described above [105].

The recognition algorithm described in this thesis appears to be the first use of the idea in support of unconstrained sketch recognition. By breaking the strokes down to line, curve, and ellipse segments, we were able to define shapes in geometric terms, then perform indexing on these terms, a luxury not afforded when indexing photo images or non-geometric sketches (such as an artist's sketch).

2.10 Geometric Constraint Solvers

Much work has been done on constraint solvers, in general. The University of Washington created the Cassowary linear geometric constraint solver [21]. The *LADDER* vocabulary includes many nonlinear constraints, such as EQUALLENGTH. Thus, this research includes the building of a nonlinear constraint solver.

Stahovich used a constraint solver on mechanical engineering constraints to generate geometries from constraints [200]. Rather than acting as a shape generation system given constraints (as the system described in this thesis does), the SketchIt sketch recognition system interprets what the user meant to draw and cleans up the diagram to ensure that the mechanical engineering simulations works as intended.

2.11 Learning Shape Descriptions

Long has created a multi-domain recognition system in which the developer is able to specify the shapes to be recognized in a domain by drawing them [150]. The system helps the developer to debug shapes by specifying which shapes are similar and may be confused with other shapes, thus, causing recognition problems. Long is solving a different problem from that discussed here in that Quill considers ambiguity to be a bug, and requires the developer to change the way things are drawn.

Gross et. al. [94] have created a multi-domain recognition system, but they have no methods for debugging the shapes specified within them.

Lu et. al. [154] have developed a system for learning shape concepts by recognizing shapes based on training data. Several attributes can be used to describe a shape, and the thresholds for these attributes for each shape concept are computed based on the teacher-provided training data. Sezgin has developed a technique to use training data to predict user-dependent temporal sequences to aid in recognition [193].

2.12 Active Learning and Near-miss Generation

Machine learning is a subfield of artificial intelligence concerned with the creation of techniques that help a computer to learn, usually from a set of input examples [162]. While there are some exceptions (such as reinforcement learning), on a whole, these input examples are either labeled (as in supervised learning), unlabeled (as in unsupervised learning), or some combination (semi-supervised learning) [128] [33] [46].

In many domains, including sketch recognition, there are not huge amounts of labeled training data sets available to the developer for each domain. Thus, since labeling examples is costly, near-miss examples can be skillfully chosen to have quick and effective learning of a shape concept. Winston argues that in concept learning, the learner usually holds only one hypothesis at a time and that the learner can learn the correct model most effectively if the example differs from the current hypothesis in only on aspect to that the learner cannot make the wrong choice and can slowly be guided to the correct model of the concept [214].

Traditionally, examples in supervised learning are generated and labeled by the teacher, although the examples can also be generated by learner and labeled by the teacher. Active learning describes the process of instruction through learner generated examples [15] [22]. This corresponds to the a type of classroom instruction of the same name where instructors encourage students to actively engage in the learning process, with the expectation that these students will be more able to recall information later [36] [40].

If the computer is better able to determine which examples are necessary to aid learning, active learning can help the learner more effectively or quickly learn a concept by skillful selection of near-miss examples. This thesis discusses a supervised learning technique in which shape concepts are learned from labeled positive and negative examples generated either by the teacher or the learner.

Effective active learning using near-miss examples requires the ability to alter only one aspect of an example at a time [51] [15] [130]. Mitchell has invented version spaces, a concept learning algorithm that can keep track of the possible hypotheses given a set of labeled examples [163]. However, the version spaces algorithm does not handle disjunction nor interrelated constraints, which are prevalent in sketch recognition shape concepts, thus this thesis discusses a modification of his algorithm developed by the researcher.

Chapter 3

FLUID: FaciLitating UI Development

3.1 Motivation

Sketch interfaces are valuable additions to natural human-computer interactions, but developing sketch interfaces requires substantial effort. They can be quite time-consuming to build if they are to handle the intricacies of each domain. Moreover, the sketch interface developer has to be an expert in sketch recognition at the signal level.

This research argues that to make a user-friendly system, the designer of the system should be an expert in building user interfaces, and/or an expert in the domain itself. This person does not necessarily have to be an expert in sketch recognition at the signal processing level.

This research aims at enabling the user interface or domain expert to be able to design and develop a sketch interface without needing to understand and code the signal processing details. To this end, this research includes the development of a framework that gives the sketch interface developer the power to implement recognition changes, herself, without having recognition expertise. By abstracting away the sketch recognition task from building sketch recognition systems, the developer no longer needs to address sketch recognition when building new sketch interfaces. As a result, we been able to build several sketch interfaces in less time than it has taken previously.

3.2 Solution

To date, most sketch recognition systems have been domain-specific, with the recognition details of the domain hard-coded into the system, making the development of a new sketch system a long and difficult process. Rather than build a separate recognition system for each domain, this researcher proposes the *FLUID* framework, consisting of a single multi-domain recognition system that can be customized for each domain.

To build a sketch system, a user interface developer would only need to describe the domain-specific information. The programmer would not have to write sketch recognition code, and could, instead, focus on other details of the user interface.

3.3 Domain-specific Information

When constructing a user interface, the domain-specific information is able to be obtained by asking the following questions:

- What are the observable states to be recognized?
- How are these states to be recognized?
- What should happen when these states are recognized?
- How can we modify these states?

In sketch recognition user interfaces, the domain-specific information is obtained by asking these questions:

- What shapes are in the domain?
- How is each shape recognized?
- What should happen after each shape is recognized?
- How can the shapes be edited?

Many domain-specific events can occur after a shape is recognized, but what is common in most domains is a change in display. Sketchers often prefer to have a change in display to confirm that their object was recognized correctly, as a form of positive feedback. Changes in display may also function as a way to remove clutter from the diagram. For example, the system may replace several messy hand-drawn strokes with a small representative image. A change in display may vary from a simple change in color, a moderate change of cleaning up the drawn strokes (e.g., straightening lines, joining edges), to a more drastic change of replacing the strokes with an entirely different image. Because display changes are so popular and so common to most domains, we have included them in the language, as described in Chapter 4. But, because other recognition effects may be desired, we have also included an API, which is described in Chapter 5.

This framework not only defines which shapes are in the domain and how they are to be recognized in the domain, it also recognizes the importance of editing and display in creating an effective user interface. Developers of different domains may want the same shape to be displayed differently: Compare a brainstorming sketch interface that develops a web page layout such as DENIM [147][148], in which shapes may be left unrecognized, to a UML class diagram sketch interface, where sketchers may want to replace box-shaped classes with an index card-like image such as in Tahuti [99][96]. Developers of different domains may want the same shape edited in different ways: Compare a cross in a mechanical engineering sketch interface, such as Assist [6], which represents a drawn symbol for an anchor, to a cross in UML class diagrams, which represents a symbol for deletion and acts as an editing gesture to delete whatever shape it was drawn on top of, as occurs in Tahuti [99].

3.4 Framework

To build a sketch interface for a new domain, a user interface developer writes a domain description defining the shapes in the domain and how they are recognized, displayed, and edited. This domain description is then automatically translated into shape recognizers, editing recognizers, and shape exhibitors. These are used with the customizable, domain-independent recognition system to create 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, which has been using this approach with some success [217] [54] [211]. We analogize sketching with speech, where interfaces are developed by writing a speech grammar, which is then used with a domain-independent speech recognition system.

In our framework, we transform a domain description into a recognizer of hand-drawn shapes [107] [101] [98] [103]. This is analogous to work done on compiler-compilers, in particular, visual language compiler-compilers [54]. 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 an arrangement of icons is syntactically valid. One main difference between that work and ours is that the visual language compiler-compiler deals with the arrangement of completed icons, where our work includes three additional levels of reasoning: first dealing with how strokes form primitive shapes (such as lines and ellipses), then how these primitive shapes form higher-level shapes or icons, and finally, how the higher-level shapes interact to form more complicated shapes or less formal shape groups.

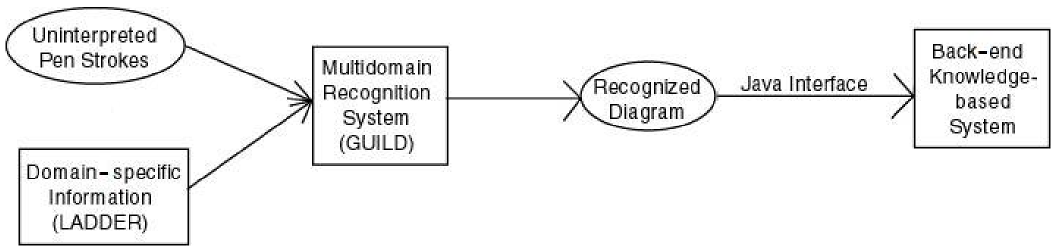

Uninterpreted Pen Strokes
Domain-specific Information (LADDER)
Multidomain Recognition System (GUILD)
Recognized Diagram
Java Interface
Back-end Knowledge-based System

Figure 3-1: The *FLUID* framework.

Figure 3-1 shows the *FLUID* framework. To build a new sketch interface:

1. A developer writes a *LADDER* domain description describing information specific to each domain, including: what shapes are included in the domain, and how each shape is to be recognized, displayed (providing feedback to the user), and edited.

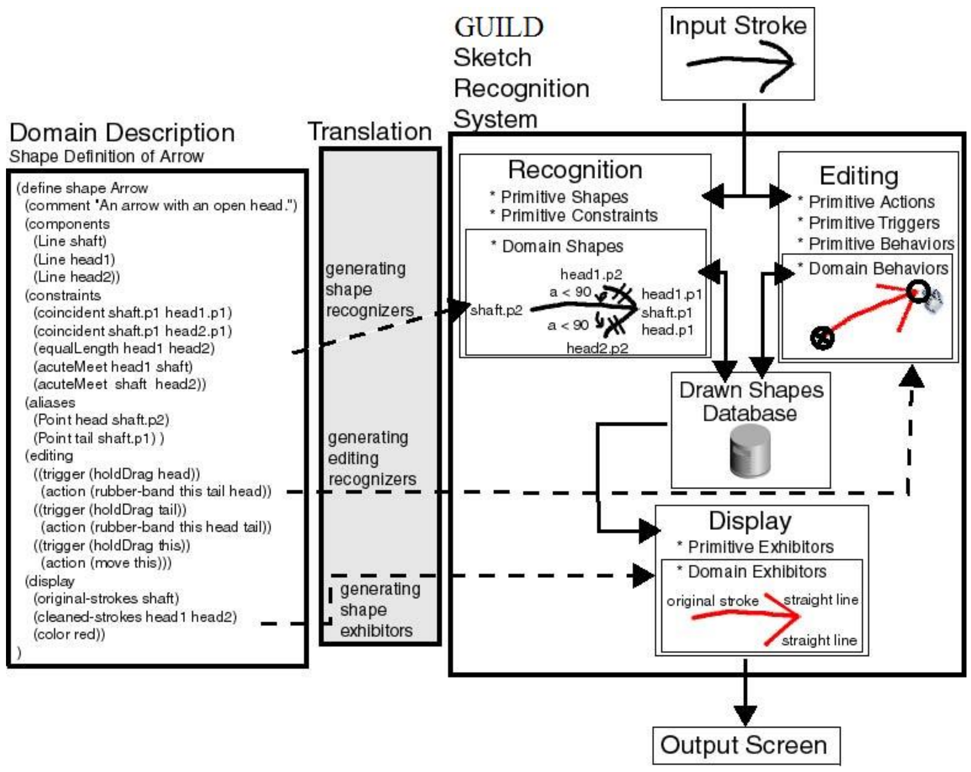
2. The developer will write a Java file that functions as an interface between the existing back-end knowledge system (e.g., a CAD tool) and the recognition system, based on a supplied API.

3. The *GUILD* (see Chapter 5) customizable recognition system translates the *LADDER* domain description into shape recognizers, editors, and exhibitors (see Figure 3-2).

4. The *GUILD* customizable recognition system now functions as a domain-specific sketch interface that recognizes, displays, and allows editing of the shapes in the domain, as specified in the domain description. It also connects via the Java interface (listed in Step 2) to an existing back-end system.

3.5 Implementation

We have implemented the *FLUID* framework by building 1) *LADDER* [100], a symbolic language to describe domain-specific information, including how shapes are



```
(define shape Arrow
  (comment "An arrow with an open head.")
  (components
    (Line shaft)
    (Line head1)
    (Line head2))
  (constraints
    (coincident shaft.p1 head1.p1)
    (coincident shaft.p1 head2.p1)
    (equalLength head1 head2)
    (acuteMeet head1 shaft)
    (acuteMeet shaft head2))
  (aliases
    (Point head shaft.p2)
    (Point tail shaft.p1) )
  (editing
    ((trigger (holdDrag head))
     (action (rubber-band this tail head))
    ((trigger (holdDrag tail))
     (action (rubber-band this head tail))
    ((trigger (holdDrag this))
     (action (move this)))
  (display
    (original-strokes shaft)
    (cleaned-strokes head1 head2)
    (color red))
)
```

Figure 3-2: GUILD: The domain description is translated into recognizers, exhibitors, and editors for each shape in the domain.

drawn, displayed, and edited in a domain; and 2) *GUILD* a customizable, multi-domain recognition system that transforms a *LADDER* domain description into recognizers, editors, and exhibitors to produce a domain-specific user interface [101].

3.5.1 *LADDER* Domain Description

The *LADDER* domain description lists the shapes in a domain and how each shape is recognized, displayed, and edited. Recognition of the shape is described primarily in terms of the shape's geometry, allowing shapes to be recognized no matter the number, order, or direction of the strokes used. In our system, descriptions should account for all of the allowable conceptual variations, but should not include signal noise, which is handled by the recognizer. (See Section 5.1.1, which defines signal error and conceptual error, and describes how they are handled by the recognition system.)

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.

The left box of Figure 3-2 gives an example of an Arrow shape definition. The shape recognition part of a shape description can be created by hand or generated automatically using techniques listed later in this thesis. (See Chapters 7-11.) The components and the constraints define what the shape looks like and are transformed into shape recognizers. The display section, which specifies how a shape is to be displayed when recognized, is transformed into shape exhibitors. The editing section, which specifies the editing behaviors that can be performed on the recognized shape, is transformed into editing recognizers.

3.5.2 Recognition

The third box of Figure 3-2 shows the generated domain-specific sketch recognition system. *GUILD* contains shape recognizers, editing recognizers, and shape exhibitors for the primitive shapes (line, ellipse, curve, arc, and point). The arrows show how the shape recognizers, editors, and exhibitors are generated from the *LADDER* domain description.

Recognition is carried out as a series of bottom-up opportunistic data-driven triggers in response to pen strokes. As each stroke is drawn, the system determines whether the stroke is an editing trigger. If not, it is taken to be a drawing gesture, and the stroke is analyzed as a collection of primitive shapes, using the primitive recognizers. The resulting primitive shapes are added to the shape database. Domain-specific recognizers then search for domain shapes. The display module then displays the result as defined by the domain description. Chapter 5 explains the recognition process in more detail.

Chapter 4

LADDER, a *La*nguage for Describing *D*rawing, *D*isplay, and *E*diting in *R*ecognition

4.1 Overview

The previous chapter described our framework for automatically generating a sketch recognition user interface from a domain description.

As explained there, the domain description includes the following:

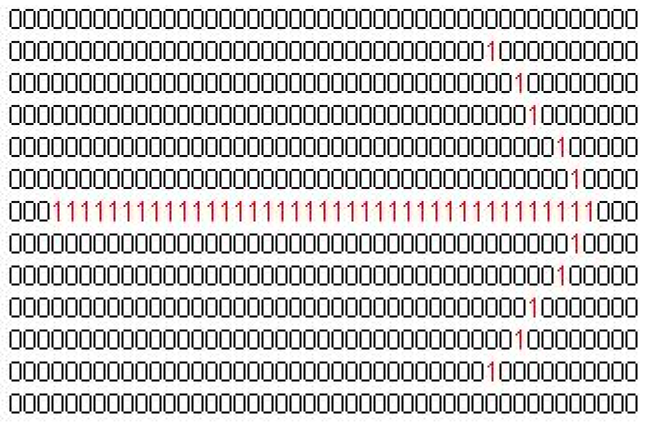
- What shapes are in the domain?
- How is each shape recognized?
- What should be displayed after each shape is recognized?
- How can the shapes be edited?

A domain description, thus, consists of: 1) a listing of all of the shapes in the domain and 2) a shape description for each shape that describes how it is recognized, displayed, and edited.

4.1.1 Goals for Shape Description Language

First and foremost, we need a language that can describe the necessary domain-specific information described above. We want the language to abstract as much information as possible in order to make a description easy to specify. Meanwhile, we also have to ensure that the language does not abstract too much and that a developer can still specify the information needed to enable the recognizer to effectively recognize shapes using these abstractions.

Domain information should be easy to specify, and it should provide a high level of abstraction to reduce the effort of and sketch recognition knowledge needed by the developer. The domain information should be accessible, understandable, and intuitive to the developer to ensure that it is easy to read, understand, and debug. The difficulty in creating such a language involves ensuring that the language is broad enough to support a wide range of domains, yet narrow enough to remain comprehensible and intuitive in vocabulary. We argue that the more understandable the domain information is, the more likely it is that its errors will be apparent.



```
000000000000000000000000000000000000000000000
000000000000000000000000000000000010000000000
000000000000000000000000000000000000100000000
000000000000000000000000000000000000010000000
000000000000000000000000000000000000000100000
000000000000000000000000000000000000000010000
000111111111111111111111111111111111111111000
000000000000000000000000000000000000000010000
000000000000000000000000000000000000000100000
000000000000000000000000000000000000010000000
000000000000000000000000000000000000100000000
000000000000000000000000000000000010000000000
000000000000000000000000000000000000000000000
```

Figure 4-1: A bit raster representation of an arrow.

The language needs to be capable of describing all acceptable variations in a shape description. For instance, we cannot use a bit raster representation of the shape, as a bit raster describes only one specific instance and does not indicate permissible variations. Figure 4-1 shows an example bit raster of an arrow. Notice that the image has no way to indicate that the shaft of the arrow can be of any length, as long as it is longer than the head of the arrow.

A shape description also should not place unnecessary drawing requirements on the user; it should allow the user to draw a shape using any number of strokes, with the strokes drawn in any order or direction. Since we want to recognize based on what the object looks like, rather than how it was drawn, we should model our language similarly. The language should allow shapes to be described primarily by user-independent geometric properties, rather than user-dependent stylistic tendencies.

While *LADDER* descriptions primarily concern shape, we still want the ability to encode information such as stroke order or direction that may be helpful in the recognition process. We also want the language to describe contextual information that could help the recognition process.

The language should encourage the reuse of shape definitions by allowing hierarchically-defined shapes. Similarly, abstract shape definitions can obviate the need to rewrite identical attributes for a class of similar shapes.

We enumerate the goals mentioned above here:

1. Able to describe domain information

2. Broad enough to support a wide range of domains

3. High level of abstraction

4. Narrow enough to remain comprehensible

5. Easy to read, understand, and debug, intuitive

6. Able to describe a generalized version of a shape
7. Can recognize description based on what the shape looks like, rather than how it was drawn
8. Able to describe how it was drawn
9. Able to describe context for recognition
10. Supports hierarchical descriptions and abstract descriptions

4.1.2 Symbolic Language Based on Shape

To achieve the goals above, we created *LADDER* [100] [103] [97]. Figure 4-3 shows a *LADDER* shape description of the open arrow drawn in Figure 4-2.

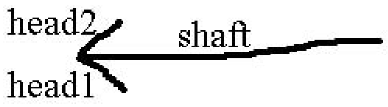


Figure 4-2: An arrow with an open head

```
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
    equalLength head1 head2
    acuteMeet head1 shaft
    acuteMeet shaft head2
  aliases
    Point head shaft.p1
    Point tail shaft.p2
  editing
    trigger holdDrag shaft
    action
      translate this
      setCursor DRAG
      showHandle MOVE tail head
    trigger holdDrag head
    action
      rubberBand this head tail
      showHandle MOVE head
      setCursor DRAG
    trigger holdDrag tail
    action
      rubberBand this tail head
      showHandle MOVE tail
      setCursor DRAG
  display original-strokes
```

Figure 4-3: The description for an arrow with an open head

A symbolic language based on shape would be easier to read and debug than a feature-based language focusing on how a shape is drawn or on non-perceptually-important geometric properties, which have been the basis of sketch recognition systems such as Graffiti^{TM}[177]. Features that have been used in such systems include the total drawing time, the size of the bounding box, stroke speed, and start/end angle. A listing of these features makes it difficult for the developer to envision the shape, making debugging difficult. These features do not ensure correlation between the drawn shape and the interpreted shape. Also, these features tend to describe user-independent stylistic tendencies, which may not be constant from person to person, but because recognition is based on these features, a sketcher must draw a shape in the same manner as the developer in order for it to be recognized (e.g., in one stroke, same speed, etc.). By defining shapes based on how they look, recognition can be performed without placing single-stroke requirements on the users, allowing users to draw the shapes as they would naturally.

While a *LADDER* shape definition is structural and includes primarily geometric information, the language also can include other drawing information, such as stroke order or stroke direction.[1] We can specify that a constraint or component is not required, using the keyword *optional*.

The language also has a number of higher-level features that simplify the task of creating a domain description. Shapes can be built hierarchically. Shapes can extend abstract shapes, which describe shared shape properties, preventing the application designer from having to redefine these properties several times; for instance, several shapes may share the same editing properties. Shapes with a variable number of components, such as a polyline or a polygon, can be described by listing the minimum and the maximum number for each component (e.g. LINE[2,N] SEGMENT would be used to define that a polyline consists of 2 or more lines called SEGMENT[i]). Contextual information from neighboring shapes also can be used to improve recognition

[1]This enables us also to describe the bulk of the characters in sketching languages, such as the Graffiti^{TM}language for the Palm Pilot [177].

by defining shape groups; for instance, contextual information can distinguish a pin joint from a circular body in mechanical engineering. Shape group information also can be used to perform chain reaction editing, such as having the movement of one shape cause the movement of another.

4.2 Intuitive Perceptual Language

In 1890, Christian von Ehrenfels said that Gestalt principles describe "experiences that require more than the basic sensory capabilities to comprehend"[212]. The brain is programmed to perceive certain visual features as more perceptually-important than others. To create an intuitive language, we looked at research in shape perception and modeled our language on these Gestalt principles. Veselova [209] lists several valuable Gestalt principles relating to shape recognition, including the grouping rules provided by Werthermeier in 1959 and the notion of singularities provided by Goldmeier in 1972 [213] [89].

Grouping rules explain how people perceptually group objects using concepts such as connectedness, nearness, and other principles. Singularities describe which geometric shape properties are most noticeable. We chose the following constraints, based on Gestalt principles for the language:

- COINCIDENT (grouping)
- CONNECTED (grouping)
- MEET (grouping)
- TOUCHES (grouping)
- NEAR (grouping)
- FAR (anti-grouping)

- INTERSECTS (grouping)
- CONTAINS (grouping)
- HORIZONTAL (singularity)
- VERTICAL (singularity)
- DIAGONAL (anti-singularity)
- PARALLEL (singularity)
- PERPENDICULAR (singularity)
- SLANTED (anti-singularity)
- EQUALLENGTH/EQUALSIZE (singularity)
- LONGER/LARGER (anti-singularity)
- SAMEX (singularity)
- LEFTOF (anti-singularity)
- SAMEY (singularity)
- ABOVE (anti-singularity)
- ACUTEMEET (grouping combined with anti-singularity)
- OBTUSEMEET (grouping combined with anti-singularity)
- BISECTS (grouping and singularity)

The language also includes many other constraints that function as syntactic sugar, being already representable by a combination of constraints in the language, such as SMALLER or CENTEREDABOVE.

The constraints in *LADDER* are modeled after these Gestalt principles, describing the perceptual importance of grouping rules and singularities. By including perceptually-important constraints, we make the language easy for humans to use. We argue that the resulting descriptions are likely to be more accurate and likely to produce better descriptions because the descriptions focus on only those details that are perceptually important.

By focusing on perceptually-important constraints, we simplify the language. As a result, we have no need for constraints that specify angles at a granularity finer than horizontal, vertical, positive slope, or negative slope. We argue that this narrowing of the language makes it more comprehensible and makes it easy to find the appropriate constraints to describe a shape.

4.2.1 Calibrating Thresholds

Gestalt singularities include horizontal and vertical lines, of which humans are particularly sensitive to. Because of this sensitivity, humans can quickly label a line as horizontal or not horizontal, as well as identify whether a line deviates from horizontal or vertical by as few as five degrees. Humans have considerably more difficulty identifying lines at other orientations, such as 35 degrees, and would have a much more difficult time determining whether a particular line were 30, 35, or 40 degrees. Humans tend to group together the angles between 15 and 75 degrees as positively-sloped lines [89].

This research attempts to calibrate how sensitive people are to these singularities in a user study. The goal of this calibration is to determine if 1) human sensitivities for horizontal and vertical lines to exist, 2) if there is a difference between vertical and horizontal sensitivities, 3) if other such sensitivities exists (i.e., are there other angles which users are more attuned to, 4) are humans more sensitive to positive or negatively sloped lines, and most importantly 5) to determine where is a good dividing point between the two locations. This calibration study just begins to answer some of these questions, and further studies are suggested in future work.

In a user study, we showed nine people a total of 116 lines in a random orientation between 0 and 180 degrees. Users were asked to report the orientations of the shown lines with as much accuracy as possible. (Any orientations in the wrong quadrant were disregarded as addition mistakes; only two lines fell in this category.) Figure 4-4 shows a graph of the actual angle of the line shown in degrees on the x axis versus the error of the guess on the y axis. Note that the errors in general seem quite high, but at 0, 90, and 180 (horizontal and vertical), the errors reduce to practically nothing. When the data is re-graphed to show on the x-axis the deviation from horizontal or vertical (as in Figure 4-5), we can clearly see the increase in error as the angle approaches 45 degrees. If we fit a line to the data in Figure 4-5, the slope of the line is .259, the y-intercept is 0, and the standard error is .016571, displaying a clear increase in the difficulty of correctly labeling the line angle as the line deviates from horizontal or vertical.

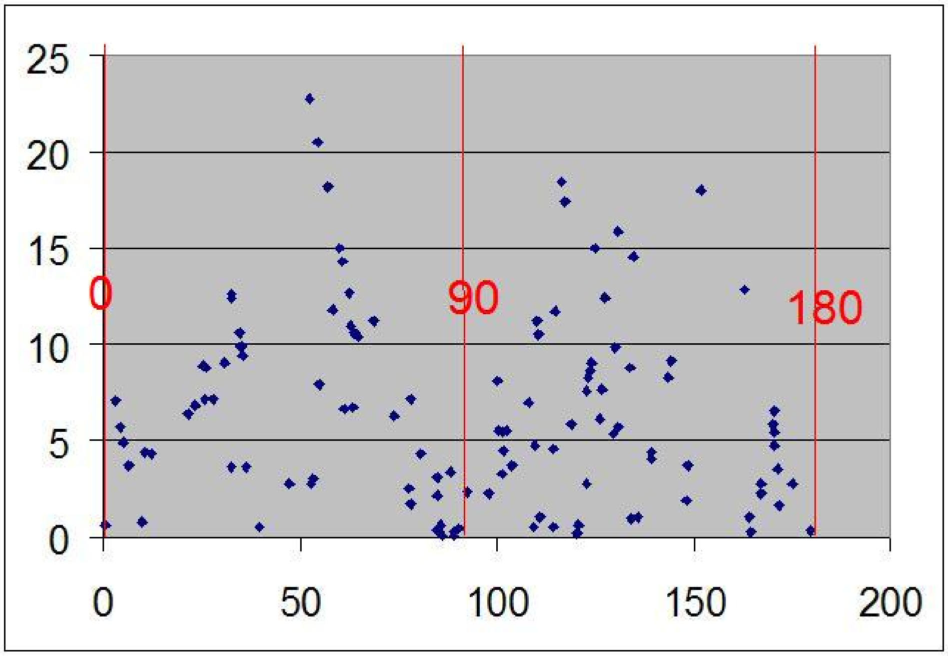


Figure 4-4: Error of the guessed angles from 0 to 180.

For the purposes of performing a t-test, we grouped the lines into two groups: 1) angles with orientations within 10 degrees of horizontal or vertical (0-10, 80-100, and 170-180 degrees) and 2) angles with orientations not within 10 degrees of horizontal

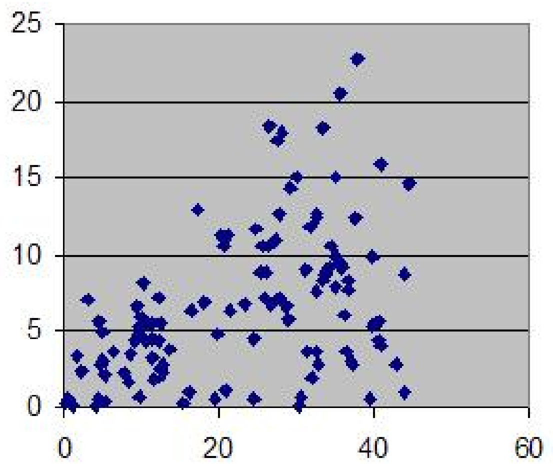


Figure 4-5: Error of the guessed angles from 0 to 45 degrees.

or vertical (10-80, 100-170). When labeling lines from group 1 (near horizontal or vertical), users had a mean error (absolute value of the reported angle minus the actual angle) of 2.8 degrees and a variance of 4.95. When labeling lines from group 2 (far from horizontal or vertical), the users had a mean error of 7.64 and a variance of 25.77. As shown by the variance, large errors between the actual orientation and the correct orientation were common: 24 lines had an error greater than 10; 8 lines had an error greater than 15; and 2 lines had an error greater than 20. The two groups were significantly different, with a p value of less than .001. The error values for positively-sloped lines is not significantly different for the error values for negatively-sloped lines, nor are the error values for horizontal lines significantly different from vertical lines. The future work section of this document proposes future studies that can more effectively select the perceptual division between horizontal/vertical lines and positive/negative-sloped lines.

4.2.2 User Study

To determine how people naturally describe shapes, we performed a user study in which 35 users described approximately 30 shapes each, both verbally and through text input. These users also had to draw shapes based on others' descriptions to test which constraints were most easily understood, as well as the validity of the original descriptions.

The experimental procedure involved four stages:

Part 1: Novice Descriptions A user is shown a shape description at the top of the screen, with the following text and a text box below it: “We want you to describe the shape you see above. Imagine you are describing this shape to a computer that understands only simple geometric shapes and relationships between them.” The user was asked first to verbally describe the shape, and then to textually describe it.

Part 2: Interpreting Descriptions The user is shown another user’s description from part 1 or 3 and asked to “Draw the shape above.”

Part 3: Experienced Descriptions Same as in Part 1.

Part 4: Structured Descriptions Users are shown a shape and asked to label the shape, using two separate text boxes. The first text box instructs, “List all of the simple geometric shapes in the shape shown. Create names for each of them of the form *line1, line2, circle1, arc1*.” The second text box instructs, “Using the names you just created above, describe the relationships between the components.”

Users were asked to speak the shape description before typing it because we were concerned that users would simply read their typed description, if asked to type and then speak. In the Novice task, users were unaffected by others’ descriptions, and, thus, more likely to use more natural language. However, the drawing task gives the users a better idea of the precision necessary in the description, so others can reproduce the drawing based on a description.

Although we expect that users of our language (a.k.a., developers of a sketch interface) will include experts in graphics-based fields in computer science, we do not want this to be a requirement for use. We expect the developers to be competent in the domain for which they are generating a sketch recognition system, and we expect them to be power computer users, but not necessarily computer programmers. In this study, we included computer programmers and non-computer programmers to mimic the intended user population.

Vocabulary

In determining how people naturally described shapes, we wanted to answer the questions: 1) Does people's shape description vocabulary align with Gestalt perceptual principles? 2) What other principles are prevalent in the common shape description vocabulary?

The subjects used 1,203 distinct words in their 520 descriptions, comprised of 209 spoken and 311 typed descriptions. The descriptions came to 19,345 total word instances, of which 766 words, totaling 18,906 instances, were used more than once. We labeled each of the words used in terms of how they were used within the description. The possible labels were: stop word (e.g., "the," "a," "those"), non-meaningful variable label (e.g., "a1," "s1"), language concept (instances of words, or their synonyms, of concepts in the Gestalt-based language vocabulary, including shapes, constraints, key-words, and other concepts describable within the capabilities of the language), everyday non-geometrical objects (e.g., golden arches, shield, coffin, bridge), and other concepts (concepts not describable in the language, such as "like" or "about").

Users tended to describe shapes using the concepts from the Gestalt principles discussed above; of these 18,906 instances, 8,992 were language concepts, 8,375 were instances of stop words or non-meaningful variable labels, 541 instances were non-geometrical objects and 998 instances (of 75 distinct words) were other concepts not included in the vocabulary. Table 4.1 lists the 14 most commonly used concepts not found in the *LADDER* vocabulary (totaling 718 out of the 998 instances) used more than 15 times.

Word	Frequency		Word	Frequency
about	77		slightly	28
as	66		inch	24
like	53		looks	24
but	45		both	22
other	36		towards	20
rotated	35		upside	18
way	32		along	18

Table 4.1: Word frequency for concepts not in the language.

The 998 instances of the 75 missing concepts were mostly concepts for describing how one shape was similar to another, followed by descriptions of how it was different. (Examples of how these missing concepts were used include: "this *looks like* X, *except* for..." or "this is *about* the *same* Y *as* X, *but rotated...*").[2] Similarities seemed to be a natural way for humans to describe shapes. This method of description agrees with the perception rule that people like to group similar things.

It would seem that the ability to refer to and extend upon other shapes is an important feature of a language, as users, several times, referred to previous shapes that they had described, despite a specific instruction not to (because descriptions were shown in a random order in Part 2[3]).

When examining the shape primitives that people used, we noticed that people made extensive use of orientation-dependent terms such as "horizontal" and "vertical," using such phrases as "horizontal rectangle" or "horizontal equilateral triangle." Users also used orientation when referring to subcomponents of a shape, e.g., "lower-right corner."

We found it interesting that users were consistent and used the same word repeatedly to describe a concept, both within their own descriptions and with others'

[2]Note that *rotated* in this list was used not to say that a shape is *rotatable*, a concept that does exist in the language, but, rather, to describe an alteration: "this *looks like* X, *but rotated* on its side." This example emphasizes that we feel that how the word was used is at least as important as, if not more than, the choice of the word itself.

[3]In cases in which users did refer to previous shapes, we included the descriptions of those shapes during the drawing task to allow other users to still be able to reproduce the shapes.

descriptions, showing the existence of a shared shape vocabulary. The extensive use of orientation-dependent descriptions showed the value of orientation-dependent constraints, including: ABOVE, BELOW, RIGHT, LEFT, HORIZONTAL, VERTICAL. Also, some shapes had an implied orientation, and users would comment on the orientation only when it was different from expected, e.g., “a sideways e.”

Users were instructed not to use non-geometrical words (such as everyday objects from their lives) and to limit themselves to a graphical vocabulary. Despite this instruction, 541 instances of non-geometrical words still occurred (e.g., golden arches, shield, coffin, bridge). They used non-geometrical words more frequently when the diagram was more cluttered, seemingly using them to simplify the description (similar to the idea of using hierarchical descriptions). Curiously, they sometimes would use non-geometrical words even if there were no clear way to draw the object from the word (e.g., the word “bridge” for the shape “][”).

Typing Versus Speaking

In determining how people naturally describe shapes, we wanted to see whether people used a different vocabulary when describing shape orally as compared to describing shapes textually. Thus, we had users do both.

We found that users’ oral language is not markedly differently from their type-written langauge. All but four users tended to type what they spoke, even when the description was quite long and difficult to remember. Of the words that were used only in speech or only in typing, none were used more than seven times each. Of those used more than four times, all were labels (e.g., “s”), typed misspellings (e.g., “diagnol”), typed shorthand (e.g., “1/3” versus “one-third”), or stop words (e.g., “what,” “guess”) that were not present in a typed description. The four users who did not type and speak the description similarly were computer science researchers. They tended to type descriptions similar to those they learned in other graphical languages (e.g., “box w 2 h 1”).

We also wanted to test whether context was more prevalent in spoken descriptions in which the user was interacting with a human as compared to interacting with a computer. We found that this was not the case; subjects used the same number of context words in both the spoken and the typed sections. In fact, the number was smaller for spoken descriptions (2.93% for spoken and 3.02% for typed), but not significantly so.

Experience

We wanted to see whether experience changed people's vocabulary. We found that, while mostly the same words were used in both the Novice and the Experienced section, the number of instances did change for those words. Of the words that were only used in one section, only one word was used more than 9 times: "some" was used 14 times in the Novice section, but never in the Experienced section.

Users definitely seemed to improve their descriptions in the second section. In the Novice section, users often would falter with how to describe something, pausing for a long time before beginning. The descriptions in the Novice section were often convoluted. Between the Novice and Experienced section, users would read other people's descriptions and attempt to draw their shapes; after seeing others' descriptions, they often would pick a simpler way to describe something, and their descriptions would be much easier to understand. A common example was the "horizontal rectangle," which was described in the Novice section in such convoluted ways as "It's a quadrilateral with two pairs of lengths, two different pairs of lengths. Each pair is congruent in length." In the Experienced section, however, it would almost always be described as a "horizontal rectangle" or "a rectangle wider than it is tall."

If we compare the number of occurrences of a word in the Novice section versus the Experienced section, we can observe the changes in word choice. If we take the absolute difference of the number of occurrences in each section, there are 8 words that have an absolute difference of 40 instances or more. Of these 8 words, all of them were much more frequent in the Experienced section, displaying a convergence to a single shared vocabulary. It suggests that, at the beginning, various words were used to describe a concept until the appropriate word was found, and then that word was used repeatedly. To give an example, "horizontal" was used 68 more times in the Experienced section than in the Novice section, which was a three-fold increase from 31 occurrences in the Novice section to 99 occurrences in the Experienced section. This was the largest factor of increase in those 8 words.

Since users were able to find the appropriate word choice more quickly, we expected that the Experienced descriptions would contain fewer words. However, the Experienced descriptions were longer, containing 7 more words per description on average (28.46 versus 35.4). Although users described concepts more concisely, they often added words to describe other parts of the shape more precisely.

Language Faults Exhibited

The language implementation allows developers to describe similarities at a primitive level (e.g., EQUALSIZE, PARALLEL). It also allows for comparison to a previous shape only when the difference with the existing shape is an addition; this is made possible by the ability to describe shapes hierarchically (e.g., "similar to the shape before, but with an extra line"). It also allows developers to describe geometrical context (geometric relations based on other shapes on the screen, e.g., this shape is bigger than that shape), but not cultural context.

However, the importance of allowing descriptions to include similarities to other shapes as well as non-geometrical cultural contextual clues was obvious given the results of this research. For the class of shapes that we have handled, so far, this has not been a problem, but it certainly could be, given how prevalent it was in the user study conducted. Both techniques could prove to be valuable additions to the language and an interesting research problem. In order to allow developers to describe shapes in terms of everyday cultural objects, we would have to 1) define each of the

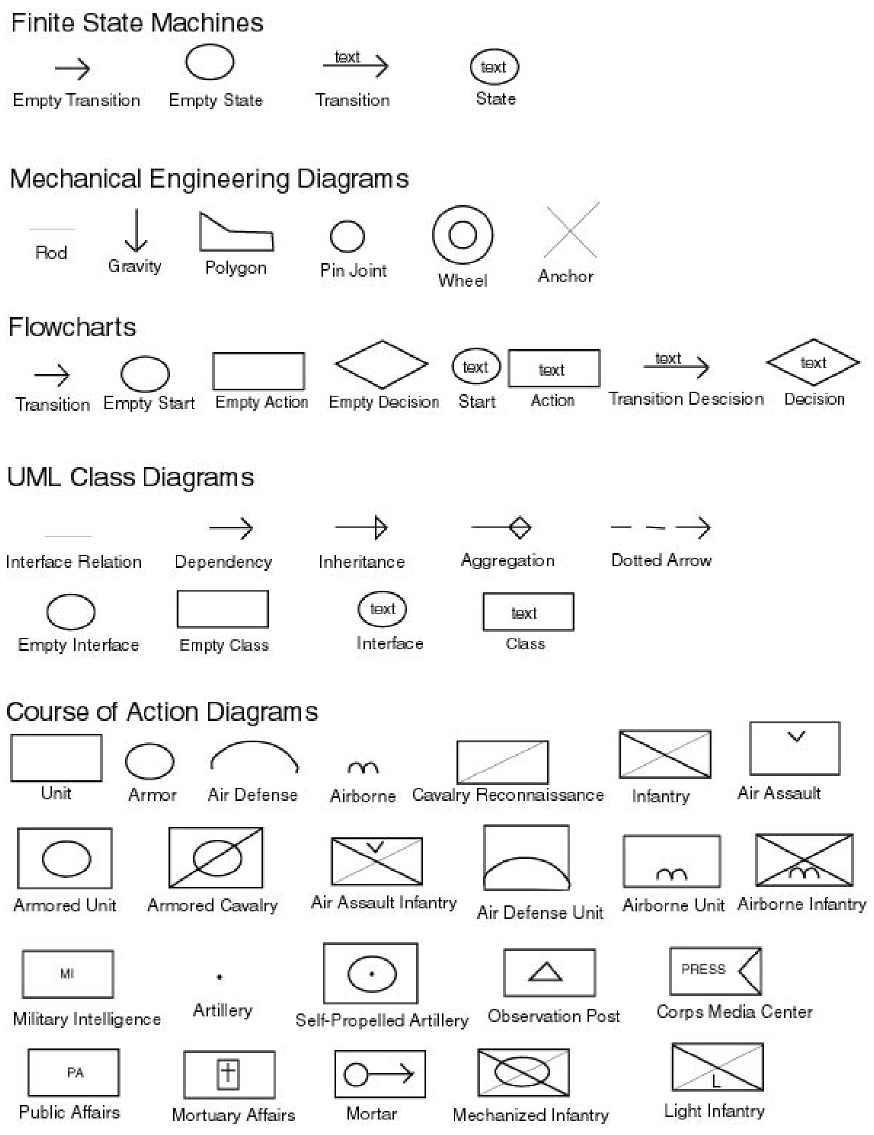


Figure 4-6: A wide variety of shapes have been described using the language.

objects that may be used in a description and 2) come up with a similarity metric for comparing them. Given the number of objects in our everyday lives, this is a difficult task. We suggest that a common-sense database, such as OpenMind[205], for accessing everyday objects, might help in implementing this technique.

4.2.3 Broad Language

To ensure that the language was broad enough to support a wide range of domains, we described a variety of shapes in many different domains, including:

- UML class diagrams
- UML sequence diagrams
- mechanical engineering
- circuit diagrams
- Tic Tac Toe
- Banesh dance notation
- sheet music notation
- block letters
- flow charts
- finite state machines
- graphical models
- an abbreviated version of course of action diagrams

Figure 4-6 shows a sampling of some of the shapes described in the language.

4.3 *LADDER* Building Blocks

The language contains predefined shapes, constraints, editing behaviors, and a syntax for combining them. A domain description is specified by a list of the shapes and shape descriptions for each shape in the domain. Figure 4-8 shows an example description for the OpenArrow drawn in Figure 4-7. 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.

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

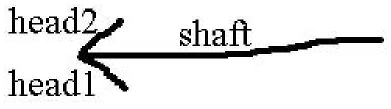


Figure 4-7: An arrow with an open head.

```
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
    equalLength head1 head2
    acuteMeet head1 shaft
    acuteMeet shaft head2
  aliases
    Point head shaft.p1
    Point tail shaft.p2
  editing
    trigger holdDrag shaft
    action
      translate this
      setCursor DRAG
      showHandle MOVE tail head
    trigger holdDrag head
    action
      rubberBand this head tail
      showHandle MOVE head
      setCursor DRAG
    trigger holdDrag tail
    action
      rubberBand this tail head
      showHandle MOVE tail
      setCursor DRAG
  display original-strokes
```

Figure 4-8: The description for an arrow with an open head

4.3.1 Predefined Shapes

The language includes the primitive shapes Shape, Point, Path, Line, Curve, Arc, Ellipse, Spiral, and Text,[4] as well as a library of predefined shapes built from these primitives, including Rectangle, Diamond, and Triangle. (For example, the shape definition OpenArrow in Figure 4-3 is built from three lines.) We chose these primitive shapes, as they were sufficient to describe all of the shapes we came across in the wide variety of domains examined.

The language uses an inheritance hierarchy; Shape (a language keyword) is an abstract shape which all other shapes extend. Shape provides a number of components and properties for all shapes, including *boundingBox, centerpoint, width, height, minx, maxx, miny, maxy, x, y, size*, and *time*. Each predefined shape may have additional components and properties; a Line, for example, also has *p1, p2* (the endpoints), *midpoint*, *length* (used instead of *size* for lines). 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. The accessible properties of the primitive shapes are predefined in the language's syntax.

Because *LADDER* is hierarchical, any defined shape can be used as a component of other shape descriptions. Thus, as we define more shapes, we form a library of shapes to be shared across domains. The predefined shapes and their accessible properties are listed in Appendix A.1.

4.3.2 Predefined Constraints

New shapes are defined in terms of previously-defined shapes and the constraints between them. For instance, the OpenArrow shape description in Figure 4-3 con-

[4]Text is entered through the keyboard or through the handwriting recognizer input pad built into the Windows Tablet PC operating system.

tains the constraint (acuteMeet 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.) The other constraints ensure that all of the lines meet at one point, and that the lines forming the heads of the arrow are the same length.

A number of predefined constraints are included in the language, such as perpendicular. The orientation-dependent constraints include horizontal, vertical, negSlope, posSlope, above, leftOf, horizAlign, vertAlign, pointsDown, pointsLeft, pointsRight, and pointsUp. The orientation-independent constraints include: acute, acuteDir, acuteMeet, bisects, coincident, collinear, concentric, connects, contains, drawOrder, equalAngle, equalArea, equalLength, intersects, larger, longer, meets, near, obtuse, obtuseDir, obtuseMeet, onOneSide, oppositeSide, parallel, perpendicular, and sameSide.

The vocabulary also includes equal, greaterThan, and greaterThanEqual, allowing the developer to compare any two numeric properties of a shape (e.g., stating that the height is greater than the width). The constraint modifiers, or and not, are also present to allow the developer to describe more complicated constraints.

The vocabulary also contains a number of constraints that can be composed from other constraints. We include these constraints to simplify descriptions and to make them more readable. They include: smaller, below, rightOf, aboveLeft, aboveRight, belowLeft, belowRight, centeredAbove, centeredBelow, centeredLeft, centeredRight, centeredIn, lessThan, lessThanEqual.

A more detailed explanation (including the predefined constraints and their arguments) is listed in Appendix A.2.

4.4 Predefined Editing Behaviors

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.

LADDER provides the ability to describe editing gestures so that the recognition system can discriminate between sketching (pen gestures intended to leave a trail of ink) and editing gestures (pen gestures intended to change existing ink), and because editing behaviors are different in different domains.

Although we do encourage standardization between different domains by including some predefined editing behaviors, it is important to 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 one domain (a check inside of a checkbox, or the letter "X" in a textbox) or as an editing command (deletion of the box) in another domain.

Depending on the domain, a *LADDER* description might define an "X" as a drawing gesture or a deletion gesture. For example, the domain TIC TAC TOE listed in Appendix B defines a CROSS as a drawable shape in its domain. Because it is a drawable shape in the domain, we do not allow the "X" to also signify deletion, as it could cause confusion. However, since the UML class diagrams domain does not commonly include an "X," the developer is free to define the "X" to signify deletion, rather than defining it as a new shape. In this case, we add an editing behavior to signify that "X" should delete a shape (using the trigger: DRAWOVER CROSS THIS and action: DELETE THIS).

In order to encourage interface consistency, the language includes a number of predefined editing behaviors, described using the predefined actions and triggers. One such example is the editing behavior DRAGINSIDE, which indicates that if one holds the pen for a brief moment inside the bounding box of a shape, and then starts to drag the pen, the entire shape automatically translates along with the motion of the pen. Another example is the editing behavior SCRIBBLEDELETE, which states that if one scribbles over the strokes of a shape, that shape will be deleted. To turn on these editing behaviors, the developer must add them to the shape's description itself.

The arrow definition in Figure 4-9 defines three editing behaviors. The first editing behavior says that if one clicks and holds the pen over the SHAFT of the ARROW when dragging the pen, the entire ARROW will translate, along with the movement of the pen. The second editing behavior states that if one clicks and holds 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 to keep the ARROW as one whole shape. The third is similar to the second, defining rubber-banding over the tail. All of the editing behaviors also change the appearance of the pen's cursor and display moving-handles to the sketcher to indicate an editing command.

```
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
    equalLength head1 head2
    acuteMeet head1 shaft
    acuteMeet shaft head2
  aliases
    Point head shaft.p1
    Point tail shaft.p2
  editing
    trigger holdDrag shaft
    action
      translate this
      setCursor DRAG
      showHandle MOVE tail head
    trigger holdDrag head
    action
      rubberBand this head tail
      showHandle MOVE head
      setCursor DRAG
    trigger holdDrag tail
    action
      rubberBand this tail head
      showHandle MOVE tail
      setCursor DRAG
  display original-strokes
```

Figure 4-9: The description for an arrow with an open head

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, then *move-point* translates according to the movement of the mouse. RUBBERBAND is used in the editing definition in Figure 4-9.

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. "Chain reaction" editing allows the developer to use a triggered action to also trigger another action.

Shape groups also 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.

The language has a number of predefined triggers and actions to aid in describing editing behaviors. The possible triggers include all of those listed in Appendix A.3.1, as well as all of the actions listed in Appendix A.3.2, allowing for "chain-reaction" editing.

4.4.1 Predefined Display Methods

An important part of a sketching interface is controlling what the sketcher sees after shapes are recognized. Altering the display can help to beautify the document and serve as a feedback mechanism illustrating that the shape has been recognized. The designer can specify either that the original strokes should remain, that a cleaned-up version of the strokes should be displayed, that the constraints should be solved and the ideal shape should be shown, or that the strokes should be replaced entirely with Java swing shapes or the contents of an image file. In the cleaned-up version, the original stroke segments are fit to their recognized straight lines, clean curves, clean arcs, or perfect ellipses.

LADDER also can display the ideal version of the strokes, in which all constraints are solved using MATLAB. For example, lines that are supposed to connect at their endpoints are connected, and lines that are supposed to be parallel are made parallel. In the ideal version of the strokes, all of the signal noise (defined in Section 5.1.1) arising from sketching is removed.

It may be that we do not want to show any version of the strokes at all, but want to display 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 and their arguments are listed in Appendix A.3.3.

4.5 Syntax for Defining a Domain Description

This section describes the syntax of a domain description. As an illustrative example, we will walk through the creation of a domain description for UML (Unified Modeling Language) [37]. The Appendix B shows this and other domain descriptions in their entirety. We refer to several examples in the UML class diagram domain description throughout this chapter. UML is chosen here for its simplicity (few shapes) and sophistication (similar shapes, several display and editing option) for easier explanation of the language. However the Appendix B lists several more complicated examples for perusal.

Recall that to create a domain description, one must specify the list of shapes and shape interactions in the domain. One also must specify how each of the shapes and its interactions are drawn, displayed, and edited.

4.5.1 Domain List

The functional shapes in a UML class diagram are listed below. The name of the shape is followed by a geometric description in parenthesis.

- UML Class (represented by a rectangle)
- UML Interface (represented by a circle)
- UML Interface Association (represented by a line)
- UML Dependency Association (represented by a open-headed arrow - the arrow shaft can be solid or dashed)
- UML Aggregation Association (represented by a diamond-headed arrow)
- UML Inheritance Association (represented by a triangle-headed arrow)

The system requires a similar listing of the domain shapes. The domain list for UML class diagrams is shown in Figure 4-10. This listing is saved in an .ldl file (*LADDER* Domain List) and includes all of the functional shapes used in the domain, as well as any geometrical or abstract shapes used to build those shapes. Primitives do not need to be included in this list.

If we want, we also can specify how many times a domain object is expected to occur in a particular sketch in the domain. For example, in mechanical engineering, every diagram is expected to contain, at most, one symbol for gravity, and, in electrical engineering, every electrical circuit has at least one ground. We can specify this in our domain list by adding one of the following wildcards after the shape name:

- $*$: specifying that a domain object can occur any number (0 or more) of times in the domain
- positive integer n : specifying that a domain object will occur n times in the domain

```
define domain UML
  AbstractArrow
  OpenArrow
  TriangleArrow
  DiamondArrow
  DashedLine
  DashedArrow
  Rectangle
  DependencyAssociation
  AggregationAssociation
  InheritanceAssociation
  InterfaceAssociation
  AbstractAssociation
  InterfaceClass
  GeneralClass
  AbstractClass
```

Figure 4-10: The domain listing of shapes for UML class diagrams.

- a positive integer n followed by a plus sign (+) : specifying that a domain object will occur n or more times in the domain
- a positive integer n followed by a minus sign (-) : specifying that a domain object will occur n or fewer times in the domain
- nothing : same as ***
- *+* : same as *1+*
- - : same as *1-*

If a shape that violates this requirement is drawn, it is not recognized. If the developer would prefer that the shape be recognized, even if this constraint is violated, the developer should not include the constraint. For instance, if a sketcher draws two gravity arrows, but only one is allowed, the developer may want both of them to be recognized and remove the wildcard. However, since the same symbol (a downward arrow) could also represent a downward force for a particular object, the developer may choose to enforce this requirement with a wildcard, leaving it as an unrecognized and unfinished force that is waiting for a physical body to push. (Note that, in practice, we usually have chosen not to use these features. There has not been evidence, thus far, that these features are useful. In fact, strictly constraining the user can prevent some alternate interpretations. For example, this may cause the problem that if the first downward arrow drawn is awaiting a body to push, then the second downward arrow is for gravity. In certain circumstances, the system may recognize the first as gravity, and then leave the second unrecognized, even when the body was added to the first.)

4.5.2 Shape Definition

New shapes are defined in terms of previously-defined shapes and the constraints between them. Figure 4-9 shows a shape definition for an OpenArrow. Figure 4-11 shows a shape definition for a TriangleArrow built from the OpenArrow. The definition of a shape consists of seven sections. All sections are optional except the *components* section.

1. The *description* contains a textual description of the shape. For example, the description of the TriangleArrow in Figure 4-11 is "an arrow with a triangle-shaped head."

2. The *is-a* section 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. This section may be excluded if the shape only extends Shape.

3. A list of *components* specifies the elements from which the shape is built. Note that the OpenArrow is built from 3 lines. The TriangleArrow in Figure 4-11 is built from the OpenArrow from Figure 4-9 and a Line. Components can be accessed hierarchically. For example, the TriangleArrow accesses the shaft of its OpenArrow component with the statement oa.shaft.

4. Geometric *constraints* define the relationships on those components. The OpenArrow shape definition requires that the HEAD1 and SHAFT meet at a single point and form an acute angle from line HEAD1 to line SHAFT when traveling in a counter-clockwise direction.

   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 4-11, the shaft of the arrow is commonly drawn before the head of the arrow, but the arrow still should be recognized, even if this constraint is not satisfied. All constraints are hard constraints unless specified otherwise.[5]

5. A set of *aliases* is used to simplify this description and descriptions built using this description to rename properties and shapes for ease of use later. Both the OpenArrow and TriangleArrow descriptions have added aliases for the HEAD and TAIL to simplify the task of specifying editing behaviors.

6. *Editing* behaviors specify the editing gesture triggers and how the object should react to these editing gestures. The TriangleArrow 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 is 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.[6]

7. *Display* methods indicate what is to be displayed when the shape is recognized.

[5] *Soft* constraints are not currently supported by the implemented recognition system.

[6] Rubber-banding allows sketchers to simultaneously rotate and scale an object, assuming a fixed rotation point is defined [74]. This action has proved useful for editing arrows and other linking shapes.

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.) or images (jpg, gif) to draw and where. The TriangleArrow definition specifies that the arrow should be displayed in the color red and drawn using cleanedStrokes (i.e., using straight lines in this case).

```
define shape TriangleArrow
  description   "An arrow with a triangle-shaped head"
  is-a Shape
  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 holdDrag shaft
    action
      translate this
    trigger holdDrag head
    action
      rubberBand this head tail
    trigger holdDrag tail
    action
      rubberBand this tail head
    trigger scribble shaft
    action
      delete this
  display
    cleanedStrokes
    color red
```

Figure 4-11: The description for an arrow with a triangle-shaped head.

Hierarchical Shape Definitions

To simplify shape definitions, shapes can be defined hierarchically. Note that the TriangleArrow in Figure 4-13 is composed of an OpenArrow and a Line. By defining shapes hierarchically, we can define complicated shapes more simply. For example, once the developer has defined the TriangleArrow, she can create higher-level shapes, using this newly-defined TriangleArrow as a component.

Abstract Shape Definitions

In the domain of UML class diagrams, there are three different types of associations represented by four different types of arrows: an arrow with an open-headed arrow with a solid shaft (dependency association), an open-headed arrow with a dashed shaft (dependencey association), an arrow with a triangle head and a solid shaft (inheritance association), and an arrow with a diamond head and a solid shaft (aggregation association). All of these arrows have the same editing behaviors. Rather than repeat the editing behaviors four times, we, instead, create an AbstractArrow (shown in Figure 4-12, which specifies the repeated editing behaviors). The *is-a* section, used in Figure 4-11, 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 shapes 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 that 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.

```
define abstract-shape AbstractArrow
  required
    Point head
    Point tail
    Line shaft
  editing
    trigger holdDrag shaft
    action
      translate this
      setCursor DRAG
      showHandle MOVE tail head
    trigger holdDrag head
    action
      rubberBand this head tail
      setCursor DRAG
      showHandle MOVE head
    trigger holdDrag tail
    action
      rubberBand this tail head
      setCursor DRAG
      showHandle MOVE tail
    trigger scribble shaft
    action
      delete this
  display
    cleanedStrokes
    color red
```

Figure 4-12: The description for the abstract class AbstractArrow.

```
define shape TriangleArrow
  is-a AbstractArrow
  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
```

Figure 4-13: A shortened description for a TriangleArrow by taking advantage of properties from AbstractArrow.

Shape Context

We also can use surrounding shapes to provide context so that we may more effectively recognize shapes in a domain. For example, in mechanical engineering, both forces and gravity are represented by arrows. However, forces are drawn to push a particular body (a mechanical engineering term describing a physical mass), whereas gravity is not. Figure 4-14 defines FORCE as an arrow, but the definition also gives contextual information to imply that not all arrows are FORCEs. Rather, the shape definition indicates that the ARROW must be *pushing* a BODY (where we define *pushing* geometrically to mean that the head of the arrow touches the body). We use the keyword *context* to emphasize that the context shape (the BODY shape in our example) indicated in the definition is not part of the complete shape (the FORCE in our example), but simply provides contextual information to aid recognition.

We also can use shape context to describe how shapes interrelate in order to define "chain reaction" editing behaviors, which allow the editing of one shape to trigger the editing of another. For instance, in Figure 4-14, when we move a BODY, we want the FORCE to move, as well.

```
define shape Force
  description "An arrow is a force only if the arrow head is
    pushing an object."
  components
    Arrow force
    context Polygon body
  constraints
    meet force.head body
  editing
    trigger dragHold body
    action
      move force
```

Figure 4-14: The definition of a Force in mechanical engineering, which uses contextual information.

Vectors

The arrow defined in Figure 4-9 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 keyword *vector*; the keyword must be accompanied by a specification of the minimum and the maximum number of components allowed in the shape. 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 Figure 4.5.2).

```
(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 4-15: Shape description of a polygon.

Likewise, a DASHEDARROW is made from an ARROW and a DASHEDLINE (both defined in Figure 4-16), 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 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
```

Figure 4-16: Description of a dashed line and a dashed open arrow.

4.6 Limitations

4.6.1 *LADDER* 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 drawn using the same graphical components each time. For instance, we cannot describe abstract shapes, such as people or cats, that would be sketched 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. Curves are inherently difficult 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 [24] [25].
- Likewise, *LADDER* can only describe shapes that have a lot of regularity and not too much detail. If a shape is highly irregular and complicated, so that it cannot be broken down into subshapes that can be described, it will be cumbersome to define.

4.6.2 Curves

Curves are inherently difficult to describe in detail because of the difficulty in specifying a curve's control points. Future work should include investigating more intuitive ways of describing curves.

4.6.3 Language Faults Exhibited in the User Study

The language implementation allows developers to describe similarities at a primitive level (e.g., EQUALSIZE, PARALLEL). It also allows for comparison to a previous shape only when the difference with the existing shape is an addition; this is made possible by the ability to describe shapes hierarchically (e.g., "similar to the shape before, but with an extra line"). It also allows developers to describe geometrical context (geometric relations based on other shapes on the screen, e.g., this shape is bigger than that shape), but not cultural context.

However, the importance of allowing descriptions to include similarities to other shapes as well as non-geometrical cultural contextual clues was obvious given the results of this research. For the class of shapes that we have handled, so far, this has not been a problem, but it certainly could be, given how prevalent it was in the user study conducted. Both techniques could prove to be valuable additions to the language and an interesting research problem. In order to allow developers to describe shapes in terms of everyday cultural objects, we would have to 1) define each of the objects that may be used in a description and 2) come up with a similarity metric for comparing them. Given the number of objects in our everyday lives, this is a difficult task. We suggest that a common-sense database, such as OpenMind[205], for accessing everyday objects, might help in implementing this technique.

Chapter 5

GUILD (Generator of User Interfaces from a *LADDER* Description) Recognition System

Once we have a domain description, we need to translate that into a user interface for that domain. This researcher built a system called *GUILD* (Generator of User Interfaces from a *LADDER* Description) that automatically creates a sketch recognition system from a *LADDER* domain description.

A *LADDER* domain description must be translated into shape recognizers (from the *components* and *constraints* sections), exhibitors (from the *display* section), and editors (from the editing section) that recognize, edit, and display each domain shape within the generated domain user interface.

The translation process is analogous to work done on compiler compilers, in particular, visual language compiler compilers by Costagliola et. al. [54]. 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 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, while our primitives are geometric.

5.1 Signal Noise

5.1.1 Signal Noise versus Conceptual Variations

By signal noise, we mean the unintentional deviations introduced into a shape by the imprecision of hand control. For instance, when drawing a square, all four sides may turn out to be of different lengths even though the sketcher meant for them to be the same length. By conceptual variations, we mean the allowable variations in a symbol that are drawn intentionally. For example, a capacitor in an electronic circuit may be drawn as two parallel lines, or as one straight and one curved line (see Figure 5-1).

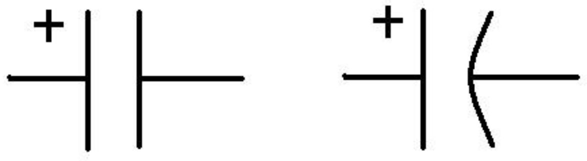


Figure 5-1: A capacitor can be drawn with two lines or a line and a curve.

In our system, signal noise is handled by the recognition system. For example, the system can successfully recognize a quadrilateral with uneven sides as a square because the EQUALLENGTH constraint has some built-in tolerance (discussed below). Thus, shapes should be described to the system without accounting for signal noise, i.e., as if drawn perfectly (e.g., a square should be described as having equal length sides). As the system does not automatically take into account the possible conceptual variations (indeed, how could it?), they must be provided for in the shape descriptions.

Other signal errors include a sketcher intending to draw a single line, but using several strokes to do so. In order for the system to deal with these phenomena, it first joins lines by merging overlapping and connecting lines. Figure 5-2 shows

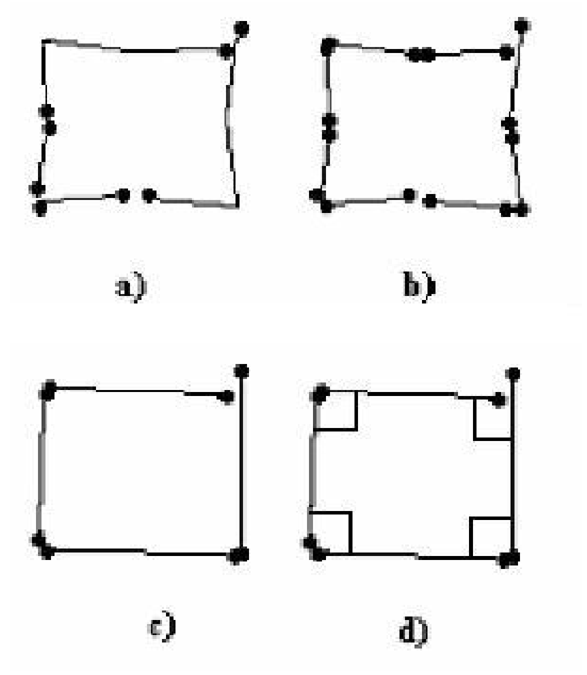


Figure 5-2: Stages of square recognition; a) original strokes, b) primitive shapes, c) joined/cleaned primitives, and d) higher-level recognition.

the steps that go into recognizing a square. Figure 5-2a shows the original strokes. Figure 5-2b shows the original strokes broken down into primitives. The system has recognized the strokes as lines or polylines; the figure shows the straightened lines that were recognized by the recognition system. (The dots represent their endpoints.) Figure 5-2c shows the primitives (line segments, in this example) joined together to form larger primitives (again lines, in this example) using the merging techniques described above. Figure 5-2d shows the higher-level recognition performed on the recognized shapes; the method for this is described in the next section. A higher-level shape can then use the square as one of its components.

5.2 Recognition

The current recognition system is based on an internal indexing system. It is an improvement on two prior recognition/translation systems built by this researcher.

5.2.1 Past Implementations

The first system generated Java classes for each shape's recognizer, editor, and the shape itself which specified its display capabilities. These Java classes did recognition by examining different subset collections of shapes, checking whether that subset could be an example of the desired shape. Constraints were recomputed for each higher-level shape tested, and, thus, recognition using this method was not as fast as it could be. However, the main problem with this method was that, if a small change was to be made about how to recognize a shape (e.g., if a single constraint was added), the entire system had to be shut down, and the code had to be recompiled and restarted in order to reflect the change.

The second system generated Jess rules to perform recognition from a series of bottom up opportunistic data driven triggers [82]. Figure 5-3 shows an example of a Jess rule that was generated to recognize a TriangleArrow. 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). The Jess-based system improved on the previous implementation in that it took advantage of the Rete algorithm to perform quicker pattern matching on shapes. Also, changes could be made to recognition rules (such as adding a new shape to be recognized or modifying a current shape) in run time. While recognition is initially quick using this system, the system slows down exponentially when unrecognized strokes are added to the screen. After 30 unrecognized lines are added to the screen, the system is so backed up that it seems to be indefinitely halted (i.e., more than an hour to catch up).

```
(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 5-3: Automatically-generated Jess rule for the arrow definition shown on the left side of Figure fig:recognition:translator.

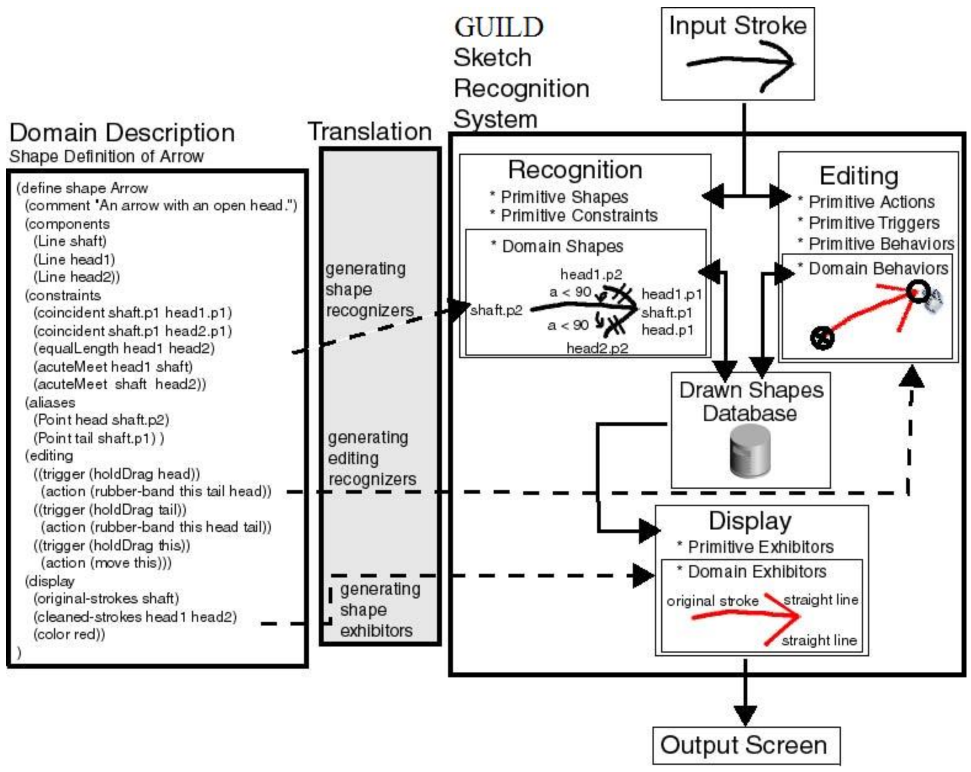
5.2.2 Motivation for Current Implementation

The current system is a major improvement in two ways. Translation from a shape description to its recognizer is done internally, which means that when a new shape is defined using the debugger (described in Chapters 9-11), the new shape is immediately and seamlessly recognized in the recognition system without stopping the system and recompiling, or requiring any other human intervention. 2) The new approach uses an indexing algorithm to perform much faster recognition. It processes each stroke as it is drawn and puts each stroke in a variety of indices (explained throughout the rest of this chapter). These indices are used later for fast look up. This method is much faster, and allows recognition to occur with almost no delay, even when many unrecognized shapes lie on the screen and have to be considered to form higher-level shapes.

Sketch recognition is the process of combining lower-level shapes on the screen to create higher-level shapes. These higher-level shapes are defined by the subshapes of which they are composed and constraints specifying how the subshapes fit together. To recognize all higher-level shapes, the recognition system must examine every possible combination of subshapes, as well as every permutation of these subshapes. This implies that any straightforward algorithm would take exponential time, which is clearly impractical for any non-trivial sketch.

To combat this problem, other recognition systems have placed drawing requirements on the user [185] [150], such as requiring users to draw each shape in its entirety before starting the next shape, or forcing users to draw each shape in a single stroke. This produces an unnatural drawing style. Mahoney et. al. [157] have discussed the inherent complexity involved in the structural matching search task. They reduce the problem to a constraint satisfaction subgraph problem, but their solutions still take an exponential time in practice.

Our goal is to make sketch systems as natural as possible by placing as few requirements on the user as possible. In our experience, observing users sketch in a



```
(define shape Arrow
  (comment "An arrow with an open head.")
  (components
    (Line shaft)
    (Line head1)
    (Line head2))
  (constraints
    (coincident shaft.p1 head1.p1)
    (coincident shaft.p1 head2.p1)
    (equalLength head1 head2)
    (acuteMeet head1 shaft)
    (acuteMeet  shaft  head2))
  (aliases
    (Point head shaft.p2)
    (Point tail shaft.p1) )
  (editing
    ((trigger (holdDrag head))
      (action (rubber-band this tail head))
    ((trigger (holdDrag tail))
      (action (rubber-band this head tail))
    ((trigger (holdDrag this))
      (action (move this)))
  (display
    (original-strokes shaft)
    (cleaned-strokes head1 head2)
    (color red))
)
```

Figure 5-4: GUILD: The domain description (shown on the left side of the figure) is translated into recognizers, exhibitors, and editors for each shape in the domain (shown on the right side of the figure).

variety of domains, we have found that it is not uncommon for someone to draw part of a shape, stop, continue drawing other objects in a diagram, and then come back to finish the original shape. Figure 5-5 shows an example in mechanical engineering. We have seen interspersing in software design, where UML class diagram designers sometimes initially draw connections as simple dependency associations, until most of the classes have been drawn, at which point they will have a better understanding of the emerging design, and make a more informed decision about the type of association that would be appropriate between two objects. This will cause them to add different kinds of arrowheads to the associations drawn earlier, producing an interspersed sketch [99]. We have also witnessed interspersing in electrical engineering; sketchers add voltage directions only after most of the circuit has been drawn.

One way to recognize interspersed drawings is to recognize shapes by how they look, rather than how they were drawn. In order to recognize completed shapes, while

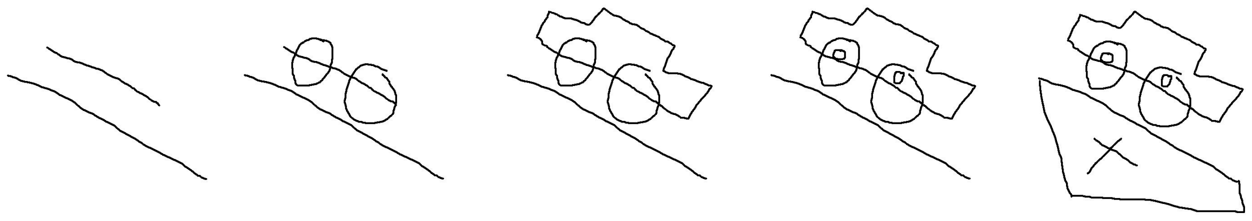


Figure 5-5: Five snapshots in the design of a car on a hill in mechanical engineering. Note that the top of the ground was drawn first, followed by the car base, to ensure that they are parallel. Then wheels (which act as the connection between the car and the ground) were added. The wheels were attached to the car only after the car was completed. The first started object, the ground, was completed last since its purpose in completion was more in terms of general functioning when attaching it to a CAD system than in design.

still allowing incomplete, interspersed shapes, the recognition system must examine all possible subsets of shapes, i.e., the power set of all of the shapes, S, on the screen, which is, of course, exponential in $|S|$ (the size of S, or the number of shapes on the screen). Yet, we also want to keep interaction close to real-time.

Here, we describe our indexing technique for sketch recognition that examines all $2^{|S|}$ shape subsets when attempting to recognize new shapes, but uses efficient indexing to keep recognition performance close to real-time. The technique takes advantage of the *LADDER* constraint language to index each stroke efficiently for quick access later. This document also reports timing data that supports the claim that the recognition of new shapes can be kept close to real-time, even when all possible shape subsets are considered.

Not only do we want to recognize shapes drawn in other orders, but previous parts of this thesis motivated the need to recognize shapes by how they look, using perceptual principles. Our algorithm takes advantage of the shape-based recognition, indexing the geometrical properties of a shape, to provide a basis for indexing and, thus, fast lookup.

5.2.3 Constraint Tolerances

In our approach, signal error is handled by the shape recognizer by giving each constraint its own error tolerance, chosen to be as close as possible to perceptual tolerance, i.e., the tolerance that humans use. Human perceptual tolerance is context-dependent, depending on both the shape in question and other shapes on the screen. Table 5.2.3 shows constraints and the error tolerances chosen. Note that some constraints have an absolute tolerance, while others are relative. Some constraints have a negative tolerance, which means the constraint has to be "blatantly true" in order to be recognized. This ensures that a constraint is not only *geometrically satisfied*, but also *perceptually satisfied*, meaning that humans will be able to perceive that the constraint is true. For example, a shape that is to the left of another shape by one pixel is geometrically to the left, but is not perceptually to the left, as it is difficult for a human to perceive such a small distance; in this case five pixels. To ensure that a constraint is *perceptually satisfied*, we add a buffer zone to the tolerance. Perceptual error tolerances were determined empirically for horizontal, vertical, posSlope, and negSlope (as shown in the previous chapter) or estimated from the Gestalt principles described previously.

5.2.4 Indexing Algorithm

Recognition is done in three stages: 1) domain-independent primitive finding, 2) domain-independent constraint indexing, and 3) domain-dependent shape formation.

Domain-Independent Primitive Finding

When a stroke is drawn (and has not been identified as an editing gesture as described in Section 5.3), low-level recognition is performed on it. During processing, each stroke is broken down into a collection of primitive shapes, including line, arc, circle, ellipse, curve, point, and spiral, using techniques from Sezgin [194]. Corners used for

CONSTRAINT	**UNIT**	**TOLERANCE**
horizontal	angle	10 degrees
vertical	angle	10 degrees
posSlope	angle	35 degrees
negSlope	angle	35 degrees
coincident	x location	10 pixels
coincident	y location	10 pixels
bisects	x location	(length / 4) pixels
bisects	y location	(length / 4) pixels
near	x location	50 pixels
near	y location	50 pixels
concentric	x location	(width / 5) + 10 pixels
concentric	y location	(width / 5) + 10 pixels
sameX	x location	20 pixels
sameX	width	20 pixels
sameY	y location	20 pixels
sameY	height	20 pixels
equalSize	size	(size / 4) + 20 pixels
parallel	angle	15 degrees
perpendicular	angle	15 degrees (of a 90 degree difference)
acute	angle	30 degrees (of a 45 degree difference)
obtuse	angle	30 degrees (of a 135 degree difference)
contains	min X,Y & max X,Y	-5 pixels
above	y location	-5 pixels
leftOf	x location	-5 pixels
larger	size	-5 pixels

Table 5.1: Constraints and their error tolerances. (Note: size = length of diagonal of bounding box. We use this formula instead of the area to enable us to compare lines to two-dimensional shapes. Rubine also chose the same formula for comparing the sizes of objects for one of his features. [185])

segmentation are found, using a combination of speed and curvature data (as in Sezgin [194]). By breaking strokes down into these primitives and performing recognition with primitives, we can recognize the shapes that have been drawn using multiple strokes, and handle situations in which a single stroke was used to draw multiple shapes as long as the corners break the stroke appropriately (e.g., the top line in Figure 5-6 cannot be broken to find the two squares).

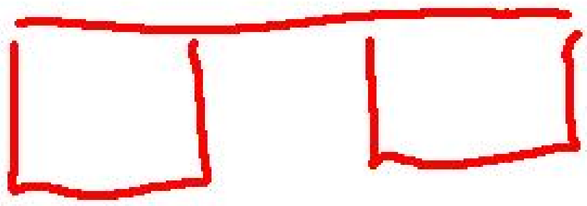


Figure 5-6: This image was drawn with only three strokes. The system cannot recognize the two squares because the top line cannot be split into three lines without corner data.

If a stroke has multiple primitive interpretations, all interpretations are added to a pool of interpretations, but a single interpretation is chosen for display. For example, both the LINE and ARC interpretation of the STROKE in Figure 5-7A will be added to the pool for recognition using any of the interpretations.

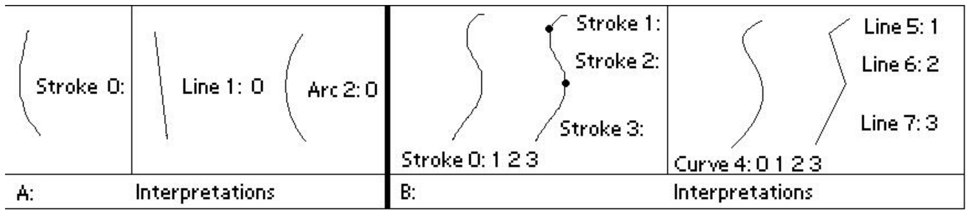


Figure 5-7: Multiple interpretations and their identifiers are added to the recognition pool. In A, STROKE 0 has two interpretations: LINE 1 and ARC 2, each composed from STROKE 0. In B, STROKE 0 can be broken down into three STROKES (1,2,3). STROKE 0 has two interpretations: CURVE 4, composed of STROKE 0 (and thus also STROKES 1,2,3); and three lines: LINE 5, composed of STROKE 1; LINE 6, composed of STROKE 2; and LINE 7, composed of STROKE 3.

We want to ensure that the shape recognition system chooses only one interpretation of a single stroke. In order to ensure that only one interpretation is chosen, each shape has an ID, and the appropriate bookkeeping is done to ensure that, while multiple interpretations are kept while the system is working (i.e., recognizing), the final displayed result contains only a single interpretation for each stroke.

In cases in which a stroke is determined to be composed of several primitives (e.g., the Polyline interpretation in Figure 5-7B), the Stroke is segmented, and the segmented substrokes added as components of the original full Stroke. Further interpretations can use either the full stroke, as the Curve does in Figure 5-7B, or one or more of the polyline-interpretation substrokes. This allows several shapes to be drawn with a single stroke.

Domain-Independent Constraint Indexing

We would prefer to place as few drawing requirements as possible on the sketcher, and must, as a consequence, find a way to deal with the exponential. While our solution does not eliminate the exponential, we can use indexing to do a significant amount of the computation ahead of time. Because the indexing of a shape is dependent only on its own properties, the time it takes to index a specific shape is not exponential (it is actually logarithmic in the number of shapes on the screen because several of the properties are inserted into a sorted hash map). The indexing process occurs only once for each shape: right after it has been recognized. The recognition process is still exponential (as it must be if we are still to consider all possible subshapes), but, in practice, it is very fast because most of the process has been moved to the indexing pre-processing stage, which is not exponential.

When a new shape is recognized, the system computes its properties, including its orientation, angle, location, and size. Each property has its own data structure, permitting quick retrieval of shapes with the desired property value. For instance, the angle data structure is used when searching for horizontal lines or lines of equal angle. When a line is recognized, its angle is measured and categorized as "horizontal," "posSlope," "vertical," or "negSlope." The category is used as the key for a hash map whose values are a linked list of shapes. This allows constant-time retrieval of a list of all of the lines in a particular orientation. We also want to find parallel lines, so exact angles are used to add the shape to a sorted map list. This allows for a logarithmic-time retrieval of the list of lines that are close to a particular angle. Since it is faster to retrieve shapes by their predefined category (e.g., "vertical") than by a angle range (e.g., 75-105 degree angles), the system chooses to do so whenever appropriate.

Each shape and its accessible components are processing and indexed for: the components' possible name, type, angle, x, y, min-x, min-y, max-x, max-y, area, height, width, and length. Appendix E shows what properties have been indexed after the drawing of both a line and then an arrow to provide further details.

Domain-dependent Shape Formation

Once properties are computed and indexing has been done, the system tries to see whether a higher-level shape can be made from this new shape and shapes already on the screen. We need to check whether this new shape can be a part of any of the shapes defined in the domain description. For each shape in the domain, the system assigns the new shape to each possible slot component (i.e., each subshape). If there are n domain shapes, and each shape $\mathcal{S}$ is composed of m components ($\mathcal{C}_1$ to $\mathcal{C}_m$), then the just processed shape is assigned to each slot separately in different shape templates. Figure 5-8 shows an example. A newly interpreted line is added to the system. The system checks to see whether the newly interpreted line can be used to create any of the shapes in the domain. (In this example, we are checking only the domain shape OpenArrow.) The system creates three templates, one for each component of the OpenArrow of the correct type (in this case, a line), assigning the just processed line to a different component to see whether an OpenArrow can be formed with that component.[1]

[1]We state that there are three templates, for explanation simplicity, but, in actuality, when a single line is drawn, the system creates six arrow templates. The recognition creates two copies of

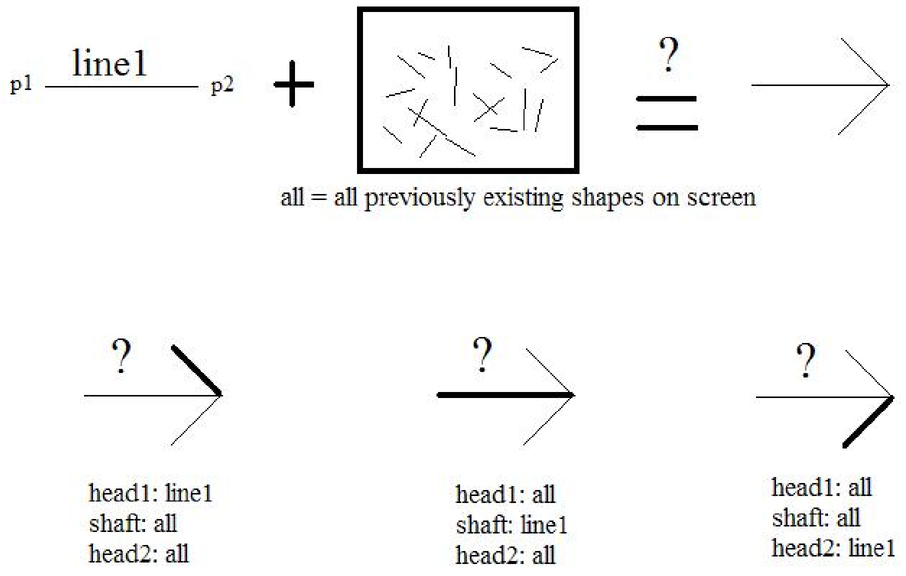


Figure 5-8: For each shape added to the system, the recognition system checks whether the new shape can be a piece of any of the higher-level shapes recognizable by the domain. This figure shows a new line added to the recognition system. The system checks whether that line, combined with any of the shapes on the screen, can form an arrow (one of the higher-level shapes in the domain). The system checks whether the new line can be any of the three lines that compose an arrow and makes a template for each of these possibilities.

The system then computes the function $\mathcal{L}_{ij} = f(S_i, C_j)$, which returns a list $\mathcal{L}_{ij}$ of shapes of type $\mathcal{S}_i$ and the components that make up these shapes, which can be formed with the just processed shape assigned to component $\mathcal{C}_j$. For example, if the domain description includes 10 shape descriptions, and OpenArrow is the third description, then S_3 = OpenArrow (as shown in Figure 5-9. An arrow has three slots (one for each line). If the system puts the recently drawn shape into slot 1, then C_1 = shaft. Thus, $\mathcal{L}_{ij}$ returns a list of all of the possible OpenArrows, with the most recently drawn shape acting as the shaft of the stroke. The length of $\mathcal{L}_{ij}$ may be 0 (no such interpretations are possible), 1 (one interpretation is possible), or > 1 (multiple interpretations are possible; see Figure 5-10).

every line, one in each direction. (i.e., the second is equal to the first, with the endpoints flipped.) Each directional line is assigned, one at a time, to each component of three arrow templates. This is actually not a phenomenon applied specifically to lines, but, any group of multiple interpretations using a single subshape.

```
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
    equalLength head1 head2
    acuteMeet head1 shaft
    acuteMeet shaft head2
...
```

Figure 5-9: The description for an arrow with an open head

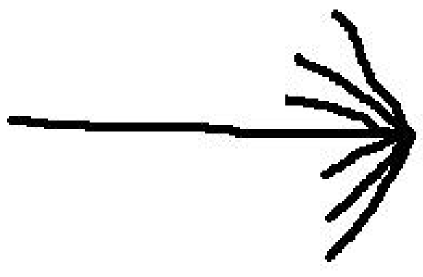


Figure 5-10: Multiple OpenArrow interpretations are possible using the center stroke as the shaft.

The new shape can be placed in any slot in any template. $\mathbb{P}$ is the union of all of the possible shapes formed with the new shape.

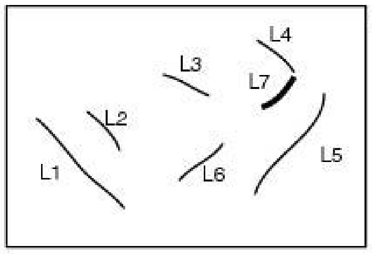
$$\mathbb{P} = \bigcup_{i=1}^{n} \bigcup_{j=1}^{m} (\mathcal{L}_{ij} = f(\mathcal{S}_i, \mathcal{C}_j))$$

Each template currently has only one slot filled (with the new shape). To compute $\mathcal{L}_{ij} = f(S_i, C_j)$, the system assigns all of the other shapes on the screen to the other slots on the template so that each slot on the template holds the possibilities for that slot.

The next stage is to reduce the possibilities for each slot. This is done from the indexing data structures that were created previously. The system holds a list of unchecked constraints. In order to remove a component from the list of possible components in a slot, we have to be sure a shape cannot be formed using that component. If only one of the component slots referenced by the constraint contains more than one possible component, then we can determine, for certain, which components satisfy that constraint, and are allowed in the final shape composition. Let $\mathcal{T}_i$ be the list of constraints from $\mathcal{S}_i$ that are not yet solved. Let $\mathcal{O}_t$ be the number of slots for constraint t with more than one component in the possibilities list. For each constraint with only one slot (or no slots) with more than one component ($\mathcal{O}_t <= 1$), we compute $g(t)$. $g(t)$ removes all of the shapes from the appropriate slot that would make constraint t false, and then removes constraint t from the list of unsolved constraints $\mathcal{T}_i$. Figure 5-11 walks through an example.

Finding the constraints that do not satisfy t is a quick process when using the indexing tables formed above. For each constraint, since $\mathcal{O}_t <= 1$, only one slot is being refined at a time. Thus, the system computes the numerical value(s) for that slot that will satisfy the constraint. The system uses the indexing data structures to obtain a list of all of the shapes on the screen with that particular value(s) (e.g., for HORIZONTAL, it would retrieve all lines with an angle near 0). Then, the intersection of this list and the list of shapes in the slot is computed, and shapes that are not in the intersection are removed from the slot.

It is possible that all shapes are removed from the slot, which implies that this shape, $\mathcal{S}_i$, cannot be formed with the set of shapes in the slot, and all processing on that template is halted. This cycle is repeated until: 1) the template is determined impossible; 2) all of the constraints are solved, and each slot has only one shape in it; or 3) all of the remaining constraints have $\mathcal{O}_t > 1$, and the cycle is stuck (see the next paragraph for what happens). After each cycle, there are some slots that contain only one shape; consistency checking occurs as the system removes these shapes from all of the other slots to make sure the same shape is not used in multiple slots.



New line L7 added to screen

One Initial Template:
head1: L1, L2, L3, L4, L5, L6
shaft: L1, L2, L3, L4, L5, L6
head2: L7
(omitted here are duplicate flipped lines for ease of explanation)

1)
equalLength head1 head2
coincident head1.p1 head2.p1
coincident head1.p1 shaft.p1
acuteMeet head1 shaft
acuteMeet shaft head2
longer shaft head1

head1: L2, L3, L4, L6
shaft: L2, L3, L4, L6
head2: L7

2)
longer shaft head1
coincident head1.p1 head2.p1
coincident head1.p1 shaft.p1
acuteMeet head1 shaft
acuteMeet shaft head2

head1: L2, L3, L4, L6
shaft: L2, L3, L4, L6
head2: L7

3)
longer shaft head1
coincident head1.p1 head2.p1
coincident head1.p1 shaft.p1
acuteMeet head1 shaft
acuteMeet shaft head2

head1: L4
shaft: L2, L3, L4, L6
head2: L7

head1: L4
shaft: L2, L3,, L6
head2: L7

4)
longer shaft head1
coincident head1.p1 shaft.p1
acuteMeet head1 shaft
acuteMeet shaft head2

head1: L4
shaft:empty (no arrow)
head2: L7

Figure 5-11: Line L7 has been added to the screen where previous shapes L1, L2, L3, L4, L5, and L6 already exist. The top right shows the initial template when assigning the new line (with a particular orientation) to the HEAD2 slot of the arrow. Note that all other shapes on the screen are added. The system then attempts to satisfy each of the constraints, removing shapes that do not satisfy a constraint from the template. 1) The system tries to satisfy the constraint EQUALLENGTH and removes all shapes in the HEAD1 slot of the template that are not of equal length to L7. The EQUALLENGTH constraint is now completed and removed from the list of constraints yet to be satisfied. 2) The system attempts to satisfy the LONGER constraint, but since both arguments have more than one shape in the slot, the constraint is postponed. 3) The system tries to satisfy the COINCIDENT constraint and removes L2, L3, and L5 from the HEAD1 slot. Because L4 is now the only possibility for the HEAD1 slot, it is removed from the SHAFT slot, since the same shape cannot be used for both slots. 4) The system tries to satisfy the second COINCIDENT constraint, but since none of the shapes in the SHAFT slot can be coincident with L4, the SHAFT slot is empty, and the system determines that an OPENARROW cannot be formed with the new shape filling in the HEAD1 slot.

It is possible that all of the remaining unsolved constraints have $\mathcal{O}_t > 1$. In this case, the system branches the template, choosing the slot with more than one remaining possible assignment that has the fewest such possible assignments. It makes a new copy of the template for each of the possible assignments for that slot, placing only one in each template, then continues trying to solve the remaining unsatisfied constraints on each of the templates.

This branching process can, of course, cause its own exponential slowdown. The system's near real-time performance results from the fact that 1) branching does not happen often because most of the shapes on the screen do not obey the required constraints, and, thus, many shapes are removed from the possibility list at once. (Consider, for example, the COINCIDENT constraint. It is uncommon for many shapes to be drawn at the same location, so many possibilities are removed simultaneously from the possibilities list.) And, 2) even in the worst case, where every query results in a branching factor, the process of checking the constraints is a small proportion of the overall running time. (See the Results section below.) This is because the exponential part of the algorithm performs only list retrievals (which has been made fast with the use of sorted hash maps and other data structures) and list intersections.

At the end of this stage, we have a list $\mathbb{P}$ of all of the shape interpretations and their components. All interpretations are added to the recognition system, but a single interpretation is chosen for display. The system chooses to display the interpretation that is composed of the largest number of primitive shapes (i.e., the solution that accounts for more data). Creating shapes with the largest number of primitive shapes also results in fewer more-complicated shapes filling the screen. For example, in Figure 5-12, we choose the square interpretations rather than the arrow for display purposes, as the square accounts for four primitive shapes, simplifying the diagram to only two higher level shapes, whereas the arrow interpretation accounts for only three lines, simplifying the diagram to three higher level shapes.

If the recognition process finds multiple interpretations (i.e., $|\mathbb{P}| > 1$), the system

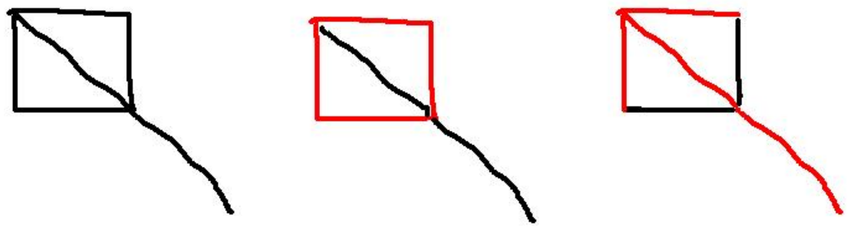


Figure 5-12: The left shows the originally drawn strokes. The middle interpretation is made up of a line and a square (in red and cleaned). The right interpretation is made up of an arrow (in red and cleaned) and two lines.

adds both interpretations to the recognition system for use in finding higher-level shapes. The system uses the bookkeeping system described earlier to ensure that only one final interpretation is chosen for display for each shape with a common subshape. The system adds all interpretations to the recognition system for use in finding higher-level shapes. The system uses the bookkeeping system described earlier to ensure that only one final interpretation is ultimately chosen for display.

5.2.5 Algorithm Results

In a test on a tablet PC with 1GB of RAM and a 1.73 GHz processor, the system recognized a resistor containing six lines in less than a second, with 189 other shapes (besides the six lines creating the resistor) on the screen (see Figure 5-13), 3 of which were higher-level shapes such as resistors, and the other 186 were random lines on the screen. We added many random lines on the screen to provide many possible recognition choices as a sort of stress test.

If the user draws many strokes quickly, the system can slow down because there is a constant amount of time necessary to preprocess each stroke. We analyzed the running time of the recognition system, using JProbe [182], and determined that, with many unrecognized shapes on the screen, approximately 74% of the recognition time was spent breaking down the strokes into lines. The next greatest amount of time was spent indexing the lines drawn; each shape took a constant amount of time in

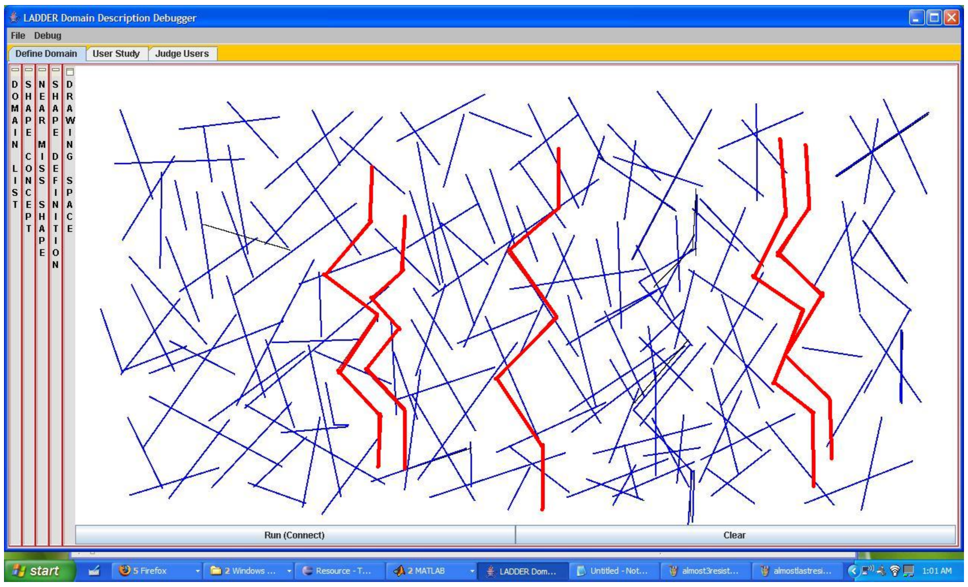


Figure 5-13: Recognition Stress Test. The resistors are shown in red and made bold for black and white viewing.

the number of shapes on the screen to compute the property values and a logarithmic amount time to insert in to the appropriate data structure. A very small portion of time was used to do the actual recognition, even though the last portion is exponential in the number of strokes on the screen. Because of our indexing, the recognition portion takes a small amount of time, with little to no constraint calculation, as the system was only performing list comparisons. As a result, the system still reacts in what can be considered close to real-time, even with 186 shapes on the screen.

5.3 Editing

A stroke may be intended as an editing gesture, rather than a drawing gesture. If an editing gesture such as click-and-hold (tap-and-hold the pen on the screen) or double-click (double-tap) occurs, the system checks to see whether 1) an editing gesture using that trigger is defined for any shape, and if 2) the mouse is over the shape for which the gesture is so defined. If so, the drawing gesture is short-circuited and the editing gesture takes over. The editing gesture takes precedence over the drawing gesture, so any ink created during the gesture is not used as part of a drawing gesture. For instance, instead of registering the ink laid down during click-and-hold as the first few points of a new drawing stroke, the ink is removed from the screen, and the shape can then be moved. Other triggers, such as SHAPE-OVER (a particular shape is drawn on top of another shape), 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, the SHAPE underneath is deleted (as is the CROSS).

5.4 Display

Section 4.4.1 discussed the various options for shape display. The shape exhibitor controls the displaying of the newly created shape and ensures that the components (e.g., the original strokes) are not drawn and only the abstract meaningful shape (e.g., arrow) itself is drawn. 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 whether an editing gesture is occurring, and 2) ensures that when a shape is moved or deleted, its components are moved or deleted with it. The shape exhibitor also keeps an original copy of each of the accessible components and aliases for use when scaling an object to ensure that precision is not lost after several scalings.

5.5 Connecting to Existing Systems

We included an API to allow developers of a sketch system to connect to an existing knowledge system, such as a CAD or CASE tool. With this API, the domain-specific recognition system can communicate with a back-end system, providing additional functionality, e.g., checking the diagram for inconsistencies, running the diagram to see whether it works as the sketcher intended, etc. Thus far, we have used the API to connect to Rational Rose^{TM}(for UML class diagrams as in Figure 1-1(c)), Working Model (mechanical engineering simulations), Spice (for electrical circuit analysis), as well as our own systems (for finite state machines and course of action diagrams).

Connecting to another program is currently quite simple, although it does require that the system be recompiled before it works. The developer needs to create a Java file that extends AppLink.java. The Java file must be the same name as the domain. For example, to build a back-end for a finite state machine sketch system in which the '.ldl' file is called FiniteState.ldl, the java file must be called FiniteState.java. The system uses reflection to find the back-end java file. It runs as a separate thread that gets invoked when the run button is pressed.

Once the run button is pressed, the connect() method is invoked. The developer overrides this button to implement the back-end functionality. The AppLink API provides access to all of the current shapes on the screen (using getViewableShapes()), as well as the drawing panel itself (getDrawPanel()). The API also provides a few other methods for ease of use, such as setPauseColor(DrawnShape shape, Color color, int milli) which temporarily changes the color of a shape on the screen for a limited amount of time. (This is used in the finite state machine application listed in the appendix.) We plan to add other useful methods as more people use the system, and we see what methods would be useful to many different applications. (We hope this will help users more easily describe animations, which is part of the future work.)

Chapter 6 and Appendix B give concrete examples and their code for the usage of the API.

5.6 Limitations

This system uses a bottom-up recognition method. 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. This researcher tries to circumvent this problem by sending all interpretations created from the lower level processor, but, if the lower level processor cannot generate a particular shape, then the system cannot find it. Top-down recognition systems such as SketchRead [12] [7] [8] have a similar problem, in that they only initially send the best interpretation, and search for another possibility, if needed, but the interpretation must still be able to be generated from the lower level processor. In the future, it may be advantageous to add a top-down recognition process that tasks lower level recognizers to perhaps lower their thresholds.

Also, this research includes future plans to add the ability to register for sketch events so as to easily connect to the system using programming languages other than Java.

The indexing algorithm described in this document was limited to the geometric constraints and *LADDER* limitations described above. It would be useful to remove some of these limitations by combining the techniques presented in this document with those that have proved useful in work in reference to vision. By processing and indexing vision features used for recognition, and concurrently indexing on geometric properties, as described in this document, we can quickly access shapes that have the needed visual and geometric features. As a very simple example, vision recognition techniques can easily locate areas of high density ink, or shading, which we are currently not able to recognize using our geometric recognition techniques. Future research includes combining vision and sketch-based features to perform more robust recognition and, perhaps, recognize a larger class of objects.

Chapter 6

Results: Automatically Generated Sketch Systems

Sketch systems using this work have been defined for several of domains including mechanical engineering, UML class diagrams, finite state machines, course of action diagrams, circuit diagrams, flow charts, Graffiti^{TM}, alphabet, and tic tac toe. These sketch applications have been built through LADDER/GUILD by this researcher, master's students working with this researcher, classroom students in the Sketch Recognition course taught by this researcher, and by high school students during a community outreach workshop that this researcher instructed on sketch recognition. Figure 6-1 shows a selection of shapes that have been recognized using LADDER.

6.1 Tic Tac Toe

Several tic tac toe recognition systems have been automatically generated by this researcher and high school students. The tic tac toe games build recognize X's, O's, and the board. The pieces can be moved and re-recognized in their new location. Two shape groups process the winning conformation. The domain description for this domain is listed in Appendix B. Figures 6-2, 6-3, 6-4, and 6-5 show screen shots of the system in action.

6.2 UML Class Diagrams

GUILD was used to build a system to recognize UML class diagrams. The system has several capabilities:

- It recognizes the shapes in the domain from the *LADDER* descriptions.
- The system draws the different shapes in different colors to provide recognition feedback.
- All of the shapes (including the text) can all be automatically beautified to make the diagram more elegant.
- All of the classes and their building blocks (lines, rectangles, circles, class, interface) can be scaled to beautify the diagram.
- The arrows can be rubberbanded from either their head or their tail.
- Once the contextual classes are attached, a string label is attached to each of the arrows to help the designer remember what type of association it is. (This is not shown in the original strokes view.)

Figures 6-6 and 6-7 show messy and cleaned views, respectively, of the beginning of a UML class diagram. Figures 6-8 and 6-9 show messy and cleaned views, respectively, of the middle of a UML class diagram. Figures 6-10 and 6-11 show messy and cleaned views, respectively, of a finished UML class diagram.

The domain list and shape descriptions for this sketch recognition application are listed in Appendix B.

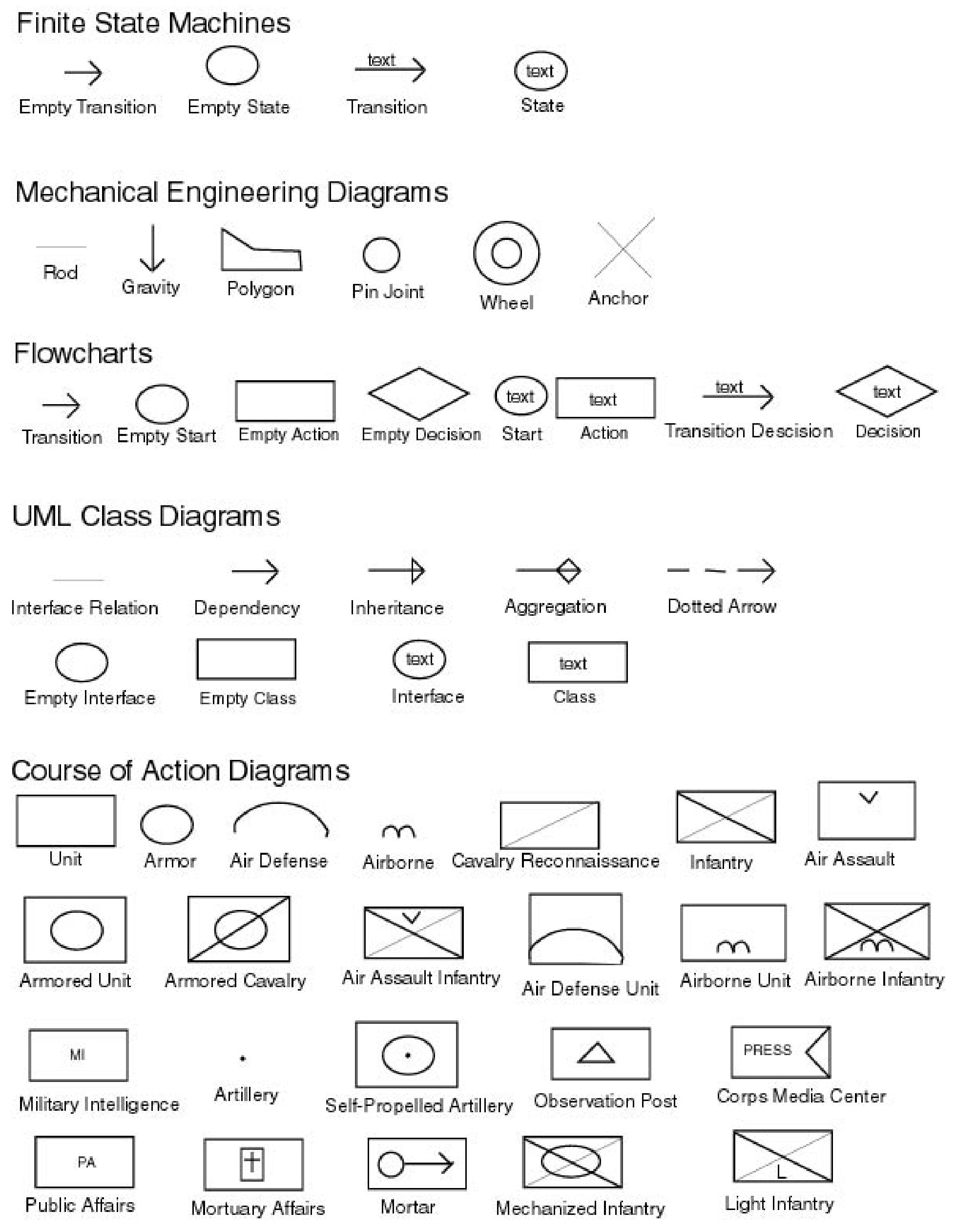


Figure 6-1: Variety of shapes and domains described and auto-generated.

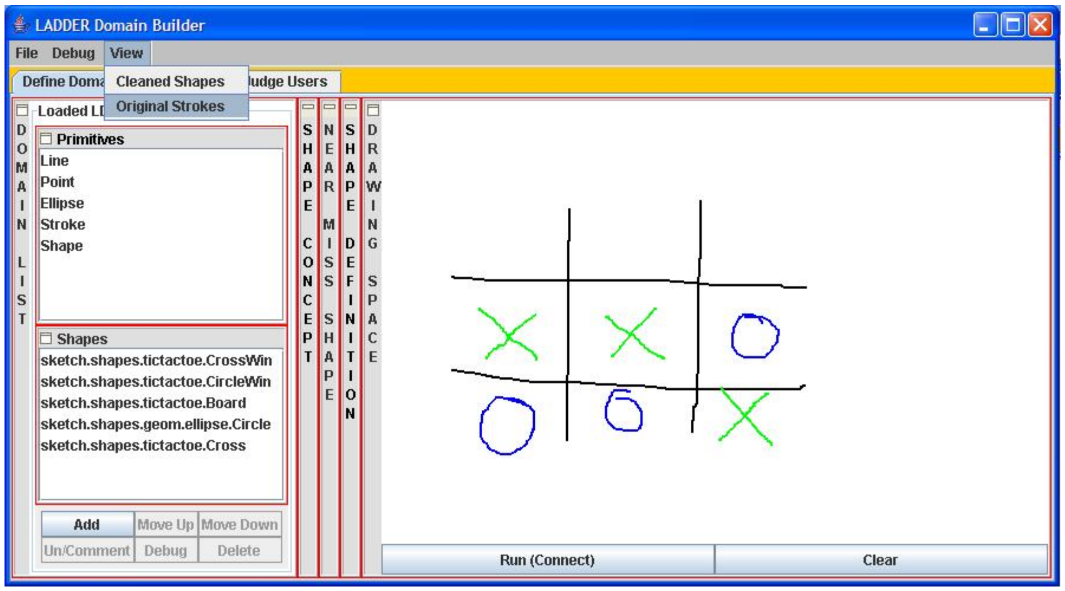


Figure 6-2: The original strokes of an unfinished game played in an automatically generated tic tac toe recognition system.

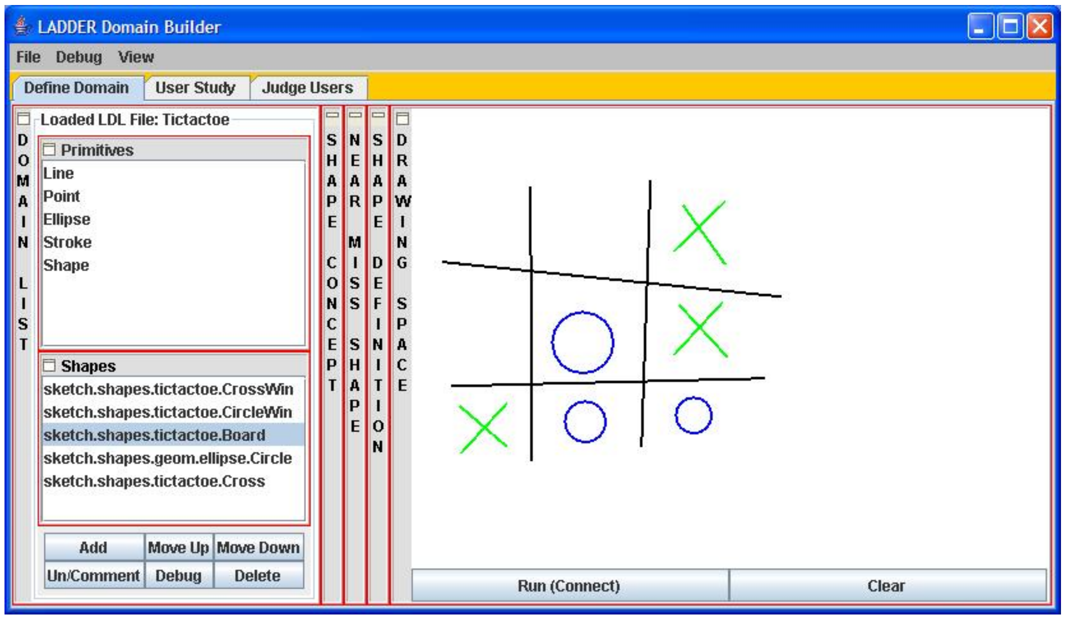


Figure 6-3: The cleaned strokes of an unfinished game played in an automatically generated tic tac toe recognition system.

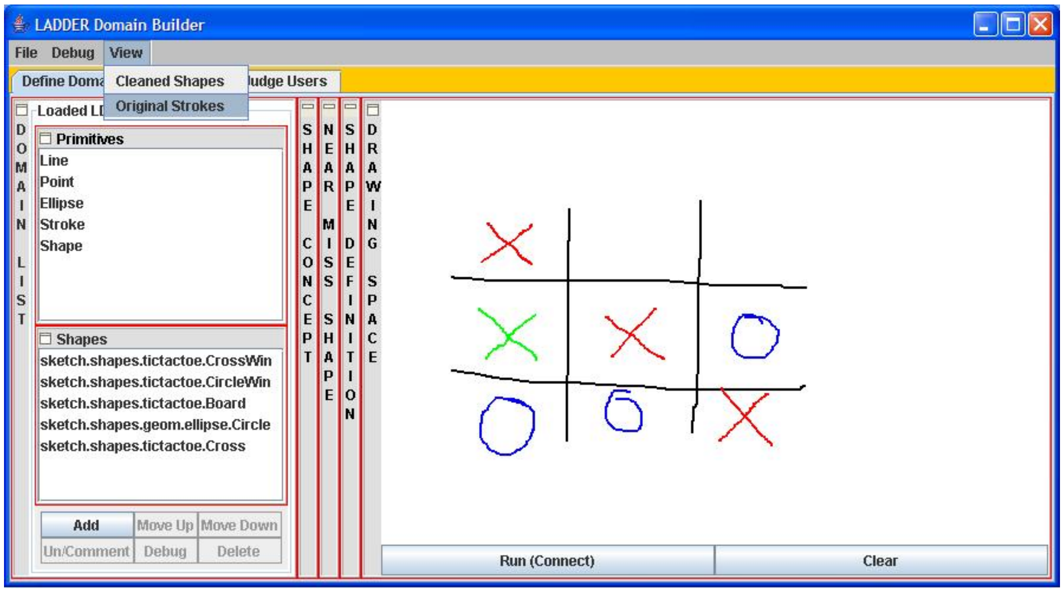


Figure 6-4: The original strokes of a finished game played in an automatically generated tic tac toe recognition system.

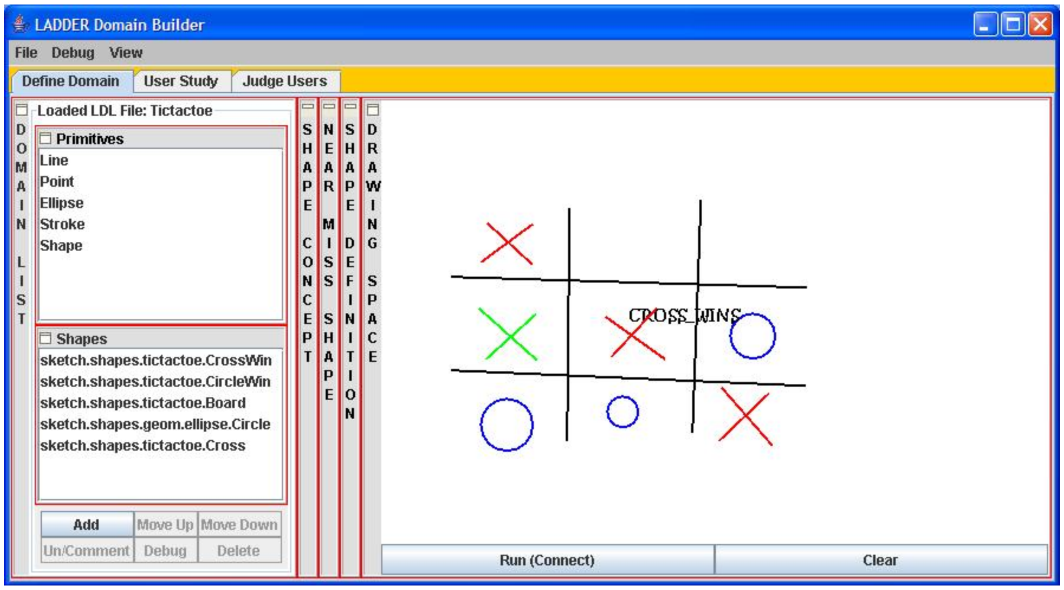


Figure 6-5: The cleaned strokes of a finished game played in an automatically generated tic tac toe recognition system.

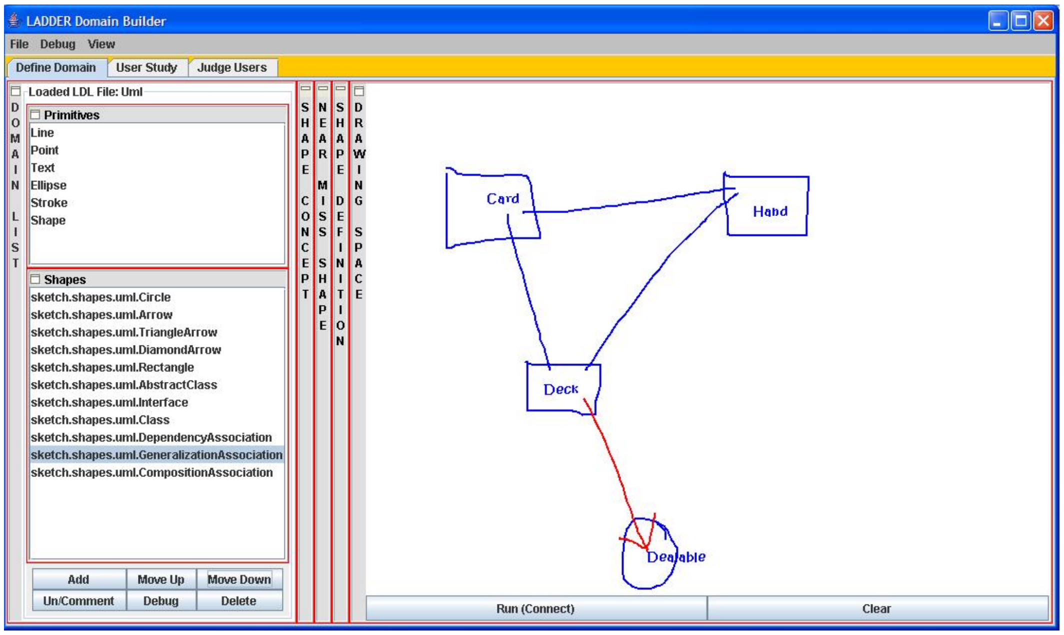


Figure 6-6: The original strokes at the beginning of a UML Class Diagram sketch.

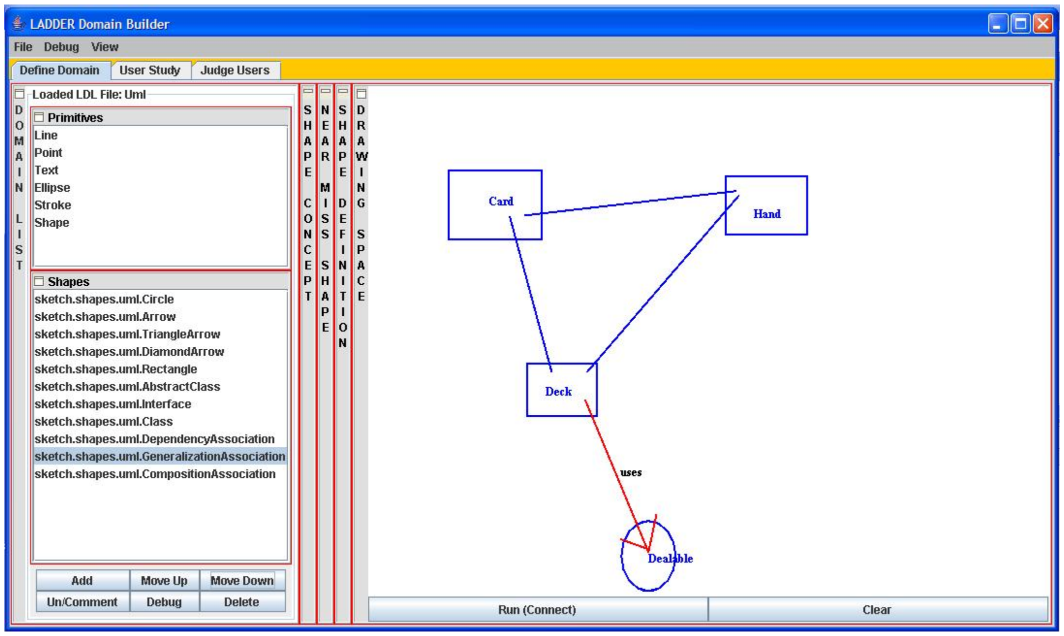


Figure 6-7: The cleaned up strokes of Figure 6-6

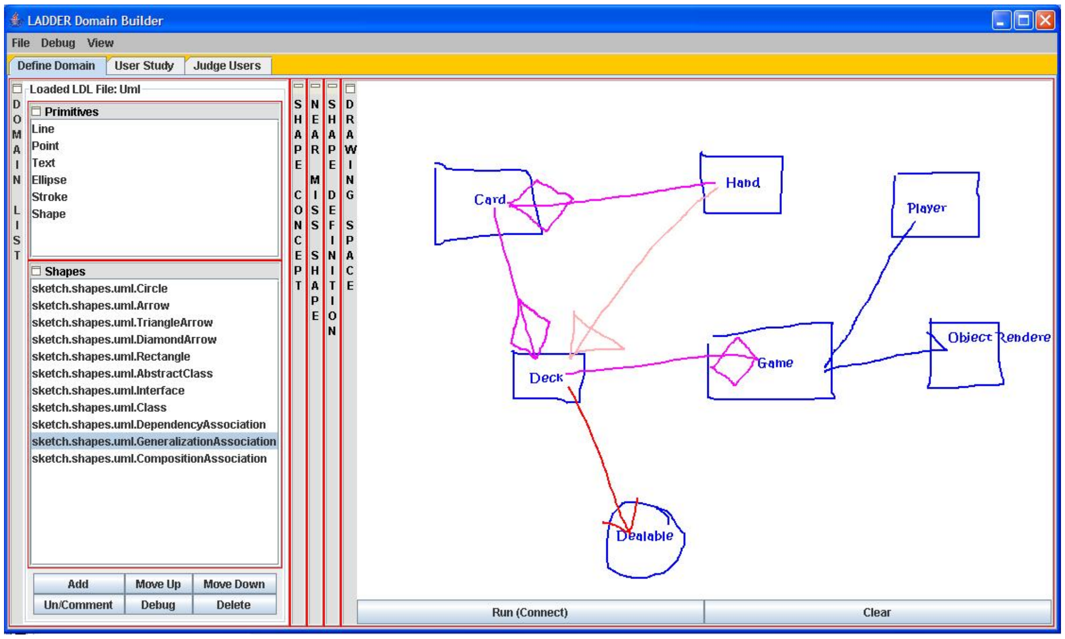


Figure 6-8: The original strokes during the middle of a UML Class Diagram sketch. Note that the some of the class have been moved or scaled to better fit the text. Also note that the different arrow types are denoted with different colors.

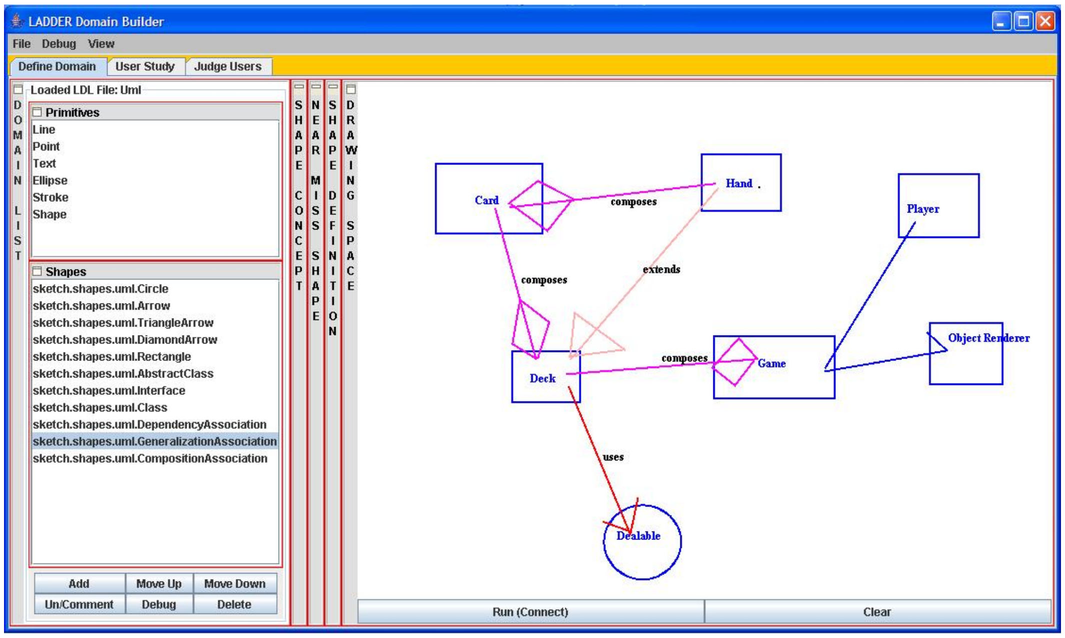


Figure 6-9: The cleaned up strokes of Figure 6-8. Note that in the cleaned up version text strings have been added to the arrow as a helpful benefit to the designer.

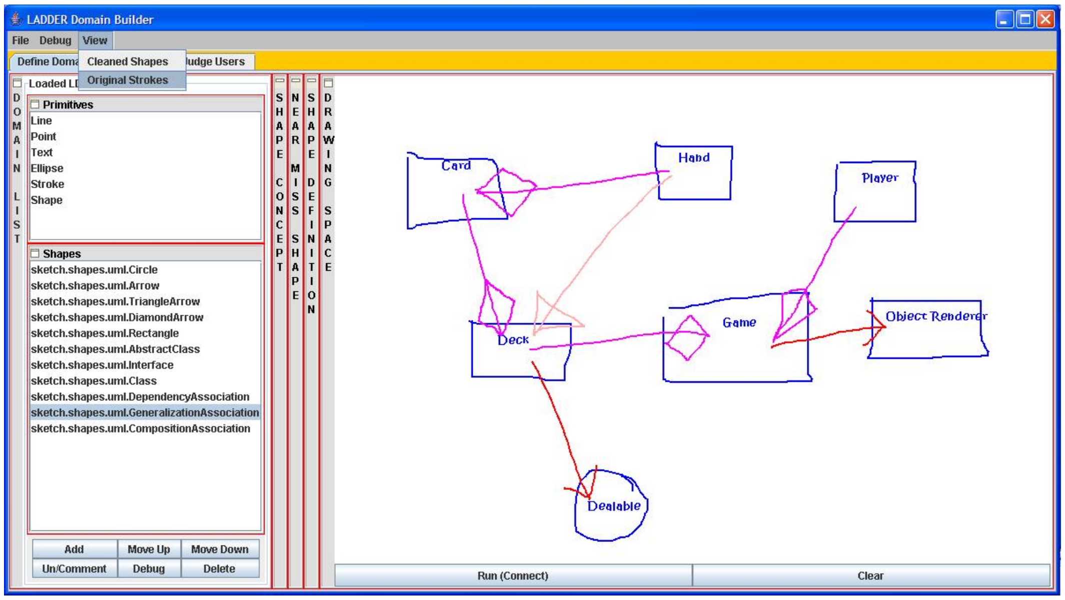


Figure 6-10: The original strokes of the final UML class diagram.

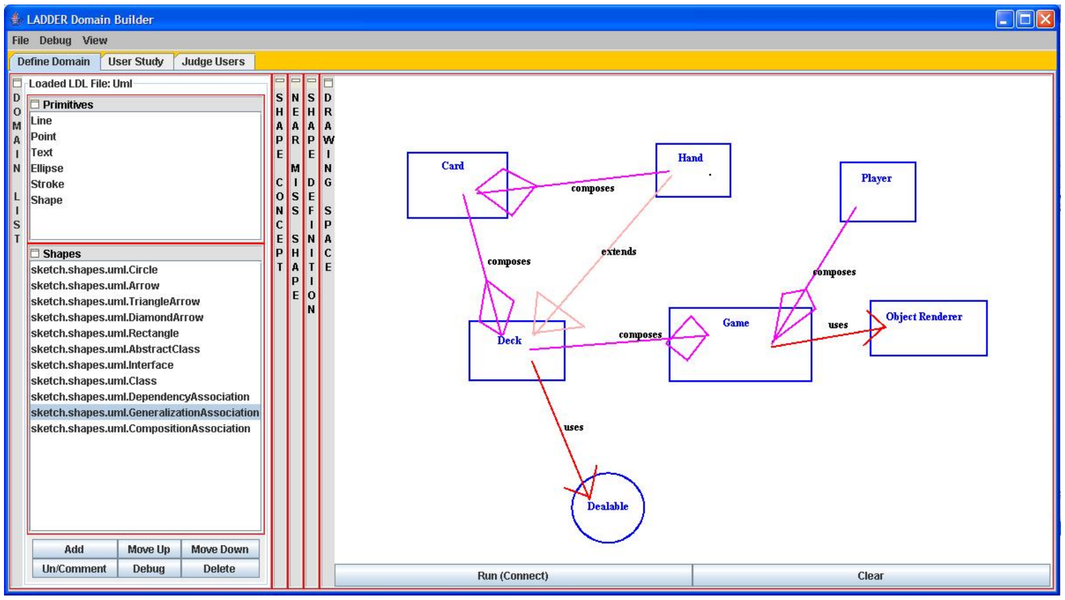
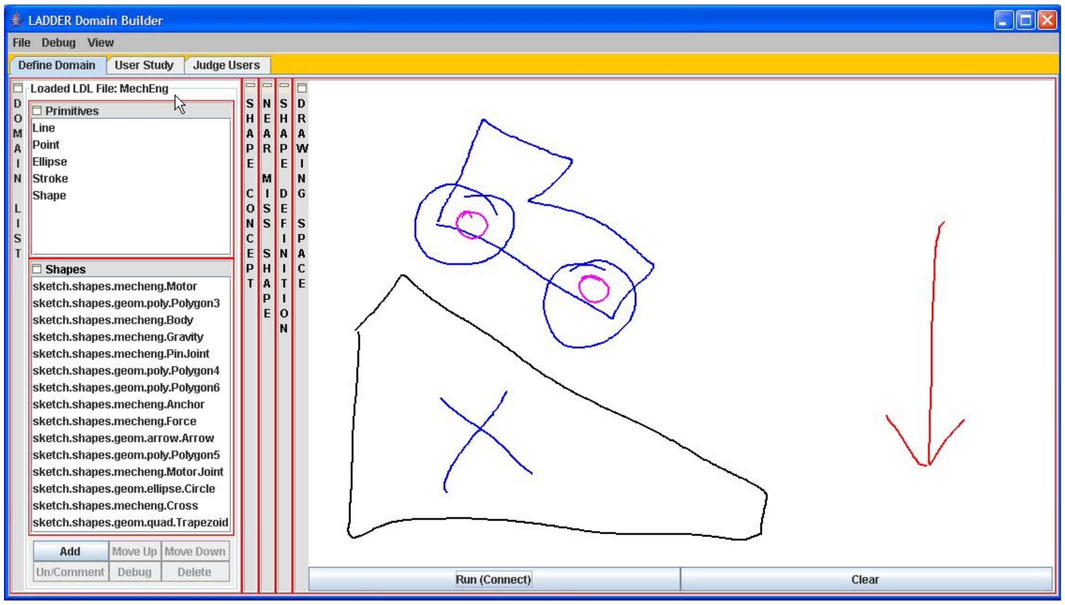


Figure 6-11: The cleaned up strokes of Figure 6-10.

6.3 Mechanical Engineering

Figures 6-12 through 6-17 show an automatically generated recognition system for mechanical engineering diagrams.


Loaded LDL File: MechEng

Primitives
Line
Point
Ellipse
Stroke
Shape

Shapes
sketch.shapes.mecheng.Motor
sketch.shapes.geom.poly.Polygon3
sketch.shapes.mecheng.Body
sketch.shapes.mecheng.Gravity
sketch.shapes.mecheng.PinJoint
sketch.shapes.geom.poly.Polygon4
sketch.shapes.geom.poly.Polygon6
sketch.shapes.mecheng.Anchor
sketch.shapes.mecheng.Force
sketch.shapes.geom.arrow.Arrow
sketch.shapes.geom.poly.Polygon5
sketch.shapes.mecheng.MotorJoint
sketch.shapes.geom.ellipse.Circle
sketch.shapes.mecheng.Cross
sketch.shapes.geom.quad.Trapezoid

Figure 6-12: Auto-generated mechanical engineering interface

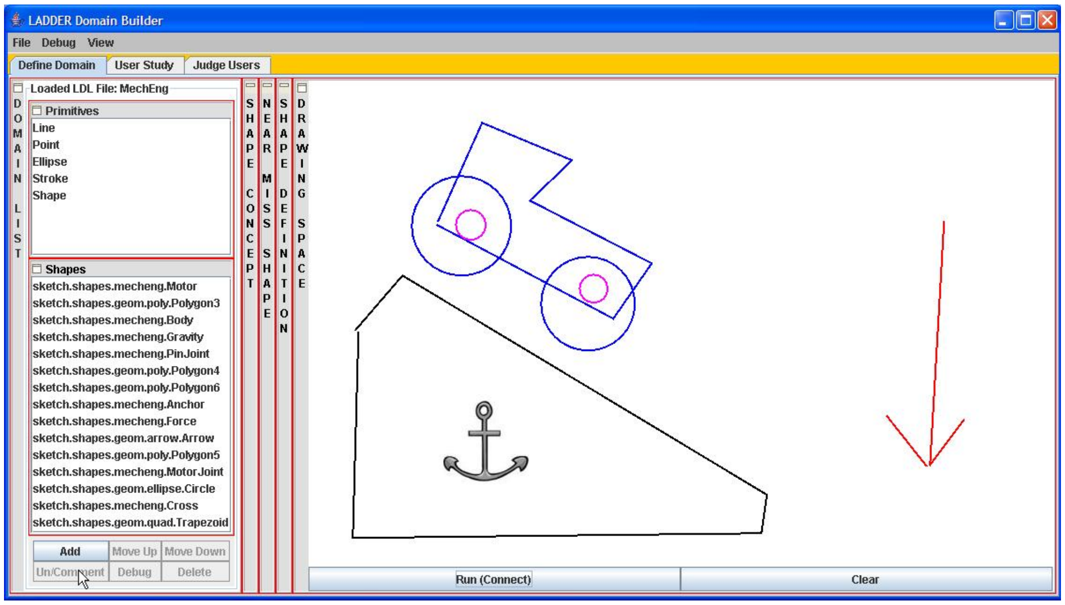

LADDER Domain Builder
File Debug View
Define Domain User Study Judge Users
Loaded LDL File: MechEng
DOMAIN LIST
Primitives
Line
Point
Ellipse
Stroke
Shape
Shapes
sketch.shapes.mecheng.Motor
sketch.shapes.geom.poly.Polygon3
sketch.shapes.mecheng.Body
sketch.shapes.mecheng.Gravity
sketch.shapes.mecheng.PinJoint
sketch.shapes.geom.poly.Polygon4
sketch.shapes.geom.poly.Polygon6
sketch.shapes.mecheng.Anchor
sketch.shapes.mecheng.Force
sketch.shapes.geom.arrow.Arrow
sketch.shapes.geom.poly.Polygon5
sketch.shapes.mecheng.MotorJoint
sketch.shapes.geom.ellipse.Circle
sketch.shapes.mecheng.Cross
sketch.shapes.geom.quad.Trapezoid
Add Move Up Move Down
Un/Comment Debug Delete
SHAPE CONCEPT
NEAR MISS SHAPE
SHAPE DEFINITION
DRAWING SPACE
Run (Connect)
Clear

Figure 6-13: Auto-generated mechanical engineering interface

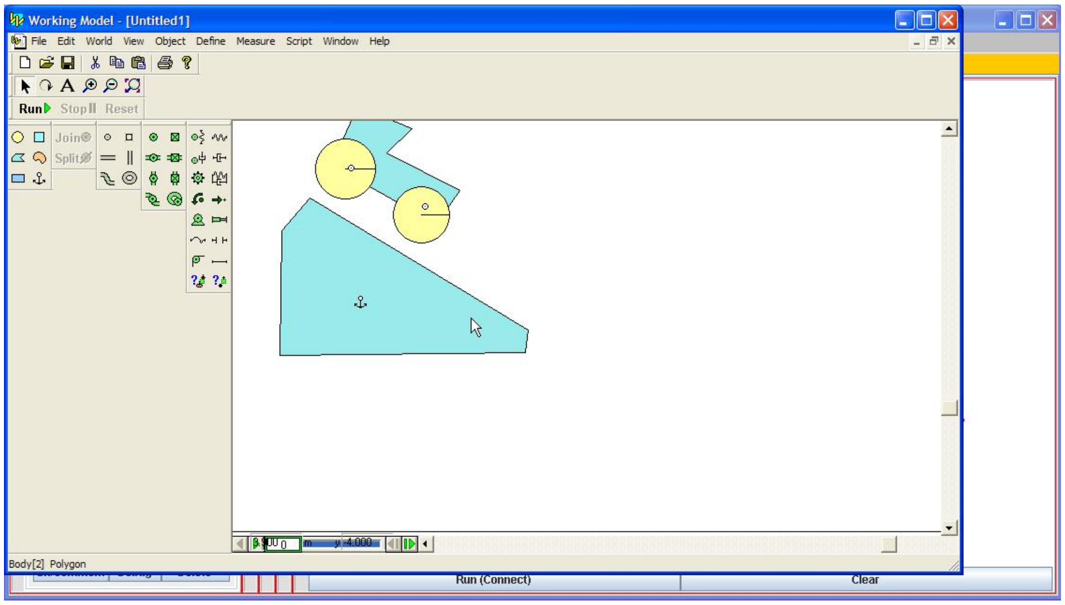

Working Model - [Untitled1]
File Edit World View Object Define Measure Script Window Help
Run Stop Reset
Join Split
Body[2] Polygon
Run (Connect)
Clear

Figure 6-14: Auto-generated mechanical engineering interface

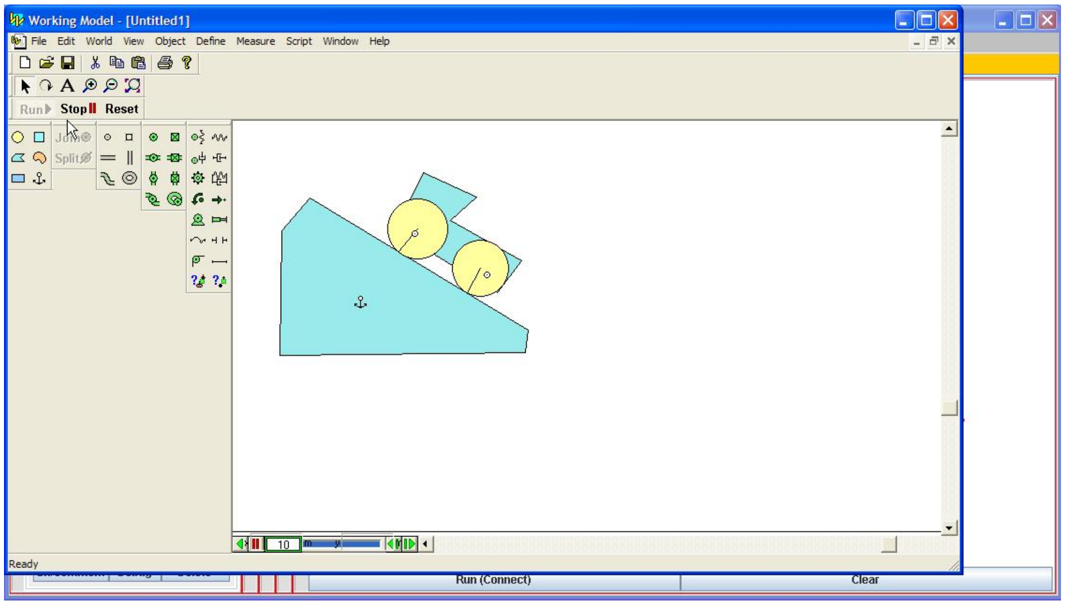

Working Model - [Untitled1]
File Edit World View Object Define Measure Script Window Help
Run Stop Reset
Join Split
10
Ready
Run (Connect)
Clear

Figure 6-15: Auto-generated mechanical engineering interface

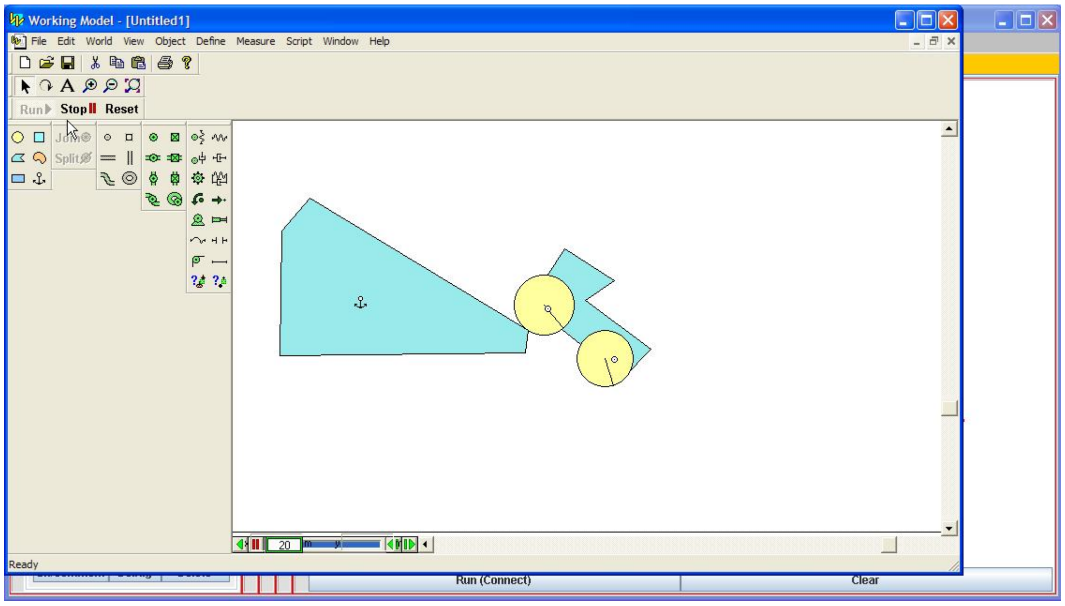

Working Model - [Untitled1]
File Edit World View Object Define Measure Script Window Help
Run Stop Reset
Join Split
20
Ready
Run (Connect)
Clear

Figure 6-16: Auto-generated mechanical engineering interface

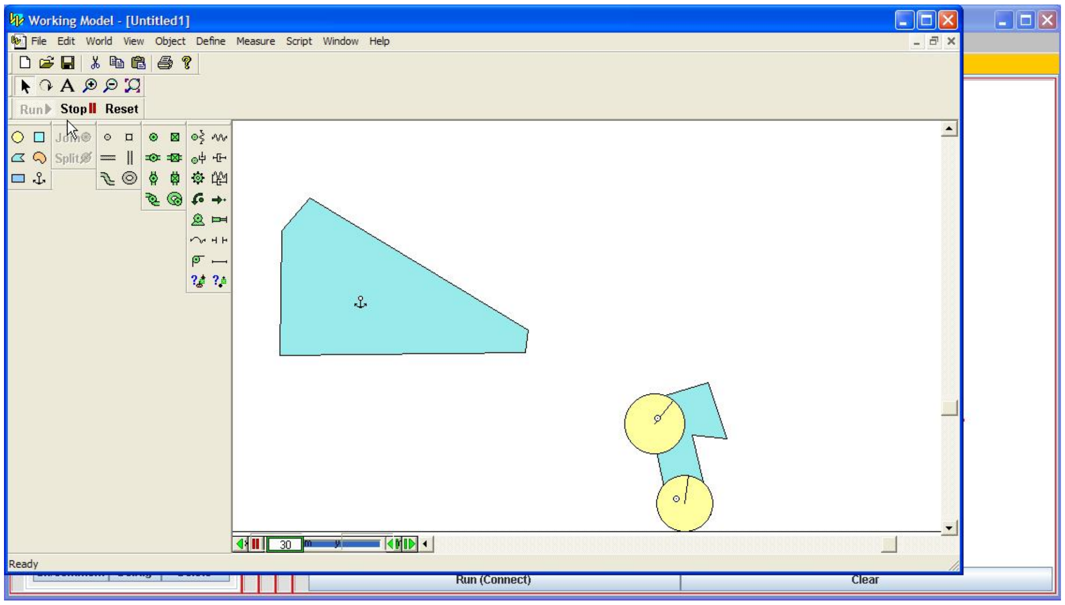

Working Model - [Untitled1]
File Edit World View Object Define Measure Script Window Help
Run Stop Reset
Join
Split
30
Ready
Run (Connect)
Clear

Figure 6-17: Auto-generated mechanical engineering interface

6.4 Flow Charts

Figure 6-18 shows an automatically generated recognition system for flow charts.

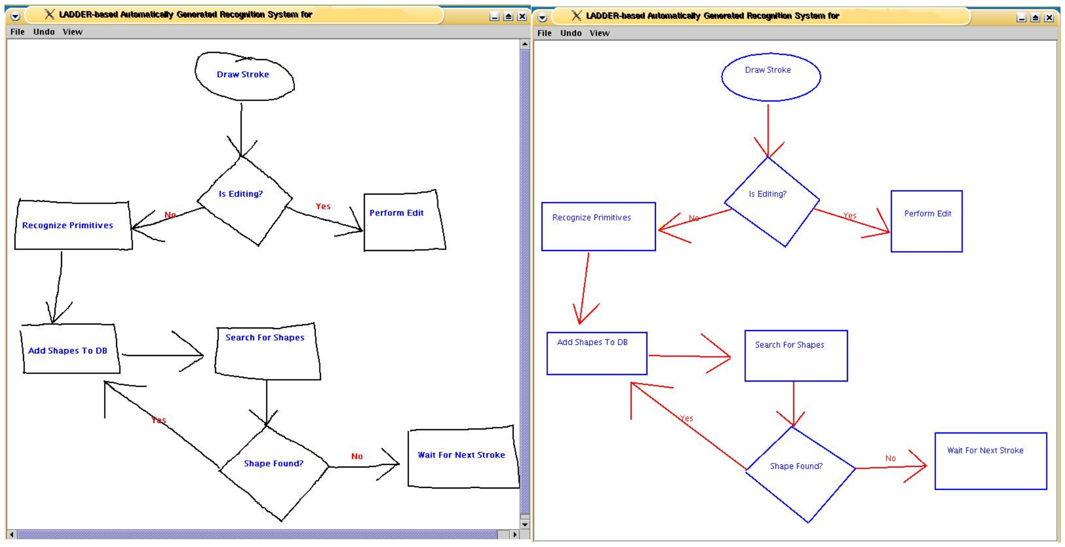


Figure 6-18: Auto-generated flowchart interface

6.5 Finite State Machines

Figures 6-19 through 6-31 show an automatically generated recognition system for finite state machines.

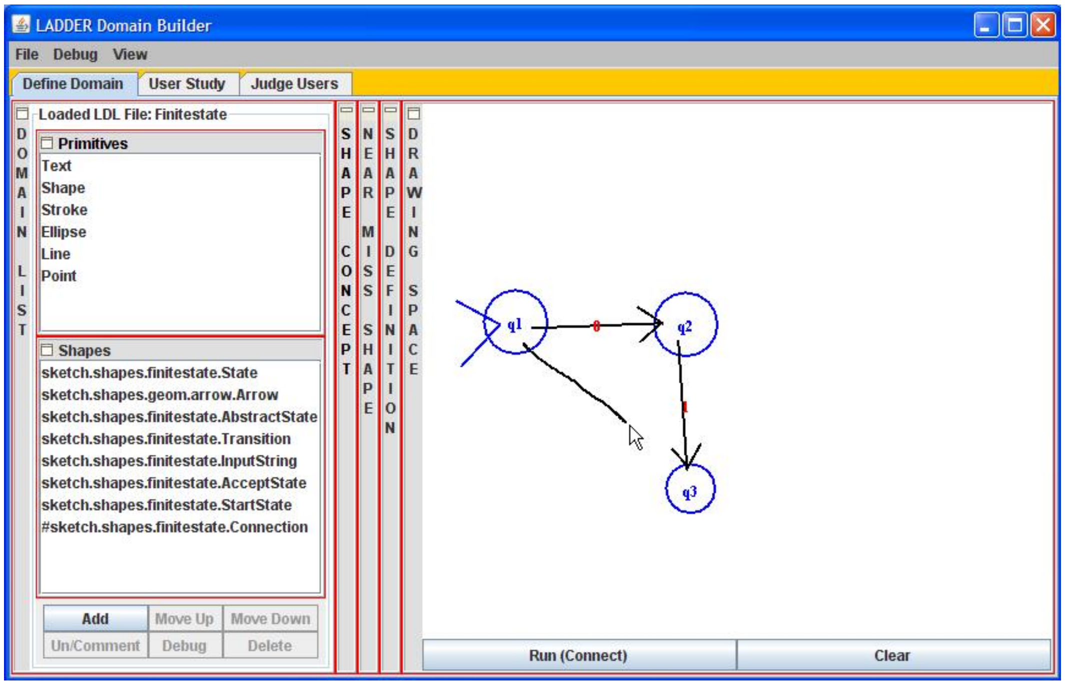

LADDER Domain Builder
File Debug View
Define Domain User Study Judge Users
Loaded LDL File: Finitestate
DOMAIN LIST
Primitives
Text
Shape
Stroke
Ellipse
Line
Point
Shapes
sketch.shapes.finitestate.State
sketch.shapes.geom.arrow.Arrow
sketch.shapes.finitestate.AbstractState
sketch.shapes.finitestate.Transition
sketch.shapes.finitestate.InputString
sketch.shapes.finitestate.AcceptState
sketch.shapes.finitestate.StartState
#sketch.shapes.finitestate.Connection
Add Move Up Move Down
Un/Comment Debug Delete
SHAPE CONCEPT
NEAR MISS SHAPE
SHAPE DEFINITION
DRAWING SPACE
q1
0
q2
1
q3
Run (Connect)
Clear

Figure 6-19: Drawing a finite state machine.

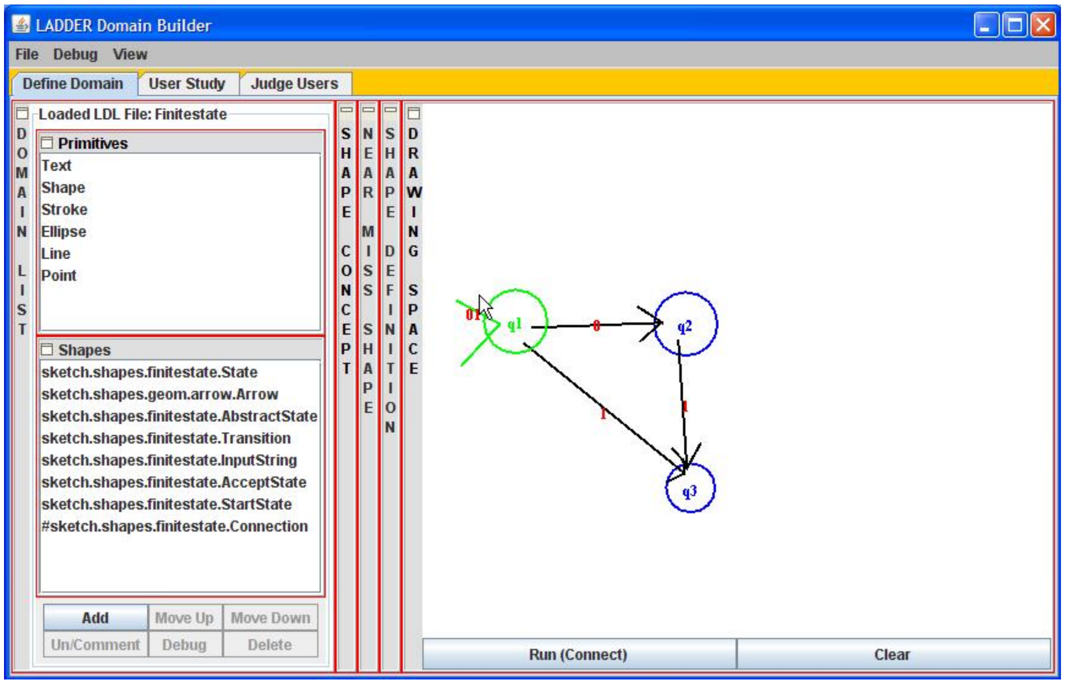

LADDER Domain Builder
File Debug View
Define Domain User Study Judge Users
Loaded LDL File: Finitestate
DOMAIN LIST
Primitives
Text
Shape
Stroke
Ellipse
Line
Point
Shapes
sketch.shapes.finitestate.State
sketch.shapes.geom.arrow.Arrow
sketch.shapes.finitestate.AbstractState
sketch.shapes.finitestate.Transition
sketch.shapes.finitestate.InputString
sketch.shapes.finitestate.AcceptState
sketch.shapes.finitestate.StartState
#sketch.shapes.finitestate.Connection
Add Move Up Move Down
Un/Comment Debug Delete
SHAPE CONCEPT
NEAR MISS SHAPE
SHAPE DEFINITION
DRAWING SPACE
01
q1
0
q2
1
1
q3
Run (Connect)
Clear

Figure 6-20: Adding an input string.

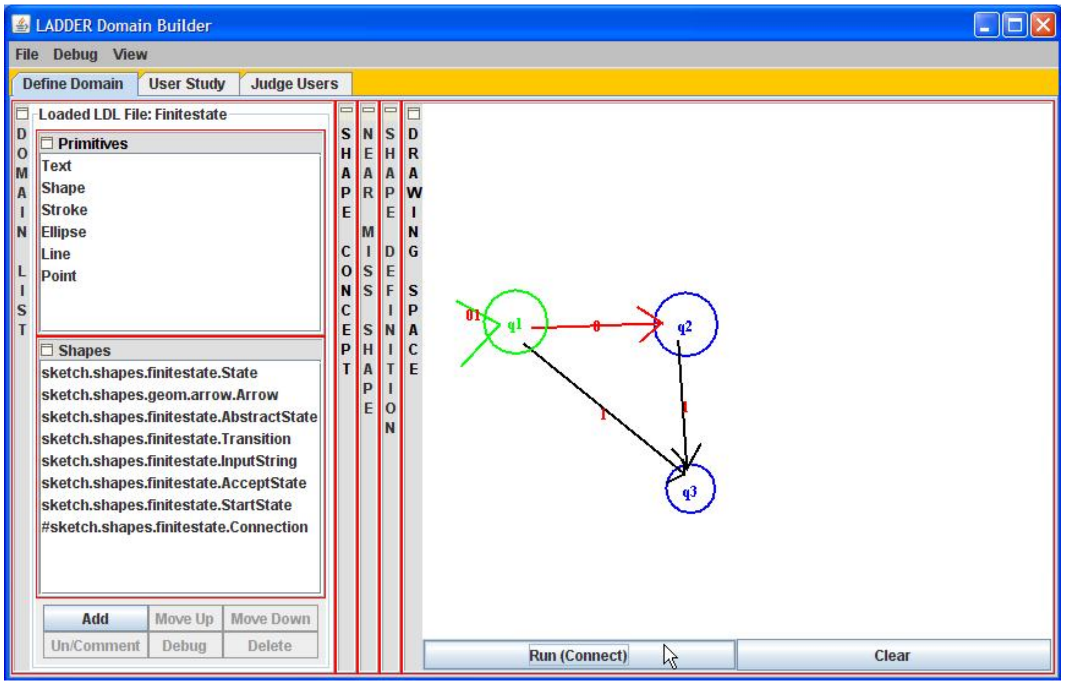

LADDER Domain Builder
File Debug View
Define Domain
User Study
Judge Users
Loaded LDL File: Finitestate
Primitives
Text
Shape
Stroke
Ellipse
Line
Point
Shapes
sketch.shapes.finitestate.State
sketch.shapes.geom.arrow.Arrow
sketch.shapes.finitestate.AbstractState
sketch.shapes.finitestate.Transition
sketch.shapes.finitestate.InputString
sketch.shapes.finitestate.AcceptState
sketch.shapes.finitestate.StartState
#sketch.shapes.finitestate.Connection
Add
Move Up
Move Down
Un/Comment
Debug
Delete
DOMAIN LIST
SHAPE CONCEPT
NEAR MISS SHAPE
SHAPE DEFINITION
DRAWING SPACE
01
q1
0
q2
1
1
q3
Run (Connect)
Clear

Figure 6-21: Testing the string on the FSM.

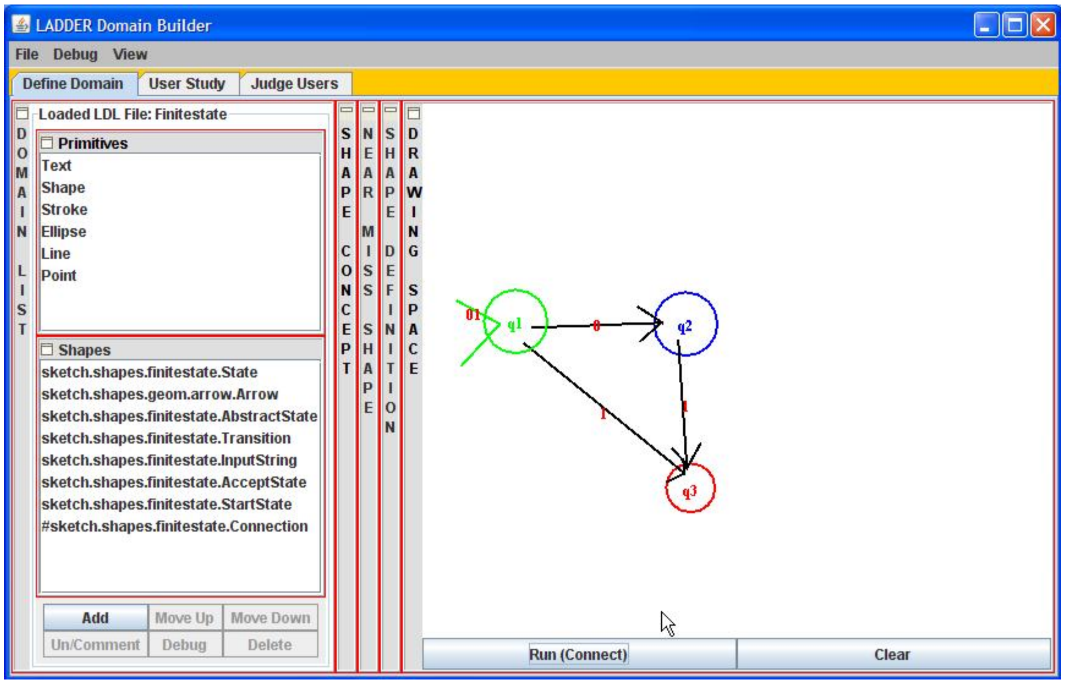
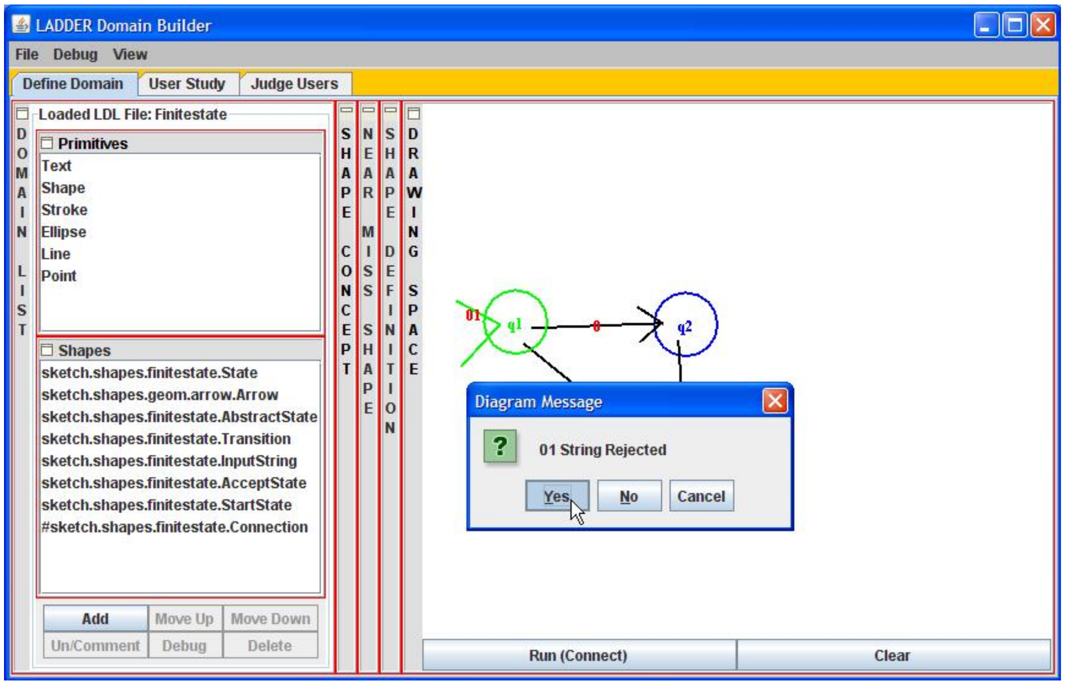

LADDER Domain Builder
File Debug View
Define Domain
User Study
Judge Users
Loaded LDL File: Finitestate
Primitives
Text
Shape
Stroke
Ellipse
Line
Point
Shapes
sketch.shapes.finitestate.State
sketch.shapes.geom.arrow.Arrow
sketch.shapes.finitestate.AbstractState
sketch.shapes.finitestate.Transition
sketch.shapes.finitestate.InputString
sketch.shapes.finitestate.AcceptState
sketch.shapes.finitestate.StartState
#sketch.shapes.finitestate.Connection
Add
Move Up
Move Down
Un/Comment
Debug
Delete
DOMAIN LIST
SHAPE CONCEPT
NEAR MISS SHAPE
SHAPE DEFINITION
DRAWING SPACE
01
q1
0
q2
1
1
q3
Run (Connect)
Clear

Figure 6-22: The string has traveled through all of the states.


LADDER Domain Builder
File Debug View
Define Domain User Study Judge Users
DOMAIN LIST
Loaded LDL File: Finitestate
Primitives
Text
Shape
Stroke
Ellipse
Line
Point
Shapes
sketch.shapes.finitestate.State
sketch.shapes.geom.arrow.Arrow
sketch.shapes.finitestate.AbstractState
sketch.shapes.finitestate.Transition
sketch.shapes.finitestate.InputString
sketch.shapes.finitestate.AcceptState
sketch.shapes.finitestate.StartState
#sketch.shapes.finitestate.Connection
Add Move Up Move Down
Un/Comment Debug Delete
SHAPE CONCEPT
NEAR MISS SHAPE
SHAPE DEFINITION
DRAWING SPACE
01
q1
0
q2
Diagram Message
Yes No Cancel
Run (Connect) Clear

01 String Rejected

Figure 6-23: The string is rejected because the last state is not an accept state.

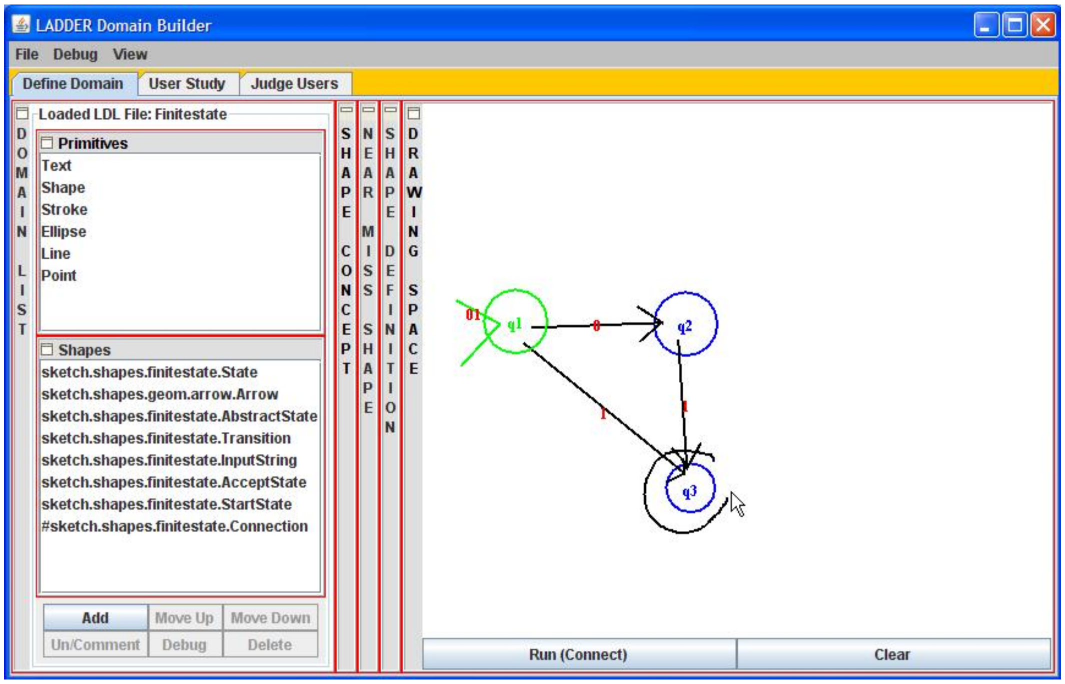

LADDER Domain Builder
File Debug View
Define Domain User Study Judge Users
DOMAIN LIST
Loaded LDL File: Finitestate
Primitives
Text
Shape
Stroke
Ellipse
Line
Point
Shapes
sketch.shapes.finitestate.State
sketch.shapes.geom.arrow.Arrow
sketch.shapes.finitestate.AbstractState
sketch.shapes.finitestate.Transition
sketch.shapes.finitestate.InputString
sketch.shapes.finitestate.AcceptState
sketch.shapes.finitestate.StartState
#sketch.shapes.finitestate.Connection
Add Move Up Move Down
Un/Comment Debug Delete
SHAPE CONCEPT
NEAR MISS SHAPE
SHAPE DEFINITION
DRAWING SPACE
01
q1
0
q2
1
1
q3
Run (Connect) Clear

Figure 6-24: Changing state q3 to be an accept state.

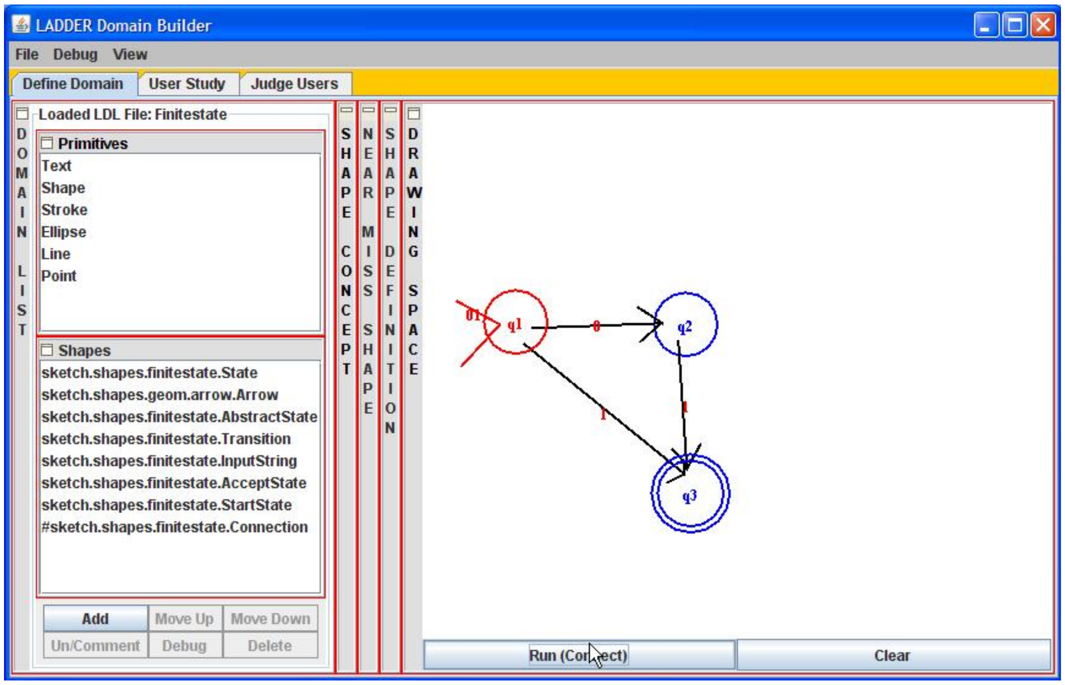

LADDER Domain Builder
File Debug View
Define Domain
User Study
Judge Users
Loaded LDL File: Finitestate
DOMAIN LIST
Primitives
Text
Shape
Stroke
Ellipse
Line
Point
Shapes
sketch.shapes.finitestate.State
sketch.shapes.geom.arrow.Arrow
sketch.shapes.finitestate.AbstractState
sketch.shapes.finitestate.Transition
sketch.shapes.finitestate.InputString
sketch.shapes.finitestate.AcceptState
sketch.shapes.finitestate.StartState
#sketch.shapes.finitestate.Connection
Add
Move Up
Move Down
Un/Comment
Debug
Delete
SHAPE CONCEPT
NEAR MISS SHAPE
SHAPE DEFINITION
DRAWING SPACE
01
q1
0
q2
1
1
q3
Run (Connect)
Clear

Figure 6-25: Testing the string on the modified FSM.

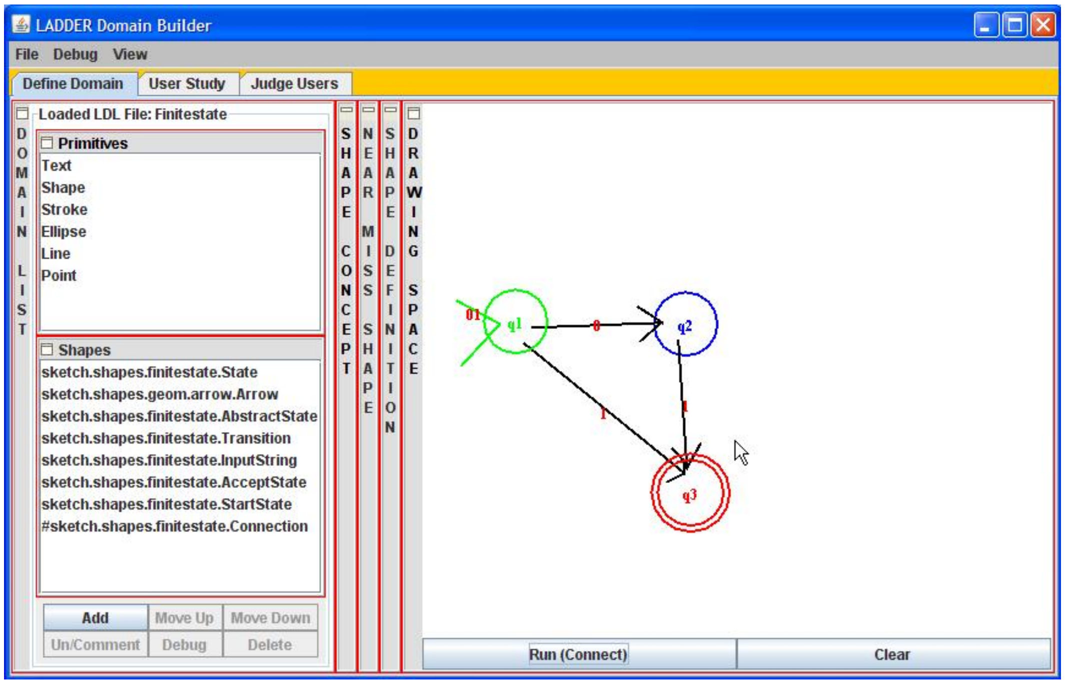
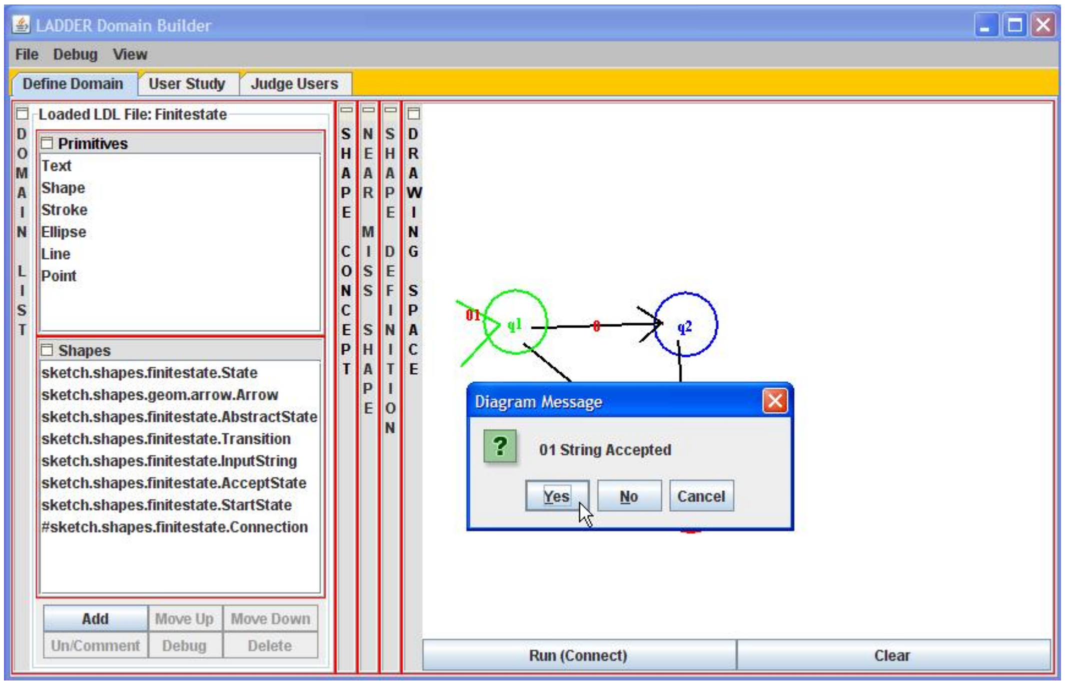

LADDER Domain Builder
File Debug View
Define Domain
User Study
Judge Users
Loaded LDL File: Finitestate
DOMAIN LIST
Primitives
Text
Shape
Stroke
Ellipse
Line
Point
Shapes
sketch.shapes.finitestate.State
sketch.shapes.geom.arrow.Arrow
sketch.shapes.finitestate.AbstractState
sketch.shapes.finitestate.Transition
sketch.shapes.finitestate.InputString
sketch.shapes.finitestate.AcceptState
sketch.shapes.finitestate.StartState
#sketch.shapes.finitestate.Connection
Add
Move Up
Move Down
Un/Comment
Debug
Delete
SHAPE CONCEPT
NEAR MISS SHAPE
SHAPE DEFINITION
DRAWING SPACE
01
q1
0
q2
1
1
q3
Run (Connect)
Clear

Figure 6-26: The last state is now an accept state.


LADDER Domain Builder
File Debug View
Define Domain
User Study
Judge Users
Loaded LDL File: Finitestate
DOMAIN LIST
Primitives
Text
Shape
Stroke
Ellipse
Line
Point
Shapes
sketch.shapes.finitestate.State
sketch.shapes.geom.arrow.Arrow
sketch.shapes.finitestate.AbstractState
sketch.shapes.finitestate.Transition
sketch.shapes.finitestate.InputString
sketch.shapes.finitestate.AcceptState
sketch.shapes.finitestate.StartState
#sketch.shapes.finitestate.Connection
Add
Move Up
Move Down
Un/Comment
Debug
Delete
SHAPE CONCEPT
NEAR MISS SHAPE
SHAPE DEFINITION
DRAWING SPACE
01
q1
0
q2
Diagram Message
Yes
No
Cancel
Run (Connect)
Clear

01 String Accepted

Figure 6-27: The input string is accepted.

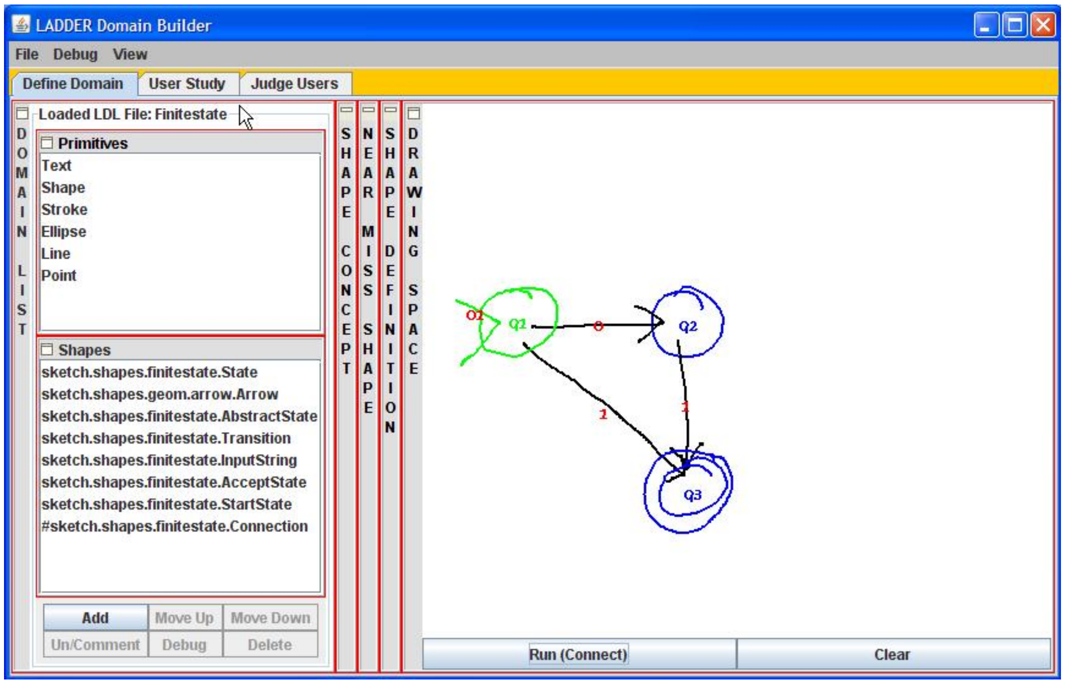
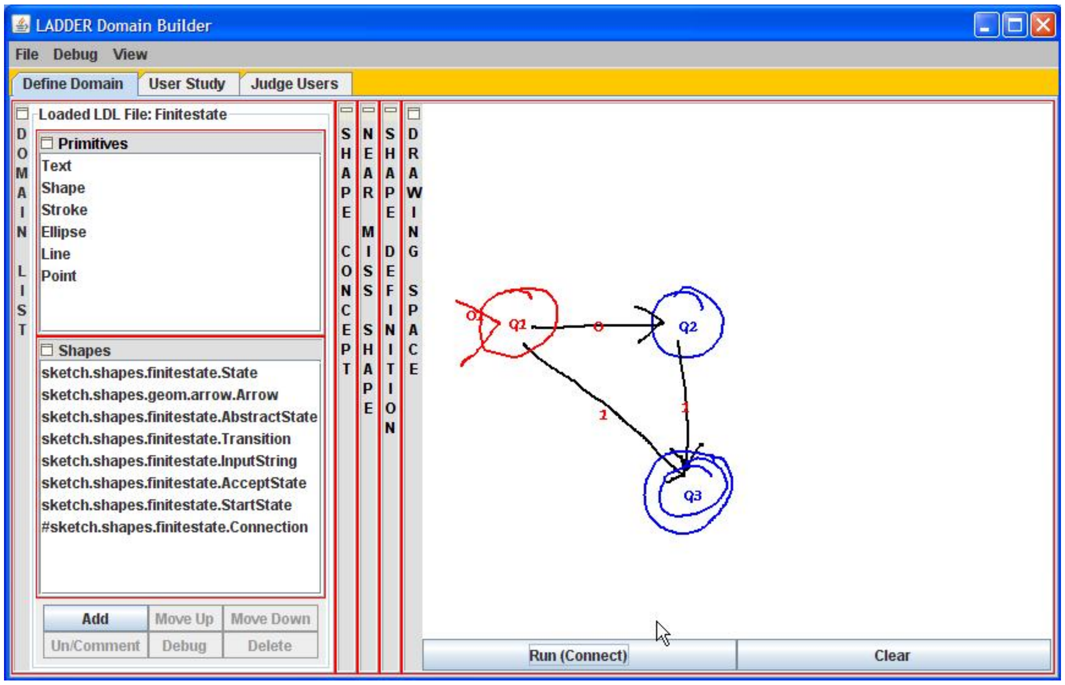

LADDER Domain Builder
File Debug View
Define Domain
User Study
Judge Users
Loaded LDL File: Finitestate
DOMAIN LIST
Primitives
Text
Shape
Stroke
Ellipse
Line
Point
Shapes
sketch.shapes.finitestate.State
sketch.shapes.geom.arrow.Arrow
sketch.shapes.finitestate.AbstractState
sketch.shapes.finitestate.Transition
sketch.shapes.finitestate.InputString
sketch.shapes.finitestate.AcceptState
sketch.shapes.finitestate.StartState
#sketch.shapes.finitestate.Connection
Add
Move Up
Move Down
Un/Comment
Debug
Delete
SHAPE CONCEPT
NEAR MISS SHAPE
SHAPE DEFINITION
DRAWING SPACE
01
Q1
0
Q2
1
1
Q3
Run (Connect)
Clear

Figure 6-28: The original strokes.

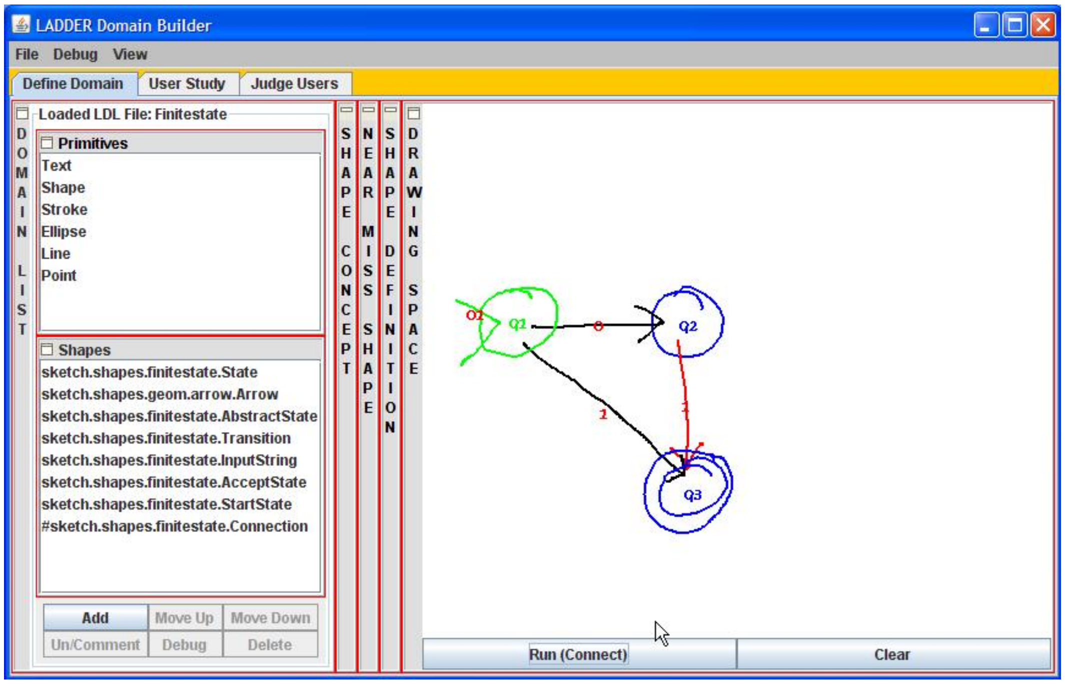


Loaded LDL File: Finitestate

Primitives
Text
Shape
Stroke
Ellipse
Line
Point

Shapes
sketch.shapes.finitestate.State
sketch.shapes.geom.arrow.Arrow
sketch.shapes.finitestate.AbstractState
sketch.shapes.finitestate.Transition
sketch.shapes.finitestate.InputString
sketch.shapes.finitestate.AcceptState
sketch.shapes.finitestate.StartState
#sketch.shapes.finitestate.Connection

Figure 6-29: Running the input string on the messy strokes.



Loaded LDL File: Finitestate

Primitives
Text
Shape
Stroke
Ellipse
Line
Point

Shapes
sketch.shapes.finitestate.State
sketch.shapes.geom.arrow.Arrow
sketch.shapes.finitestate.AbstractState
sketch.shapes.finitestate.Transition
sketch.shapes.finitestate.InputString
sketch.shapes.finitestate.AcceptState
sketch.shapes.finitestate.StartState
#sketch.shapes.finitestate.Connection

Figure 6-30: Continuing to run the input string on the messy strokes.

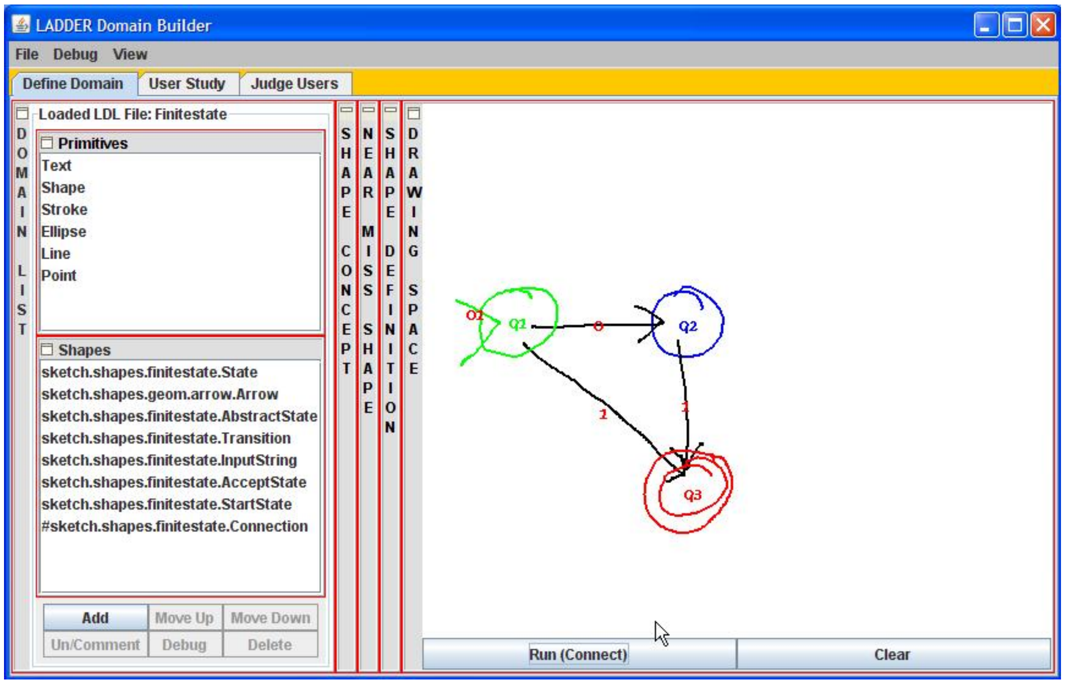

LADDER Domain Builder
File Debug View
Define Domain
User Study
Judge Users
DOMAIN LIST
Loaded LDL File: Finitestate
Primitives
Text
Shape
Stroke
Ellipse
Line
Point
Shapes
sketch.shapes.finitestate.State
sketch.shapes.geom.arrow.Arrow
sketch.shapes.finitestate.AbstractState
sketch.shapes.finitestate.Transition
sketch.shapes.finitestate.InputString
sketch.shapes.finitestate.AcceptState
sketch.shapes.finitestate.StartState
#sketch.shapes.finitestate.Connection
Add
Move Up
Move Down
Un/Comment
Debug
Delete
SHAPE CONCEPT
NEAR MISS SHAPE
SHAPE DEFINITION
DRAWING SPACE
Q1
Q2
Q3
0
1
1
Run (Connect)
Clear

Figure 6-31: The last state of the input string. (The string is accepted.)

6.6 Course of Action Diagrams

A sketch recognition system for course of action diagrams was built using the system. The code for this system is in the appendix of this document. Figures 6-32, 6-37, 6-35, 6-33, 6-36, 6-34 show a subset of the shapes recognizable from the descriptions. The images show the originally drawn images of course of action shapes that have been recognized along with system generated descriptions of each of the shapes. More images are shown in the Appendix.

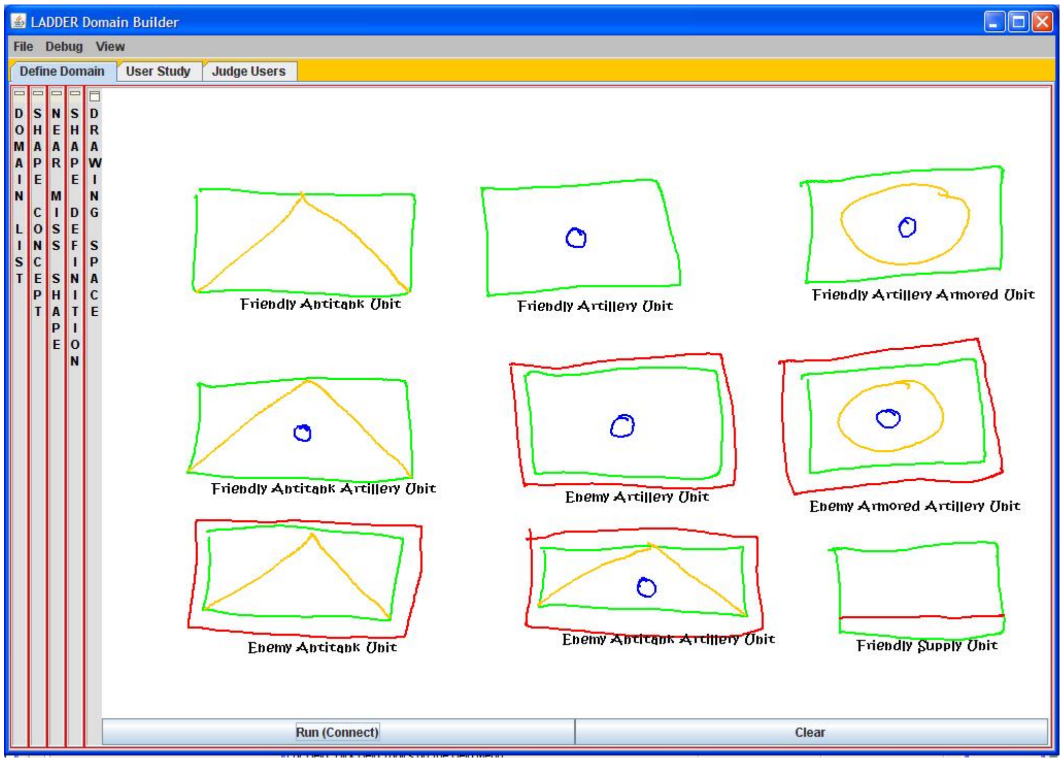


Figure 6-32: Examples of recognized hand-drawn Course of Action symbols.

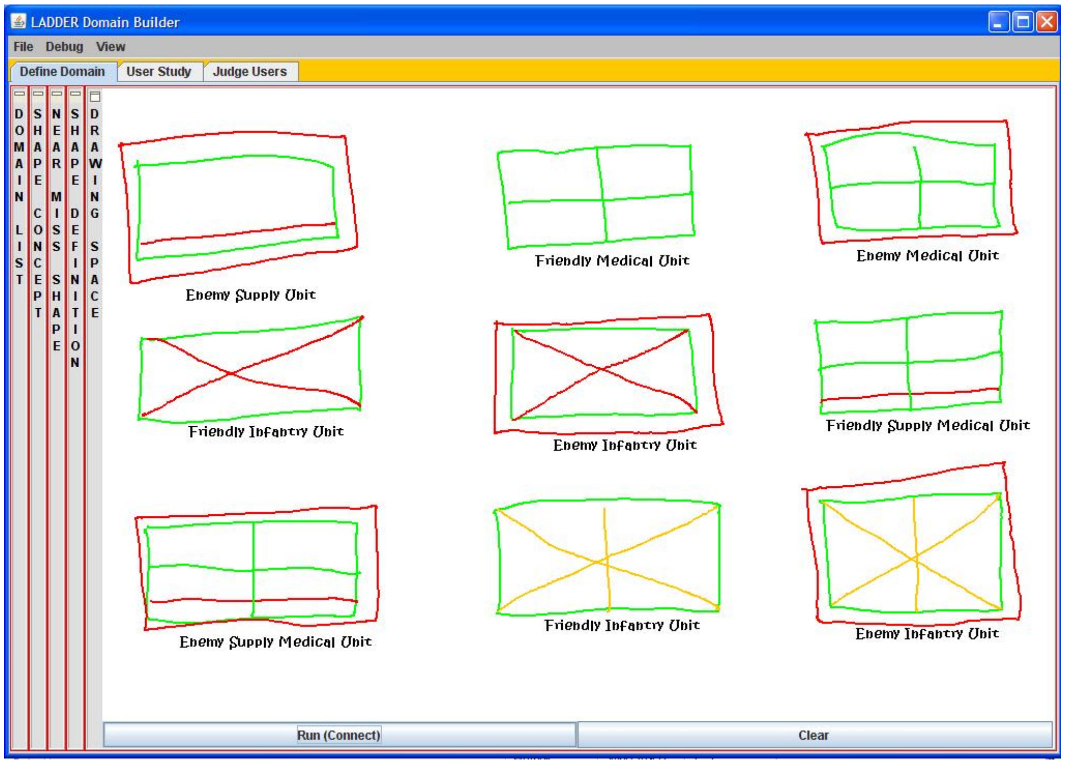


Figure 6-33: Examples of recognized hand-drawn Course of Action symbols.

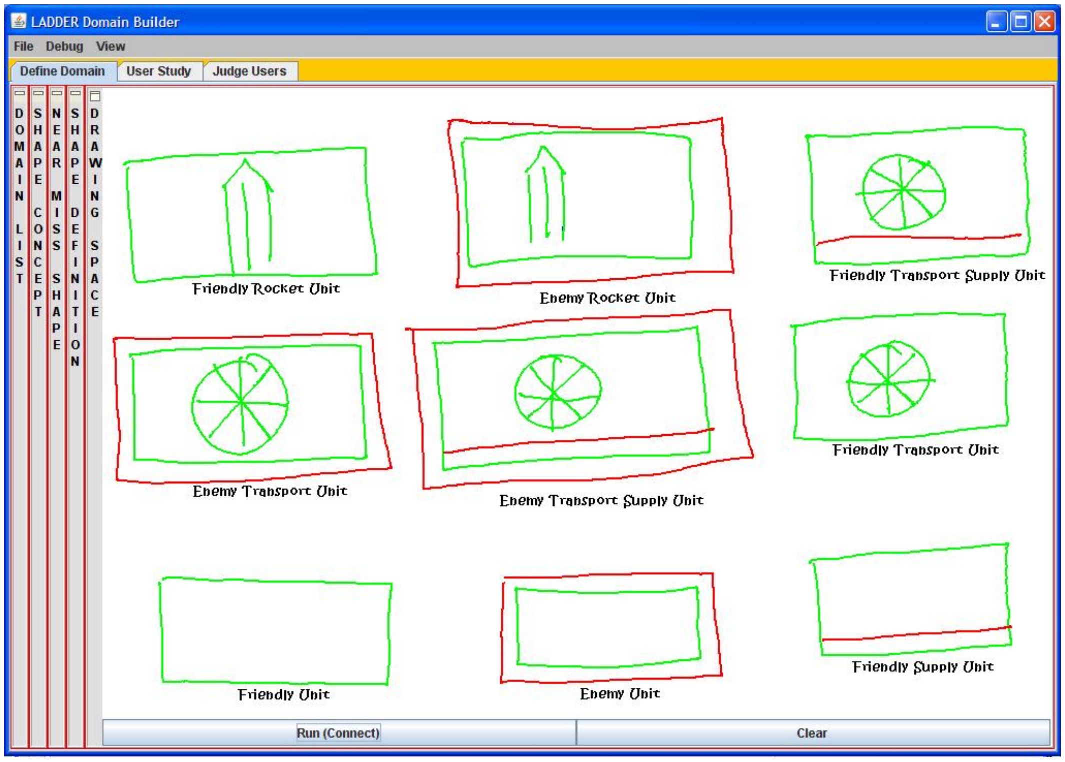


Figure 6-34: Examples of recognized hand-drawn Course of Action symbols.

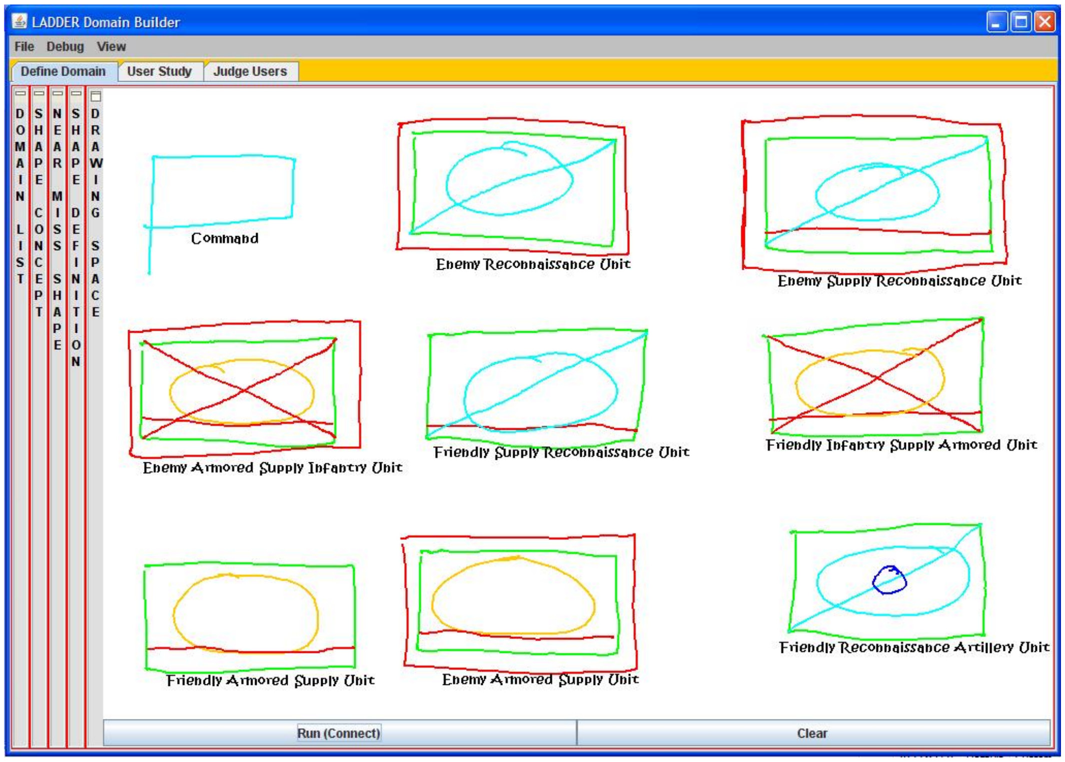

LADDER Domain Builder
File Debug View
Define Domain
User Study
Judge Users
DOMAIN LIST
SHAPE CONCEPT
NEAR MISS SHAPE
SHAPE DEFINITION
DRAWING SPACE
Command
Enemy Reconnaissance Unit
Enemy Supply Reconnaissance Unit
Enemy Armored Supply Infantry Unit
Friendly Supply Reconnaissance Unit
Friendly Infantry Supply Armored Unit
Friendly Armored Supply Unit
Enemy Armored Supply Unit
Friendly Reconnaissance Artillery Unit
Run (Connect)
Clear

Figure 6-35: Examples of recognized hand-drawn Course of Action symbols.

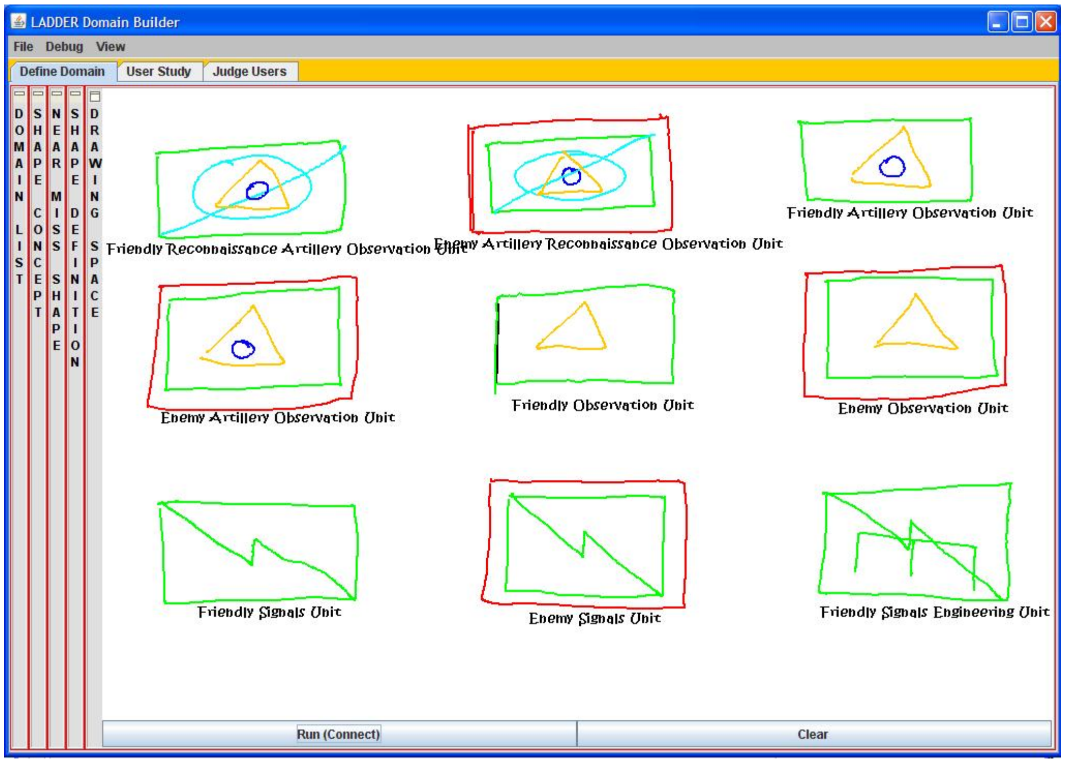

LADDER Domain Builder
File Debug View
Define Domain
User Study
Judge Users
DOMAIN LIST
SHAPE CONCEPT
NEAR MISS SHAPE
SHAPE DEFINITION
DRAWING SPACE
Friendly Reconnaissance Artillery Observation Unit
Enemy Artillery Reconnaissance Observation Unit
Friendly Artillery Observation Unit
Enemy Artillery Observation Unit
Friendly Observation Unit
Enemy Observation Unit
Friendly Signals Unit
Enemy Signals Unit
Friendly Signals Engineering Unit
Run (Connect)
Clear

Figure 6-36: Examples of recognized hand-drawn Course of Action symbols.

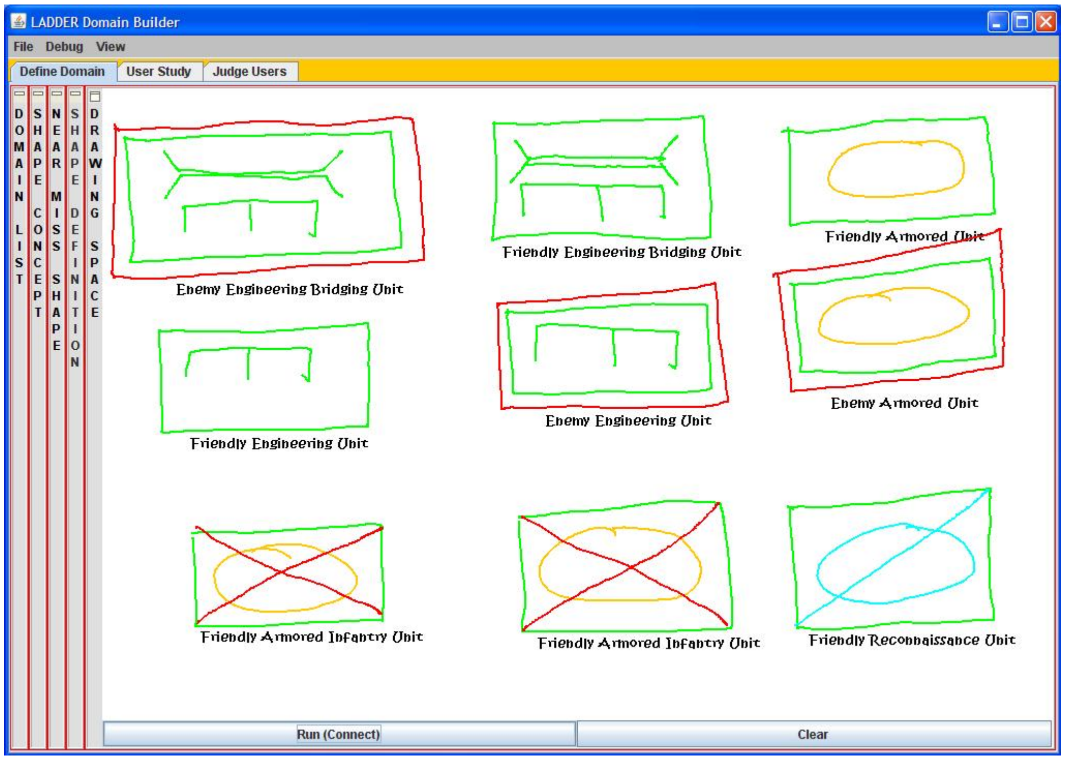


Figure 6-37: Examples of recognized hand-drawn Course of Action symbols.

6.7 User Comments

Thus far, about a dozen people have had significant experience with LADDER. We present two quotes having to do with their experiences and suggestions:

"On the most basic level, having seen, worked with, and manipulated LADDER, I am sold on the power and usefulness of the system. However, I have found some issues to still exist. There is no notion of a priority of shapes. One cannot indicate that a certain interpretation should be attempted before another. Since, once a low level shape has been determined to be a higher level shape, it cannot be re-evaluated. While this is a performance gain similar to scene graph pruning in 3d-rendering, this does mean that a false positive for a particular stroke can only be recovered by deleting and redrawing the sketch. Additionally, in some cases, if the false positive is too broad of a definition, then it will always be selected and the more specific, and more contextually correct option is not chosen."

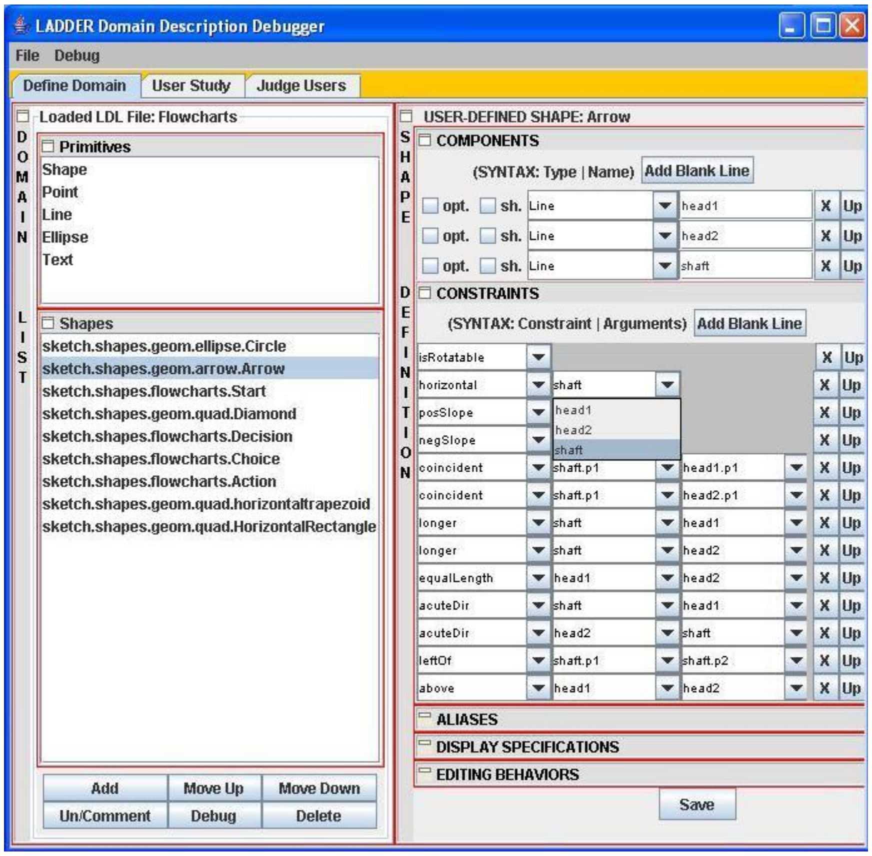
"The best part of the Ladder language was its hierarchical nature, as shapes can be built using other existing and user defined shapes. This obliterated the need to rewrite definitions for existing shapes which were part of the new shape and be able to re-use these shapes. I would also have like arcs to be handled better here because there are many significant domains that use arcs as shapes, and being a primitive in itself, it should have been handled by Ladder."

Chapter 7

SHADY: (SHApe DEbugger), an Editing Tool to Prevent Syntax Errors

As seen in the previous chapter, the language that is described in this thesis has proved to be quite powerful and capable of generating a number of different sketch systems. Manually typing a description can allow the developer to be specific about the shape to be described, and she can describe the shape as intended. However, creating a description by hand can be time-consuming. Human-generated descriptions typically contain syntactic and conceptual errors. We performed a user study in which 35 people were asked to describe shapes, using both their natural language and a more structured language, such as *LADDER*. We found both versions to contain a number of both syntactic and conceptual errors. Typical conceptual errors include omitted constraints (which allow unintended shapes to be recognized) and incorrect and conflicting constraints (preventing the intended shape from being recognized).

To deal with the problem of malformed syntax, we developed *SHADY*, a graphical user interface to input and debug shape descriptions [102]. The GUI constrains the input to allow only syntactically valid descriptions. Figure 7-1 shows a screenshot of



Loaded LDL File: Flowcharts

Primitives
- Shape
- Point
- Line
- Ellipse
- Text

Shapes
- sketch.shapes.geom.ellipse.Circle
- sketch.shapes.geom.arrow.Arrow
- sketch.shapes.flowcharts.Start
- sketch.shapes.geom.quad.Diamond
- sketch.shapes.flowcharts.Decision
- sketch.shapes.flowcharts.Choice
- sketch.shapes.flowcharts.Action
- sketch.shapes.geom.quad.horizontaltrapezoid
- sketch.shapes.geom.quad.HorizontalRectangle

USER-DEFINED SHAPE: Arrow

COMPONENTS

(SYNTAX: Type | Name)

opt.	sh.	Line	head1
opt.	sh.	Line	head2
opt.	sh.	Line	shaft

CONSTRAINTS

(SYNTAX: Constraint | Arguments)

isRotatable		
horizontal	shaft	
posSlope	head1	
negSlope	head2	
	shaft	
coincident	shaft.p1	head1.p1
coincident	shaft.p1	head2.p1
longer	shaft	head1
longer	shaft	head2
equalLength	head1	head2
acuteDir	shaft	head1
acuteDir	head2	shaft
leftOf	shaft.p1	shaft.p2
above	head1	head2

ALIASES

DISPLAY SPECIFICATIONS

EDITING BEHAVIORS

Figure 7-1: A screen shot of the GUI used to enter typed descriptions of a shape. The GUI automatically checks and controls for correct syntax.

the GUI used to enter in descriptions of a shape.

The GUI consists of three collapsible panels: Panel 1 is the domain list. Panel 2 is a syntax checker for entering shape definitions. Panel 3 is a drawing panel to draw the domain diagrams to test the included recognizers in real-time.

7.1 Panel 1: Domain List

The domain list in Panel 1 is an automatically computed listing of all of the primitive shapes and domain shapes used in the domain, and is a subset of all of the available primitive shapes. The primitive shape list is used to reduce the recognition possibilities. When a stroke is first recognized into a collection of primitive shapes, the system allows only interpretations using the primitive shapes used by the domain. By reducing the possibilities, recognition is both faster and more accurate. The domain shape list includes all of the non-primitive shapes that were either part of the LADDER library of predefined shapes or were defined specifically for this domain. Shapes can be added to, deleted from, reordered in, and commented out/in of this list.

7.2 Panel 2: Shape Definition Syntax Checker

Panel 2 consists of a shape definition syntax checker. A shape can be started anew or an existing definition can be loaded. The GUI allows the user to specify any existing shape that the current shape extends.

Each input line consists of several auto-complete drop down boxes that dynamically update themselves as the shape description is typed. Users can type whatever they want, but incorrect syntax (a value not listed in the drop-down box) is turned red. The system allows the user to type in invalid entries because the user may make other changes in the descriptions in the future that may make that statement valid. The goal is not to constrain the developer, but, instead, encourage her to type syntactically correct descriptions.

The components subsection allows one to enter in the component type and name for each component. The available types in the drop down box are all primitives and all shapes included in the domain list. A shape can be marked as shared or optional. For each constraint in the constraint subsection, there is a drop down box for all available constraints in *LADDER*. Once a constraint is chosen, the number of argument boxes is automatically dynamically updated to provide the same number of arguments as are appropriate for the shape. Each argument input box is built from a drop down box whose contents are also dynamically created. The system first checks what is the valid shape type permitted in that argument box, then it dynamically checks which shapes are accessible from the list of components, and fills in the constraint drop down box with all shapes of the appropriate type. Similar checking is done for the aliases, display, and editing GUI subsections.

7.3 Panel 3: Drawing Panel

The GUI also includes a drawing panel, which is essentially the same as the domain recognition system output by this system. As new shapes are added (or updated) to the domain list, it dynamically adds (or updates) them to the list of recognizable shapes so that they are appropriately recognized when drawn on the drawing panel. To recognize the shape, the system uses the same recognition techniques employed by the *GUILD* recognition engine, as described in Chapter 5 [100].

Chapter 8

Computer-generated Descriptions from a Single Example

We correct syntax errors using a constrained GUI. Yet, we feel that it is more natural to draw a shape than it is to type it textually, so we built a system that automatically generates shape descriptions from a single example using techniques similar to Veselova [210]. It turns out to be quite difficult for non-computer scientists to list all of the constraints necessary to in a complete shape description: one needs to think very logically to complete this task. Computer generated descriptions have the added benefit that they are free of syntactic errors.

Generating a shape description consists of four steps.

1. The user draws a sample shape.

2. The system generates a list of all true constraints.

3. The system confirms, with the help of the user, that the initial list is correct.

4. The system shortens the list, using perceptual rules to create a best-guess description.

8.1 Generating a List of All True Constraints

8.1.1 Component Labeling

The system begins by assigning a label to each component in the initial hand-drawn shape (e.g., LINE1, LINE2). It then creates a list of all of the components that are accessible within one layer of indirection (e.g., LINE1, LINE1.P1, LINE1.CENTER, LINE1.P2, etc.).

8.1.2 Constraint Listing

LADDER constraint inter-relations (with some exceptions; see below) between the components listed above are considered, and the appropriate constraint is generated. For every shape, there is a fixed number of constraints to describe that shape, based on the number and type of primitives that make up the shape. For example, if there are four lines, then 12 "size" constraints will be listed, with each line pairing assigned with either EQUALSIZE[1], LARGER, or SMALLER. Each *LADDER* constraint is grouped with other mutually exclusive constraints that measure a certain property (e.g., size). Another example of a mutually exclusive group within which only one constraint can be true is orientation: HORIZONTAL, POSSLOPE, NEGSLOPE, VERTICAL.

A three-lined shape will necessarily have (among other constraint types) exactly three orientation constraints, exactly three relative size constraints, and exactly 27 coincident constraints (comparing the p1, p2, and center points of each line).

This generated list describes the specific instance of the drawn shape. A recognizer based on this complete description will recognize the initial drawn shape, but it will not recognize any shapes with differing true constraints, even if it is similar to the initial drawn shape in many ways.

[1]EQUALLENGTH is syntactic sugar for EQUALSIZE, and used to compare size in lines

8.1.3 Limited Set of Constraints

LADDER includes a large number of constraints, and, because constraints are interrelated, the same shape can be described in a number of ways. We would like to limit the number of constraints that are generated. Because indirection greatly increases the number of generated constraints, we limit its use to COINCIDENT and CONTAINS when the subshape is a line. We also do not keep constraints using the subshape's STROKE or BOUNDINGBOX, since there will exist a similar constraint using the shape itself. For example, we include INTERSECTS LINE1 LINE2, but not INTERSECTS LINE1.STROKE LINE2.STROKE nor INTERSECTS LINE1.BOUNDINGBOX LINE2.BOUNDINGBOX, as they are redundant.

Other omitted constraints, and the reason for omitting them are listed below:

equal, greaterThan, greaterThanEqual These constraints compare properties of many different types (e.g., HEIGHT, WIDTH, AREA, LENGTH, ANGLE). We do not know how to create near-misses for these constraints since error thresholds would vary greatly for different property types. (We generate constraints comparing properties using property-specific constraints such as EQUALANGLE, EQUALSIZE, or EQUALLENGTH.)

negation constraints Negation constraints are omitted because they would be redundant. Listing a positive constraint automatically implies several negative constraints. In particular, a positive constraint implies a negative constraint for all of the other constraints in a mutually-exclusive group. For instance, if we have included the constraint HORIZONTAL LINE1 is true for the hand-drawn shape, that automatically implies that NOT POSSLOPE LINE1, NOT NEGSLOPE LINE1, and NOT VERTICAL LINE1. (Note that for any constraints where the mutually-exclusive constraints in *LADDER* do not cover the entire space of possible drawn shapes, we add an additional constraint to cover the rest of the space, e.g., CONTAINS, and NOTCONTAINS.)

disjunctive (or) constraints Disjunctive constraints are omitted because we are including all possible true constraints in the positive example, and, because it is impossible for a disjunctive constraint to be present in a single example. The presence of a disjunctive constraint implies variation, and we are not yet including variation in our description (since we are attempting to describe a single specific instance of a shape). (The system can generate disjunctive constraints later in the process; see below.)

composite constraints Composite constraints, which combine two other constraints already in the language, are redundant syntactic sugar, and, thus, are omitted.

The initial list of constraints is kept short by ensuring that we do not include tautologically true constraints. For example, the center and the two endpoints of a line should not be listed as collinear.

Below is an example of some of the rules used to prevent tautologically true constraints:

- A shape should not be listed having a constraint relationship with itself.
- The center and the two endpoints of a line should not be listed as collinear.
- Even though the CONTAINS constraint examines the BOUNDINGBox of a shape, neither a line nor a point can contain an item (even a point).
- A shape should not be listed as larger than a point (since it is meaningless to state that a shape with a positive area is larger than a point, all of which have an area of size equal to 0).
- A line should not be listed as larger than another line. (The constraint LONGER is to be used, instead, for lines, as LARGER compares the bounding boxes of two shapes and LONGER compares the lengths of two lines.)
- A shape and a subpart of this shape should not be listed as being on the same side of a line.

8.2 Confirming that the Initial List Is Correct

Because the initial shape is hand-drawn, it will have some signal errors (as defined in Section 5.1.1). Signal errors are managed by the recognition system, and, while a shape description should include conceptual variations, it should not include signal variations. Thus, the system needs to generate a description that includes the conceptual variations (conjectured with techniques below), but without signal variation. To interpret what the initial drawn shape would look like without signal variation, the system needs to recognize what the sketcher intended to draw, rather than what he drew (e.g., recognize that the sketcher intended to draw two lines to be of equal length, even though, due to the messiness of his sketch, the two lines may not be exactly of equal length).

The recognition system chooses the initial list of true constraints using the built-in thresholds. These error tolerances are set to handle most data and to try to determine the user's intention when drawing a shape. Examples: Lines drawn must be within 15 degrees of 0 and 90 in order to be considered HORIZONTAL or VERTICAL, respectively. Line lengths have to be within a factor of 1.2, or have a difference of less than 20 pixels in order to be considered of equal length. Coincident (and bisecting) points must be within 20 pixels of each other, or the distance must be smaller than 1/8 of the length of both attached lines (or the diameter of a circle when comparing its center). Note that these thresholds both include a constant threshold, related to pixels and screen size, and a relative threshold, related to line lengths, as people tend to be less careful about perfectly joining their lines when shapes are drawn larger.

But human error is not deterministic; these error tolerances are imperfect. Thus, if the signal error is large (i.e., the drawing is particularly messy), it is conceivable that our recognition system incorrectly recognized a constraint, and, as a result, our initial list of all true constraints generated by the system will have an error in it.[2]

[2]The system may have incorrectly classified a line drawn at 14 degrees as horizontal, when the sketcher meant it to have a positive slope, or the system may have incorrectly classified a line drawn at 16 degrees as positively sloped, when the sketcher meant it to be horizontal.

To deal with this possibility, we use MATLAB to generate an example constraint that obeys all of the generated constraints. (For example, if the recognition system detects what it considers to be a horizontal line, it recreates the shape with that line perfectly horizontal.) By generating the shape with the chosen constraints obeyed, the interpretations made by the system become perceptually obvious.

We present the generated shape to the developer, and ask her to confirm that the shape is a "cleaned-up" version of the shape she drew. If she says "yes," the system continues to the next section. If she says "no," the system has made a classification error, and the developer is asked to redraw the shape more carefully to reduce the chance of misclassification due to signal error.

8.3 Choosing the Appropriate Generalization

The generated list of all true constraints describes the specific instance of the initial hand-drawn shape without allowable conceptual variations. We know that the correct shape description will be some generalized version of this description. It will accept some shapes (including this one) and reject others. The task of the computer is to determine, as best as possible, what the appropriate generalization is, using perceptual rules.

The computer generalizes the constraint list by selecting perceptually-important constraints to remain, and removing others. We first prune the list by removing redundant constraints.

8.3.1 Removing Redundant Constraints

Several of the constraints are redundant, providing no additional information when taking into account the other constraints. For instance, if LINE1 is above LINE2, then implied by this statement is that each of the subcomponents of LINE1 is above each of the subparts of LINE2. The system removes these more-general (in that they only require one part of the line to be to the left of the other), but redundant, constraints. We do this because, when drawing an example shape, users will often draw it in the a more general way. For example, if a sketcher is drawing a rectangle, he usually will not draw it with all four sides of equal length, even though a square is still a valid rectangle.

Examples of rules used to prune at this stage include:

- If a line is listed as VERTICAL, remove the constraints stating that any of the line's subparts are above one another or are horizontally aligned (share the same x value). HORIZONTAL is treated in the same way.
- If one shape is ABOVE another, remove constraints stating that the subshapes of those shapes that are above each other. (For example, if LINE1 is above LINE2, remove the constraint stating that LINE1.P1 is above LINE2.) The following constraints are treated in the same manner: LEFTOF, SAMEX (vertically aligned), SAMEY (horizontally aligned), SAMESIDE, or OPPOSITESIDE (of a line).
- Remove any collinear constraints in which all of the points are on a single line when taking into account coincident constraints. (For example, if LINE1.P1 is coincident with LINE2.P2, remove the constraint COINCIDENT LINE2.P2 LINE1.CENTER LINE1.P2.)
- Remove NEAR constraints, if the points are also COINCIDENT.
- Remove ACUTE and OBTUSE constraints, if OBTUSEMEETS or ACUTEMEETS (i.e., the lines are slanted and the endpoints are coincident) constraints are true.
- Remove constraints that are true because of transitivity. (For example, if LINE1 and LINE2 are equal length, and LINE2 and LINE3 are of equal length, then remove the constraint stating that LINE1 and LINE3 are of equal length.) Other relational constraints are treated similarly, including EQUALSIZE, LARGER, PARALLEL, PERPENDICULAR, LEFT OF, ABOVE, SAMESIDE, and COINCIDENT.

- Remove similar CONNECTED and MEETS constraints, if points are also COINCIDENT.

- If three lines are vertically or horizontally aligned, remove any COLLINEAR constraints among them.

8.3.2 Selecting Perceptually-Important Constraints

From the remaining constraints, the system attempts to choose which constraints are required for the shape, and which happen to be true just because the sketcher had to draw a specific instance on paper.

Certain constraints seem less accidental and are more perceptually important. Recall the example given earlier in this chapter. When drawing an example of a rectangle, a sketcher will rarely draw all four sides to be of equal length. The reason for that is that humans are particularly attuned to certain visual properties that are seen as more perceptually important. (See Section 4.2 for a more complete description of perceptual constraints.) When shapes that include these perceptually-important properties are drawn, they are considered non-accidental. To give an example: If a sketcher drew two lines of equal length, it is improbable that he meant to imply that the lengths between the two lines can have any ratio, whereas, if he drew two lines of differing length, it is probable that the two lines also can be of equal length.

The system now selects a subsets of constraints as our best-guess. By selecting only a subset, the system introduces allowable conceptual variations to the shape concept. The system limits the initial number of constraints in our best-guess description to n^2, where n is the number of primitive components in the drawn example. (For example, an arrow has three lines, and, thus, the system would choose nine initial constraints.) This has been found to to be a good heuristic, as it allows at least one constraint stating how each shape relates to every another shape (equal to $n*(n-1)/2$, which is less than n^2), but does not allow a constraint for every type of relation that a shape can have with every other shape (which would require greater than $7 * n(n-1)/2$ constraints, which is greater than the chosen n^2 limit). Future sections of this thesis describe how to improve this best-guess with near-miss examples. There is no limit to the number of constraints that may exist in the final description after that process.

For our best-guess, the system selects those constraints which are the most perceptually important, and, thus, seem less accidental.

The system loosely ranks the constraints in order of their perceptual importance. In particular, the following constraints are identified as most perceptually important: COINCIDENT, HORIZONTAL, VERTICAL, BISECTS, PARALLEL, EQUALSIZE, SAMEX (vertically aligned), and SAMEY (horizontally aligned).

The system selects all of the perceptually-important constraints from the list of all possible true constraints. If there are more then n^2 constraints after adding the above constraints, the system removes those constraints closer to the end of the above list, such as SAMEX and SAMEY, since they are perceptually weaker constraints than the others. (HORIZONTAL is stronger than SAMEY because, in the former, the horizontal line is concrete, whereas in the latter it is abstract.)

If there are fewer than n^2 constraints, the system adds constraints in an order based loosely on perceptual importance: ACUTEMEETS, OBTUSEMEETS, LARGER, PERPENDICULAR, CONNECTED, MEETS, INTERSECTS, COLLINEAR, NEAR, POSSLOPE, NEGSLOPE, LEFTOF, ABOVE, ACUTE, OBTUSE, SAMESIDE, and FAR.

This final list of constraints in the initial best-guess description will contain n^2 of the most perceptually-important constraints.

Chapter 9

Debugging Descriptions with User-generated Examples

While, we can prevent syntax errors by manually typing in shape descriptions, using the *SHADY* GUI tool, or by having them generated from a single drawn example, shape descriptions can still include conceptual errors.

Computer-generated descriptions may be imperfect because it may be difficult for the computer to determine the acceptable variations intended by the developer from a single example. The perceptual rules help the computer to produce a reasonable description, but there are several variations that are domain-specific (such as whether or not a shape can be rotated) that cannot be determined from perceptual rules alone. These generated descriptions may be over- or under-constrained. For instance, if the developer draws a square, she may intend something as specific as a square or as general as a rectangle or a quadrilateral.

Figure 9-1 shows the difficulty of automatically generating a perfect description; the components line1 and line2 look the same in both the square and the arrow. The constraint PERPENDICULAR LINE1 LINE2 is true for both shapes, and any computer algorithm that would include the constraint for one shape would include it for the

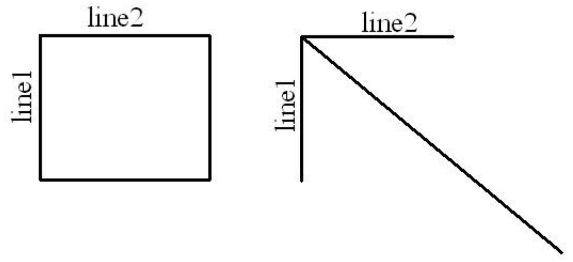


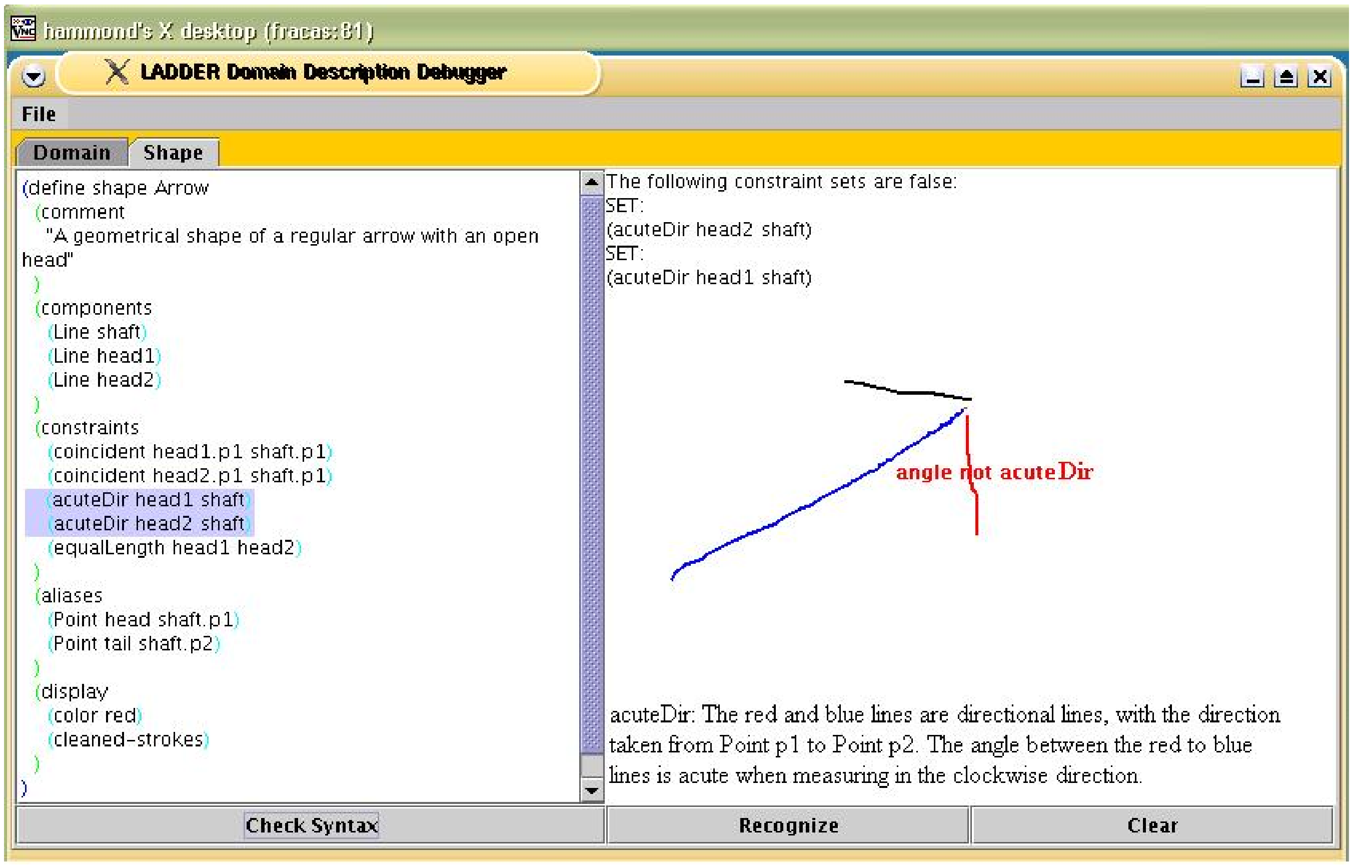
Figure 9-1: Lines 1 and 2 are identical in the square and in the arrow. However, the constraint PERPENDICULAR LINE1 LINE2 should be included in a square description, but not in an arrow description.

other. However, if the constraint is missing from the square definition, the square definition will be incorrect, as it is under-constrained, but, if the constraint is included in the arrow definition, it will be over-constrained and incorrect. If a shape definition is over-constrained, a drawn shape will not be recognized (giving false negatives), while, if it is under constrained, drawn shapes other than the one intended will be recognized (giving false positives).

9.1 System Interaction

To remove conceptual errors, the system needs to know the allowable variations in a shape. The obvious way to solve this problem is to have users supply several positive and negative examples, and have the system learn from these user-supplied examples. So, this research includes the development of a system that learned allowable variations from user-provided near-miss examples [214]. The user first provides the system with a shape description (either manually typed in or computer-generated from a single example), then draws several example shapes to test whether the description would properly recognize the acceptable variations.

The system would attempt to recognize the drawn shapes using the provided shape description. If a shape description is over-constrained, it will produce a false negative, i.e., fail to recognize a shape that it should have recognized. The system provides a panel on which the user draws a positive example of the shape described. If the shape is not recognized based on the description given, the system highlights the failed constraint or constraints. The developer can then decide to remove or adjust the specified constraint(s). Figure 9-2 shows a description of an arrow being debugged.


hammond's X desktop (fracas:81)
LADDER Domain Description Debugger
File
Domain
Shape
angle not acuteDir
Check Syntax
Recognize
Clear

```
(define shape Arrow
  (comment
   "A geometrical shape of a regular arrow with an open
head"
  )
  (components
    (Line shaft)
    (Line head1)
    (Line head2)
  )
  (constraints
    (coincident head1.p1 shaft.p1)
    (coincident head2.p1 shaft.p1)
    (acuteDir head1 shaft)
    (acuteDir head2 shaft)
    (equalLength head1 head2)
  )
  (aliases
    (Point head shaft.p1)
    (Point tail shaft.p2)
  )
  (display
    (color red)
    (cleaned-strokes)
  )
)
```

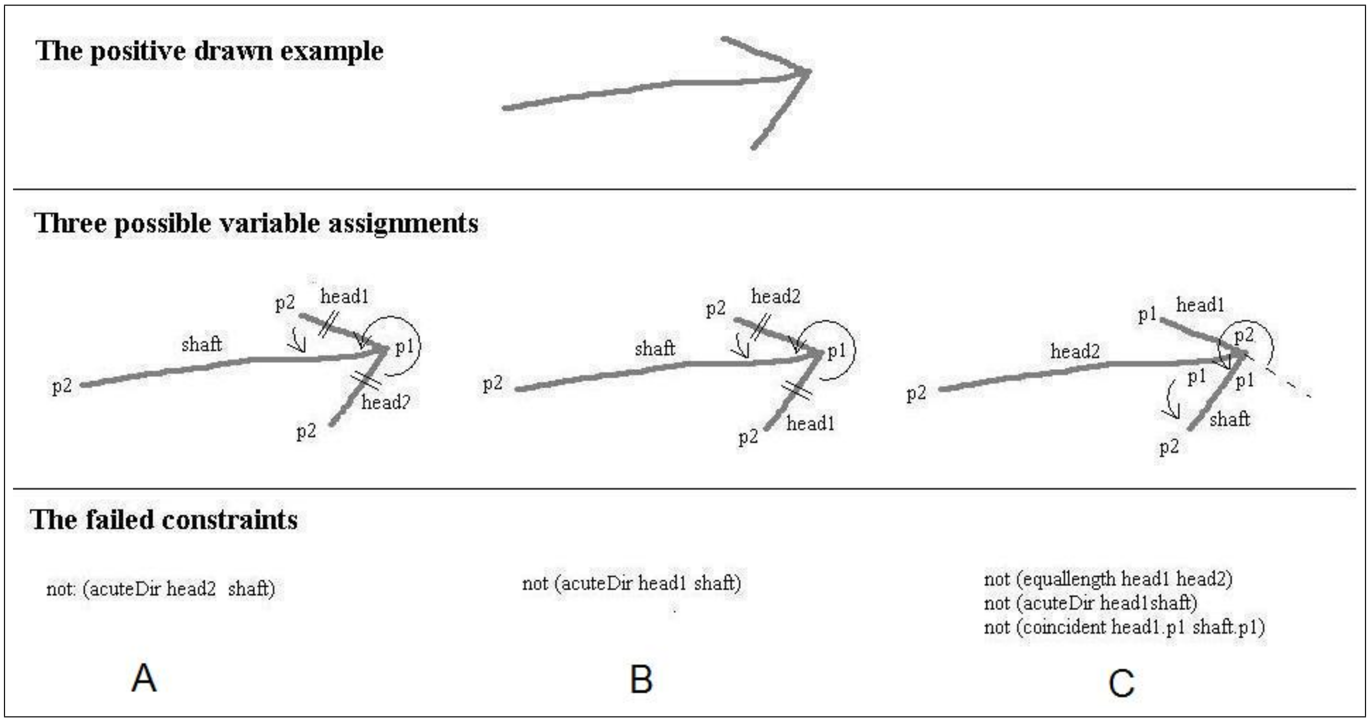
```
The following constraint sets are false:
SET:
(acuteDir head2 shaft)
SET:
(acuteDir head1 shaft)
```

acuteDir: The red and blue lines are directional lines, with the direction taken from Point p1 to Point p2. The angle between the red to blue lines is acute when measuring in the clockwise direction.

Figure 9-2: An incorrect arrow description being debugged from user-provided examples The arguments in the second ACUTE constraint are incorrectly flipped. The constraint should read (ACUTE SHAFT HEAD2), indicating that the counter-clockwise angle formed from the directional lines is acute.

```
define shape Arrow
  components
    Line shaft
    Line head1
    Line head2
  constraints
    coincident head1.p1 shaft.p1
    coincident head2.p1 shaft.p1
    acuteMeet head1 shaft
    acuteMeet shaft head2
    equalLength head1 head2
```

Figure 9-3: A correct arrow description.

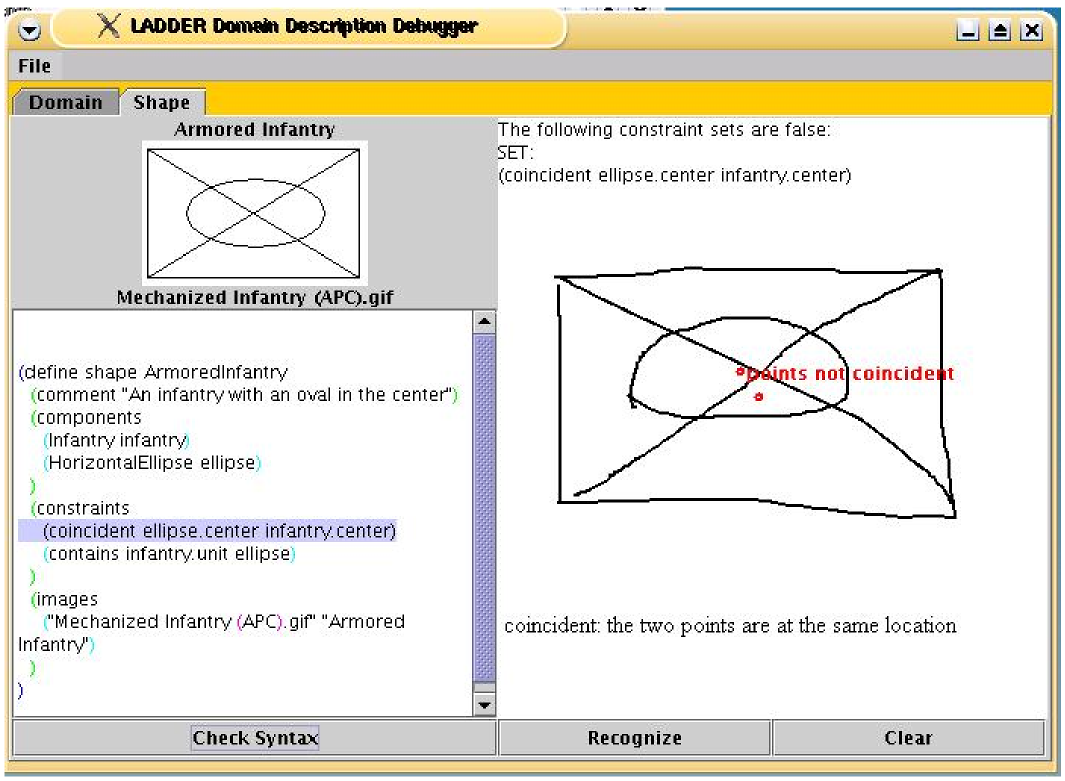


```
not: (acuteDir head2  shaft)
```

```
not (acuteDir head1 shaft)
```

```
not (equallength head1 head2)
not (acuteDir head1shaft)
not (coincident head1.p1 shaft.p1)
```

Figure 9-4: Three different variable assignments for an arrow, with the over-constrained descriptions from Figure 9-2, and their failed constraints.



```
(define shape ArmoredInfantry
  (comment "An infantry with an oval in the center")
  (components
    (Infantry infantry)
    (HorizontalEllipse ellipse)
  )
  (constraints
    (coincident ellipse.center infantry.center)
    (contains infantry.unit ellipse)
  )
  (images
    ("Mechanized Infantry (APC).gif" "Armored
Infantry")
  )
)
```

```
The following constraint sets are false:
SET:
(coincident ellipse.center infantry.center)
```

coincident: the two points are at the same location

Figure 9-5: An armored infantry shape description from the Course of Action domain is being debugged with the debugger.

9.2 Determining Failed Constraints

For any given shape and its description, there are many ways that the variable names of the components can be assigned. For example, the arrow described in Figure 9-3 has 48 possible variable assignments.[1]

Each different variable assignment causes different constraints to be false. The system generates all possible variable assignments and evaluates the user-provided constraints for each of them. Figure 9-4 presents three of the 48 possible assignments. Figure 9-4A and B give only one false constraint, whereas Figure 9-4C gives several false constraints.

9.3 Selecting Variable Assignments

If the system were to display all of the possible variable assignments and their failed constraints, the user would be overwhelmed. Instead, *SHADY* tries to choose the assignment or small collection of assignments that most closely matches what the user intended.

In Figure 9-2, a user actually intended to describe an arrow as in Figure 9-3, but he mistyped one of the constraints. The arguments in the second ACUTEMEET constraint are incorrect. The constraint should read ACUTEMEET SHAFT HEAD2, indicating that the angle formed is acute, if one 1) shifts the lines to collocate their P1 endpoints, and then 2) travels in a counter clockwise direction from HEAD2.P2 to SHAFT.P2.

The system makes the assumption that the description given by the developer is mostly correct and selects the variable assignments with the fewest failed constraints.

[1]The three variables, shaft, head1, head2, can be assigned to the three drawn lines (using combinatorics) in $C(3,3) = 3{*}2{*}1 = 6$ different ways. Each of the lines can have its two endpoints assigned in two ways (2^3), giving the total possible number of assignments to be $C(3,3) * 2^3 = 48$. (Notice that this number grows quickly as the four lines of a rectangle can be assigned in $C(4,4) * 2^4 = 384$ possible ways.)

If there are several variable assignments containing the minimum number of failed constraints, the system chooses all of them. In the case of Figure 9-4, the system chooses the variable assignments represented by Figure 9-4A and B.

9.4 Displaying the Failed Constraints

At this point, each of the selected variable assignments has the same number of failed constraints, and the system cannot further distinguish among them. Often, because of symmetry in the shape, different variable assignments can give the same failed constraint(s). When this occurs, the system collapses the two assignments into one, selecting only one of the variable assignments.

The system lists the collection of failed constraints for each chosen variable assignment and displays the failed constraints visually on the shape. In Figure 9-4A and 9-4B, both failed constraints constrain the same angle between the same two lines.[2] The system presents the failed constraint by changing the color of the components referenced in the failed constraint. The system also explains any failed constraint at the bottom of the screen, in case the developer has misused it. Figure 9-2 shows a screen shot of the system telling the developer which constraints have failed. Figure 9-5 provides another example of a more complicated hierarchically-defined shape being debugged.

9.5 System Faults

As noted previously, this system does help users to debug descriptions by identifying description errors from user-generated examples, and the system was useful for generating correct descriptions.

[2]The system has a unique identifier on each of the lines so that it can keep track of which line is which, even if it has a different label across different constraint lists.

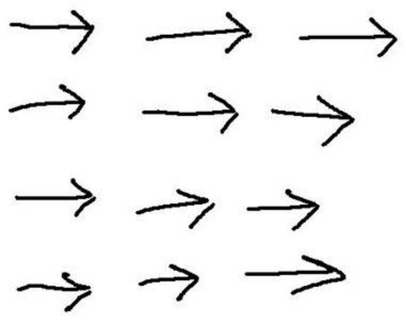


Figure 9-6: When asked to draw many examples of a particular shape in a user study, users tended to draw the shape in the same fashion repeatedly.

Unfortunately, users proved to be poor at generating sufficiently informative examples. In order for developers to completely test their description, they must generate a shape that tests each necessary constraint. However, there is no guarantee that the user will ever draw the shape in a way that exposes the bug in the description. Humans, especially non-computer scientists, are notoriously poor at generating informative near-miss examples, just as they are unreliable at generating informative test cases for a program.

As an experiment, nine users were asked to generate several variations of an arrow; one user produced the examples shown in Figure 9-6. When asked why he did not vary the arrow, he pointed out several minor variations in the arrows that he drew. When asked why he did not draw any rotated arrows, he paused for a minute, then said that he did not think of doing that, but that he would be happy to draw some now. When asked why he did not include other variations, he had a similar response. While other users provided more variations than in this example, none of them were complete in providing all of the allowable variations. Users are poor at drawing shape variations for the same reason that they are poor at textually typing shape variations; they forget constraints, making conceptual errors appear in their descriptions. It is also unlikely that the developer will draw such examples exposing forgotten constraints, as we are asking her to illustrate the faults in the description, and if she were aware of such faults, she would simply edit the description.

Chapter 10

Active Learning with System-generated Near-miss Examples to Refine a Concept

Winston developed a method for learning structural descriptions from examples [214]. He argued that the ideal training sequence is one in which each example is a near-miss. His work supposes that a human teacher supplies the system with appropriate near-misses. However, as mentioned earlier, this research has found users unable to produce a sufficient range of near-miss shapes that would make evident missing or superfluous constraints. As suggested previously, this is a generic phenomenon: whether debugging code or geometric descriptions, good test cases are difficult to generate.

To overcome this problem, the system itself provides the examples that help to refine its model. The work in this document applies this framework to the field of sketch recognition, and shapes are recognized based on the learned structural description.

As the sketch recognition interface is produced directly from the shape descriptions, the interface will only be as accurate as the descriptions. Descriptions with too

```
define shape Square
  components
    Line top
    Line left
    Line bottom
    Line right
  constraints
    horizontal top
    horizontal bottom
    vertical left
    vertical right
    equalLength left right
    equalLength top bottom
```

Figure 10-1: An under-constrained definition for a square. It does not specify that all four sides need to be the same (which could be rectified by including the constraint EQUALLENGTH TOP LEFT).

few relations (constraints) will recognize non-examples of the symbol (false positives), while descriptions with too many relations will produce false negatives, not permitting the degree of variation in a symbol's appearance that is routinely accepted by people using that graphical language.

10.1 Types of Conceptual Errors

There are two types of conceptual errors: an omitted constraint yielding an under-constrained description, and an erroneous constraint producing an over-constrained description. A simple example of an under-constrained description is given in Figure 10-1, where the definition for a square fails to require all four sides to be the same length (i.e., is missing EQUALLENGTH TOP LEFT). An example of an over-constrained shape description is given in Figure 10-2, where the definition of a rectangle contains the erroneous constraint EQUALLENGTH TOP LEFT. Substitution errors (e.g., VERTICAL TOP instead of HORIZONTAL TOP) are not considered to be an additional error type; rather, they are the result of the combination of an over-constrained definition (the vertical constraint should be removed) and an under-constrained definition (the

```
define shape Rectangle
  components
    Line top
    Line left
    Line bottom
    Line right
  constraints
    horizontal top
    horizontal bottom
    vertical left
    vertical right
    equalLength left right
    equalLength top bottom
    equalLength top left
```

Figure 10-2: An over-constrained definition for a rectangle. It contains the erroneous constraint EQUALLENGTH TOP LEFT, instead, defining a square.

horizontal constraint should be added). Redundant constraints (where both are true, but one is not necessary) are not considered errors as they do not affect the correct recognition of the shape.

10.2 Solution

To solve this problem, this research includes the development an algorithm using a novel form of active learning [51] that automatically generates its own suspected near-miss [214] examples, which are then classified as positive or negative by the developer. The algorithm is a modification of the traditional one of machine learning of concepts, in which a teacher supplies labeled examples (and non-examples) of the concept (e.g., "This is an arrow" "This is not an arrow"), and the system constantly updates its evolving version of the concept. This research suggests a change to this model, to one in which the system selectively generates its own (near-miss) examples, and uses the teacher as a source of labels. The system generates these examples to test whether components of its current concept description are necessary to the concept, or merely happened to be true of the initial example. For example, is it necessary for both lines in the head of an arrow to be the same length or was this accidental in the original example?

System-generated near-misses offer a number of advantages. They work even when the teacher does not know the complete concept description in advance; e.g., the teacher might not have thought about whether an arrow-like figure with unequal head lines is still an arrow, or whether an arrow can have a shaft that is shorter than its head lines. Also, the system can be an efficient learner simply by virtue of keeping careful track of which parts of the concept description have been verified as necessary and which are yet to be tested, thereby generating only informative examples, such as near-miss examples which differ in only one respect from the current concept.

The result is a system that behaves somewhat like a persistent, literal-minded, but intelligent student who wants to get all of the details right, and does so, by asking, "And would this be an example? How about this one? And this one...?" When learning a concept, while working within a fixed vocabulary and rule-set, the computer learner knows exactly where its uncertainties of the concept lie, and which hypotheses need to be tested and confirmed. For example, in a 2 line sketch, there are 114,000 total possible shapes that can be displayed.[1] Because there are so many possibilities, choosing examples that quickly reduce the space of possible variations is important. This requires that the learning participant who is generating the examples needs to keep track of all of the shapes that have been shown, which shapes are no longer possible, which shapes are still allowed, and which example shapes still to be shown can most effectively reduce the ambiguity that still remains, given a series of positive and negative shapes. Because of the large number of variations possible and the discrete nature of the task, a computer is more effective at generating the near miss examples.

Active learning is a dialogue between a teacher and a student. The goal of the dialogue is to teach the student a concept that is known by the teacher. Examples are

[1]Several of these shapes are impossible because of conflicting constraints. For example, POSSLOPE LINE1, HORIZONTAL LINE2, PARALLEL LINE1 LINE2.

selected, either by the teacher or the student, and the teacher labels these examples. Our learning model is based on the principle that the learning participant (either the teacher or the student) who knows better which information is lacking in the student's formation of the concept, should be the one who generates the near-miss examples.

In human-human (teacher-student) learning, humans are poor at knowing what they do not know; initially, the human teacher knows better what information the student is missing from his concept and provides the near-miss examples. However, as the student begins to learn the concept, at some point, there is a transfer of knowledge to the student. As the student begins to understand the concept, he knows what information still needs to be confirmed. At this point the student begins to generate his own near-miss examples, confirming and removing uncertainties, saying such things as, "Oh, I think I get it. So, is this an example? What about this one? Yes, that makes sense, now. I understand."

In our task of learning structural shape descriptions, the human developer is the teacher, and the computer is the student. Our situation is different from the human-human learning environment in that the task of learning structural shape descriptions occurs in a fixed domain with a fixed vocabulary and syntax. The computer student can easily keep all existing possibilities concurrently in memory and know exactly what information is necessary to confirm the current shape concept. Conversely, a human teacher is not good at keeping all possible uncertainties in her mind at one time. In this case, the computer-as-student is better able to provide informative near-miss examples, allowing it to more quickly and effectively refine a concept.

10.3 Initial Conditions

Our approach needs a positive hand-drawn example and a description that will correctly recognize that one example (Figure 10-3). The developer can choose to type the description or have one generated automatically from the hand-drawn example,

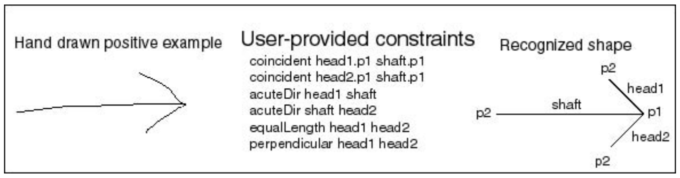

Hand drawn positive example
User-provided constraints
coincident head1.p1 shaft.p1
coincident head2.p1 shaft.p1
acuteDir head1 shaft
acuteDir shaft head2
equalLength head1 head2
perpendicular head1 head2
Recognized shape
p2
head1
p2
shaft
p1
head2
p2

Figure 10-3: Hand-drawn positive example and description.

using techniques developed by our group [210]. In either case, descriptions are built from the *LADDER* vocabulary of constraints.

We begin with the first steps of the debugging process for user-typed descriptions, because these require initial debugging steps not required for machine-generated descriptions.

10.3.1 Debugging User-Typed Descriptions

User-typed descriptions are first checked for syntactic validity, using *SHADY*. The system then checks to make sure that the initial description accepts the initial positive example using techniques described previously. If this (known to be correct) example is not recognized, the description must be over-constrained and needs to be corrected. The system displays the subcomponents of the failed constraints in red and asks the developer whether the indicated failed constraints could be removed to correct the description. If there are several variable assignments containing the minimum number of failed constraints, the system chooses all of them and displays the collection of failed constraints, one at a time.

The developer is required to remove enough constraints to permit her description to recognize the initial hand-drawn shape. This ensures that the process starts with a positive (hand-drawn) example and a description capable of recognizing it.

10.3.2 Automatically Generating a Description

If the initial description is generated by the system using techniques described earlier in this thesis, it is guaranteed to recognize the hand-drawn example.

10.3.3 The Initial Description

The important point now is that no matter how the description was entered (manually or computer-generated), the description now is known to recognize the hand-drawn positive example. Hence, the description is known *not* to be over-constrained with respect to the single example seen so far.

10.4 Over-constrained Descriptions

The current challenge is that, while the initial description may recognize the initial hand-drawn example, it may be over-constrained compared to the actual concept for the shape. The arrow in Figure 10-1 happens to have two perpendicular lines at its head; it is a positive example of an arrow, but over constrained in the sense that a figure without a perpendicular head is still an arrow. Hand-drawn examples will almost always be over-constrained because the sketcher is required to make arbitrary choices, and it is difficult for the computer to determine which choices are purposeful and which are accidental. Even if the sketcher had drawn an arrow with a non-perpendicular head, the initial hand-drawn example may still be over constrained, as an acute (or obtuse) constraint may be generated instead. The arrow and square in Figure 9-1, displayed in a previous chapter, express these difficulties.

10.4.1 Constraint Candidate List

Earlier, we indicated that the system generates the complete list of constraints true of the initial sketch. This list is saved and used as the initial value for a list called the *constraint candidate list*. Each time a positive example shape is encountered, the system removes from the constraint candidate list any constraints not true of the new positive example: Any constraint not true of a positive example cannot be true of the concept.

The system also generates a list of negative constraints. Each time the system encounters a negative example, it knows for certain that at least one constraint in the shape caused the negative example, but we do not know which one. It first removes all of the constraints already seen in a positive example, as it knows none of these constraints caused the negative example. At least one of the constraints in the negative constraint list caused the negative example. Each time the system encounters a negative shape with only one variable change, i.e., with only one negative constraint left in the negative constraint list, the system knows that constraint caused the negative example. In this case, it can add the single constraint to the list of constraints known to cause a negative example. Otherwise, the system saves the generated list of possible negative constraints for later processing.[2] Elements of this collection (lists with more than one constraint) may eventually be reduced to one constraint when some of the constraints are removed after a positive example shape.

10.4.2 Selecting the Constraints to Test

To remove all uncertainties, the system would have to generate all possible variations of the shape, which would imply testing all possible permutations of the constraints found in the original positive hand-drawn example. This would be unreasonable,

[2]Constraint lists of negative examples frequently contain more than one constraint because constraints are interrelated in the sense that one constraint cannot be falsified without falsifying another: an example shape in which two lines are constrained not to meet, for instance, is necessarily also an example in which those two lines are not connected.

requiring several hundred permutations for even a three-line shape. This research uses the perceptual knowledge gained from Gestalt principles to reduce the number of generated shapes.

Instead of generating a shape to test each constraint, the system chooses a small number of constraints which it feels are likely to be necessary in the final concept and tests those. This small number of constraints is chosen from the initial best-guess description created from a single hand-drawn example described above. A shape is then generated that tests that particular constraint.

10.4.3 Initial Over-constrained Testing

Because it is often the case that shapes can be rotated and scaled, the system first rotates and scales the shape and presents several examples all at once to the user. In the case of scaling, the developer is asked to indicate the status of each example individually; the individual positive and negative examples are handled as in the previous section.

For rotation, the user is permitted only to say whether or not *all* of the examples are positive. (We currently do this in order to avoid problems with shapes having rotational symmetry.) If the user indicates that all of the examples are positive, they are handled in the fashion described in the previous section.

10.4.4 Testing Other Constraints

The system checks to see whether the description is over-constrained by examining each constraint in turn, and generating a suspected near-miss shape to test whether that constraint is necessary. For example, in the description in Figure 10-4, the system tests the six listed constraints.[3]

[3]It is possible that a single constraint, when made false, produces a set of inconsistent constraints. In this case, the system removes a constraint that which shares an argument and retests.

A constraint is tested by creating a description in which the constraint is replaced by its negation, then a shape that fits this description is generated. (The shape generation technique is described below and in Appendix C). Figure 10-5 shows the shapes generated when testing (COINCIDENT HEAD1.P1 SHAFT.P1) and (PERPENDICULAR HEAD1 HEAD2).

Because the topology of a shape is considered to be the most perceptual property of a shape [210], the system tests all coincident constraints first, presenting several examples all at once to the developer. The system modifies the constraint candidate list to take into account the positive and negative examples as specified by the developer.

Imagine that the developer indicates that the shape generated by the revised description does not agree with her mental model of an arrow (as in the case of testing the COINCIDENT constraint of Figure 10-5). This shows that the constraint in question is a necessary part of the description because there exists both a positive example where the constraint is met (the originally hand-drawn shape) and a negative example in which the only thing changed is that the constraint is now not met (the generated shape).

Imagine, on the other hand, that the developer indicates that the shape generated by the revised description does agree with her mental model of an arrow (as in the case of testing the PERPENDICULAR constraint of Figure 10-5). Thus, the original description was over-constrained: the constraint is superfluous since there exists a positive example in which the constraint is not met.

10.5 Under-constrained Descriptions

Once the shape description is known to be not over-constrained, the system checks whether it is under-constrained by making a list of possible missing constraints. As the list of possible missing constraints can be very large, the system generates it by

```
define shape Arrow
  components
    Line shaft
    Line head1
    Line head2
  constraints
    coincident head1.p1 shaft.p1
    coincident head2.p1 shaft.p1
    acuteMeet head1 shaft
    acuteMeet shaft head2
    equalLength head1 head2
    perpendicular head1 head2
```

Figure 10-4: An over-constrained description of an arrow; it should not contain the constraint PERPENDICULAR HEAD1 HEAD2.

the same filtering process used in Section 10.3.2, with several additional filters: The system also removes constraints that are more general than and that follow transitively from those in the current description. The system then chooses n^2 constraints from this list (using the same perceptual ranking scheme as mentioned previously) to test for possible accidental exclusion.

The system tests each of those n^2 constraints to determine whether it is missing from the description by adding its negation to the description (e.g., NOT HORIZONTAL SHAFT), then generating a shape based on this description. Figure 10-7 shows two generated possible near-miss examples which test constraints HORIZONTAL SHAFT and LONGER SHAFT HEAD1.

Imagine that the developer indicates that the non-horizontal example in Figure 10-7 agrees with her mental model of an arrow. Because the system contains a positive example of an arrow with the constraint met (the original hand-drawn shape) and not met (the generated shape), the system concludes that the constraint should not be included (i.e., it was an accident that the original arrow happened to be drawn horizontally).

However, imagine that the developer indicates that the NOT LONGER example in

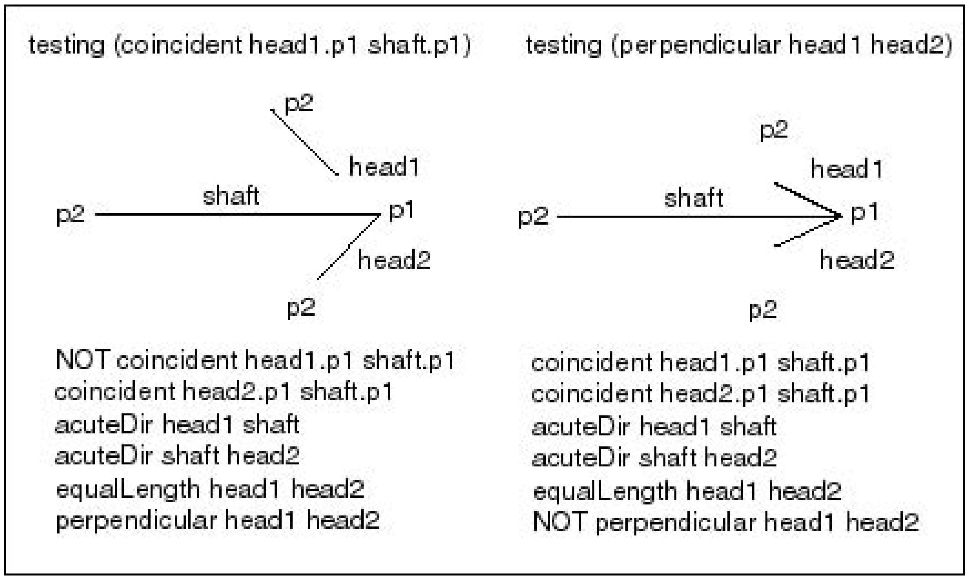

testing (coincident head1.p1 shaft.p1)
p2
head1
p2
shaft
p1
head2
p2
NOT coincident head1.p1 shaft.p1
coincident head2.p1 shaft.p1
acuteDir head1 shaft
acuteDir shaft head2
equalLength head1 head2
perpendicular head1 head2
testing (perpendicular head1 head2)
p2
head1
p2
shaft
p1
head2
p2
coincident head1.p1 shaft.p1
coincident head2.p1 shaft.p1
acuteDir head1 shaft
acuteDir shaft head2
equalLength head1 head2
NOT perpendicular head1 head2

Figure 10-5: Near-miss examples testing whether description is over-constrained.

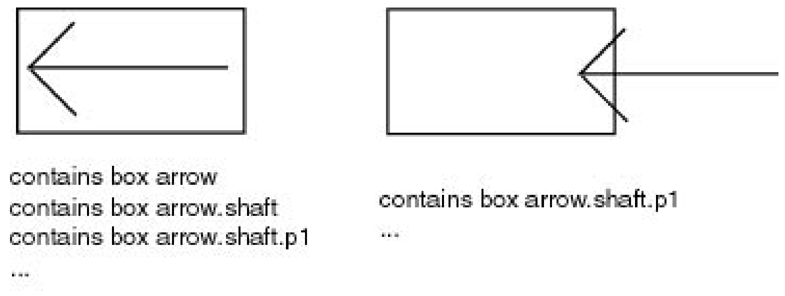


Figure 10-6: Hierarchical examination of the CONTAINS constraint.

Figure 10-7 is not an arrow. In this case, the system contains two examples with identical descriptions except for the LONGER constraint; the (hand-drawn) shape in which the constraint is met is a positive example, and the (generated) shape in which the constraint is not met is a negative example. This indicates that the constraint is necessary to the concept of the shape.

10.5.1 Generating Shapes

To test if a constraint is necessary, the system generates a list of constraints with all of the constraints true in the initial positive example, and the negation of the constraint to be tested. For example, to test if LONGER LINE2 LINE3 is required,

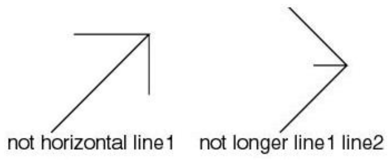


Figure 10-7: Possible near-miss shapes for an arrow.

then the system removes that constraint from the initial description and replaces it with either LONGER LINE2 LINE3 or LONGER LINE3 LINE2. The system submits the constraint list to MATLAB, which then produces a shape that satisfies those constraints. The system then shows this newly generated shape to the user, asking the user to label it as positive or negative. The collection of positive and negative examples is used to generate a shape description that properly classifies all examples.

It is possible that the list of constraints submitted to MATLAB cannot produce a feasible shape. In this case, MATLAB will produce a closest feasible solution. The system will then examine this generated shape and its true constraints. If the system already has a labeled example with the same constraints true, or if the system can already classify the shape (with certainty) based on the previously labeled examples, the system will not show the shape to the user. Otherwise, if the system cannot yet classify the shape, then labeling this solution will still provide extra information, and the system will display the shape to the user. Likewise, sometimes MATLAB will fail to produce a feasible solution even when one exists because it got stuck in a local minima. In this case, the system acts as if it cannot produce a feasible shape (as described above), since it cannot tell the difference. Because the desired shape may not be generated, the system may not be able to fully determine the exact desired shape description, but, hopefully, the displayed near-miss examples will help to effectively converge the shape description to the correct answer.

10.5.2 Some Limitations

The technique described in this chapter has been tested solely on shapes composed of lines and circles. Future research includes plans to incorporate curves and arcs in the near future. At this point, we are uncertain of the difficulties that incorporating arcs and curves might cause.

Chapter 11

Concept Learning Algorithm

The system uses these labeled examples to automatically build a *LADDER* shape description using a modification of the version spaces algorithm that enables it to handle interrelated constraints and has the ability to learn negative and disjunctive constraints.

The algorithm holds a collection of possible concepts called a concept space. At least one concept is correct at all times, although it is possibly too general. The algorithm assumes that all data is properly labeled, and that the final concept can be described using the constraints provided in the *LADDER* language. The algorithm starts with an initially empty concept which accepts everything. Concepts are refined, created, and pruned with positive and negative examples.

11.1 Creating the Initial Concept

A shape concept specifies what constraints can be true in a drawn shape in order for it to be recognized. The constraints are grouped together based on the shape property that they measure (referred to in this thesis as *measurable properties*). All of the constraints in a single group are designed to be *mutually exclusive* from the

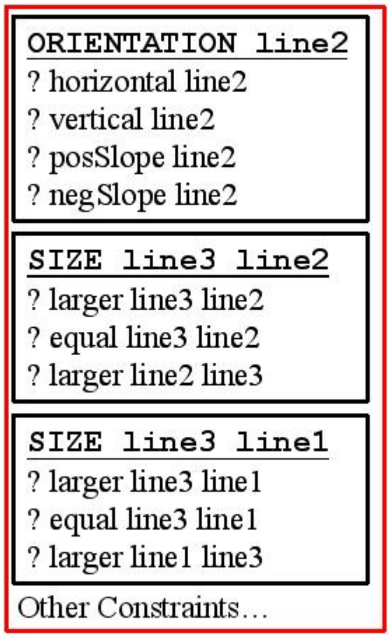
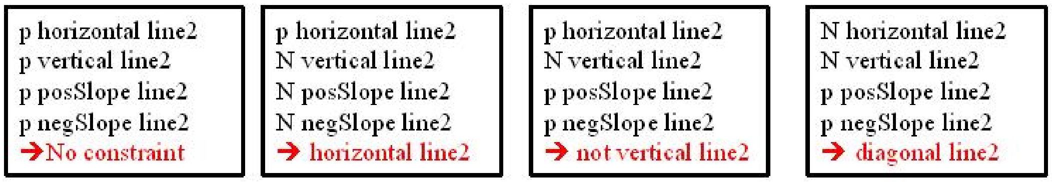


Figure 11-1: Abbreviated initial concept for a three-lined shape.

others in the same group. An example of a measurable property is the orientation of a line, the mutually-exclusive constraints on line orientation are HORIZONTAL, VERTICAL, POSSLOPE, and NEGSLOPE. The number of variables in a shape concept is equal to the number of measurable constraints for a shape, which can be computed from the number and type of subshapes. A shape consisting of three lines will have more measurable constraints than a shape consisting of two lines. For example, a three-line shape can have three orientation constraints, three relative size constraints, and 27 coincident constraints, as well as other types of constraints. Figure 11-1 shows a shortened initial concept space for a three-line shape. If we do not know whether a constraint is allowed in a positive example, we label the constraint with a question mark. Once a constraint is seen in a positive example, the system replaces the question mark with a *p*. If we know that a constraint causes a negative example, the system replaces the question mark with an *N*.

Figure 11-2 shows several examples of orientation of a line after the *p*'s and *N*'s have been learned; the constraints implied from these multi-valued variables are shown.



p horizontal line2
p vertical line2
p posSlope line2
p negSlope line2
➔No constraint

p horizontal line2
N vertical line2
N posSlope line2
N negSlope line2
➔ horizontal line2

p horizontal line2
N vertical line2
p posSlope line2
p negSlope line2
➔ not vertical line2

N horizontal line2
N vertical line2
p posSlope line2
p negSlope line2
➔ diagonal line2

Figure 11-2: Constraints implied by multi-valued variable for line orientation.

11.1.1 Refining the Concepts

For each positive example, the algorithm labels each constraint present in the positive example with a *p* in the concept. Figure 11-4 shows the arrow concept after the initial positive example shown in Figure 11-3. Note that HORIZONTAL LINE2, LARGER LINE3 LINE2, and LARGER LINE3 LINE1 were all true in the positive example in Figure 11-3, and that all of those constraints were labeled with a *p* in the concept presented in Figure 11-4.

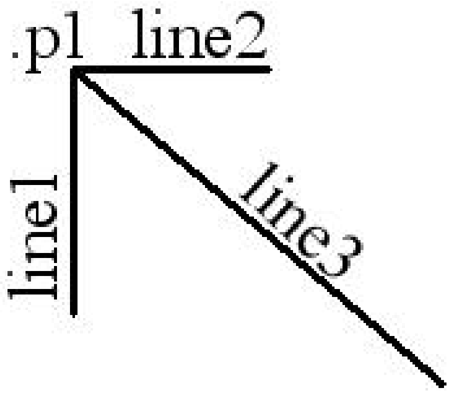
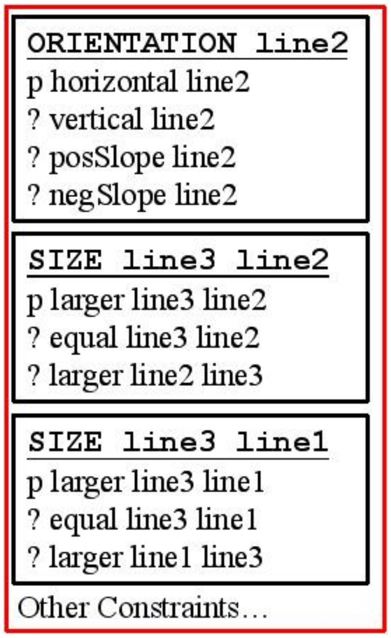


Figure 11-3: Initial positive example of an arrow.

Because constraints are often interrelated in our domain of structural shape descriptions, it may be hard to tell which constraint caused a negative example. For example, in the negative example shown in Figure 11-5, it is unclear which constraint – (LARGER LINE2 LINE3) or (LARGER LINE3 LINE2) (or both) – caused the negative example. Thus, the system must divide each shape concept into two shape concepts, where one concept states that (LARGER LINE2 LINE3) caused the negative example, and the other states that (LARGER LINE3 LINE2) caused the negative example.



ORIENTATION line2
p horizontal line2
? vertical line2
? posSlope line2
? negSlope line2

SIZE line3 line2
p larger line3 line2
? equal line3 line2
? larger line2 line3

SIZE line3 line1
p larger line3 line1
? equal line3 line1
? larger line1 line3

Other Constraints…

Figure 11-4: Arrow concept after initial positive example.

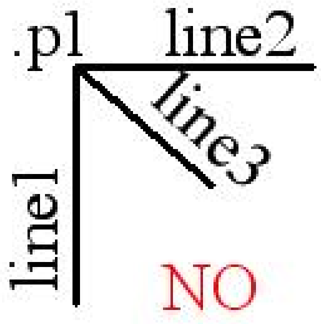
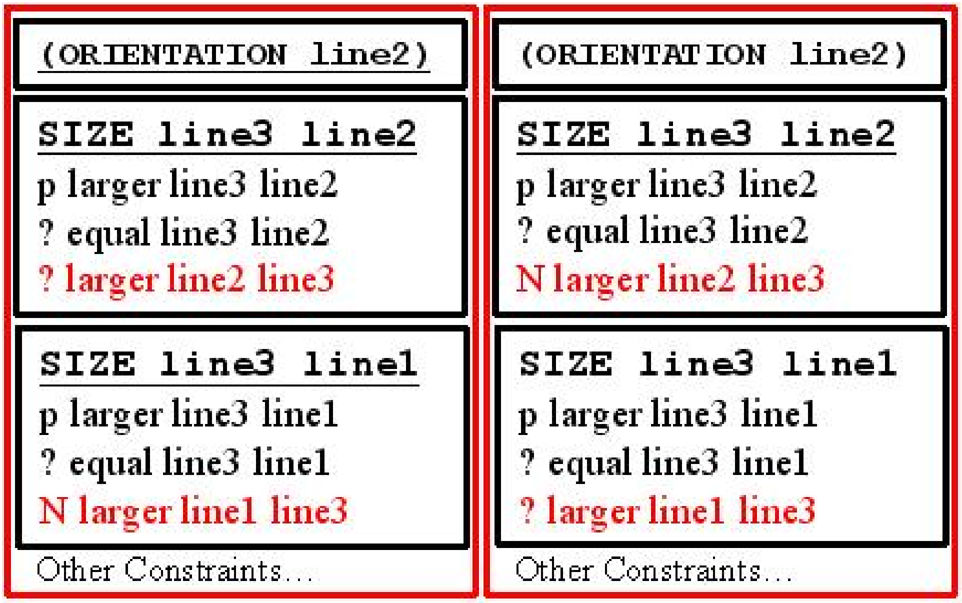


Figure 11-5: Negative arrow example.

11.1.2 Converging to a Single Concept: Purple Cow Heuristic

Many concepts can be created during the branching process. However, shape concepts are naturally pruned while learning. Duplicate concepts are often created and will be pruned. The algorithm also prunes examples that imply impossible concepts which place a p over an N.

If all shape permutations are generated, it is possible to converge eventually to


(ORIENTATION line2)

SIZE line3 line2
p larger line3 line2
? equal line3 line2
? larger line2 line3

SIZE line3 line1
p larger line3 line1
? equal line3 line1
N larger line1 line3

Other Constraints...

(ORIENTATION line2)

SIZE line3 line2
p larger line3 line2
? equal line3 line2
N larger line2 line3

SIZE line3 line1
p larger line3 line1
? equal line3 line1
? larger line1 line3

Other Constraints...

Figure 11-6: Two possible concepts after a negative example.

a single concept using this algorithm. However, producing all of the permutations is undesirable because that would mean that the user would have to label several hundred examples. Ideally, few negative examples would cause concept branching (as branching may require many more examples to prune down again to a single concept); however, this is difficult to do as many of the geometric constraints are interrelated. Thus, this researcher has developed a heuristic called the *purple cow heuristic* to reduce to a single concept.

The *purple cow heuristic* works as follows: "I have never seen a purple cow, so I am going to assume one does not exist." This heuristic, applied to the structural shape domain, is reworded as, "I have never seen this constraint exist in a positive example shape, thus I am going to assume it will never exist in a positive example shape." The system does this at an application level by updating all *?*'s labels to *N*'s.

Any constraint seen in any positive example will necessarily have a *p* labeling in all possible concepts in our concept space and will never be mislabeled in our final concept using the purple cow heuristic. However, a *?* may be mistakenly be changed into a *N*.

Because constraints are often interrelated, and because it is impossible to generate shapes with certain constraint combinations, it may be impossible for the system to gain sufficient proof that a certain constraint caused a negative example. For example, we cannot create a near-miss example to test an arrow that keeps LINE3 longer than LINE2, LINE2 equal to LINE1, while making LINE3 shorter than LINE1. Because of these features of structural shape descriptions, This heuristic works particularly well for our problem.

11.1.3 Creating the Final Description

Our final single concept is then translated to a description by translating each multi-valued variable to the appropriate constraint. We then remove redundant or transitive constraints, and output our final description. This description can then be used to automatically generate a recognizer from that description, using the techniques and system described above. Figure 11-7 provides an example.

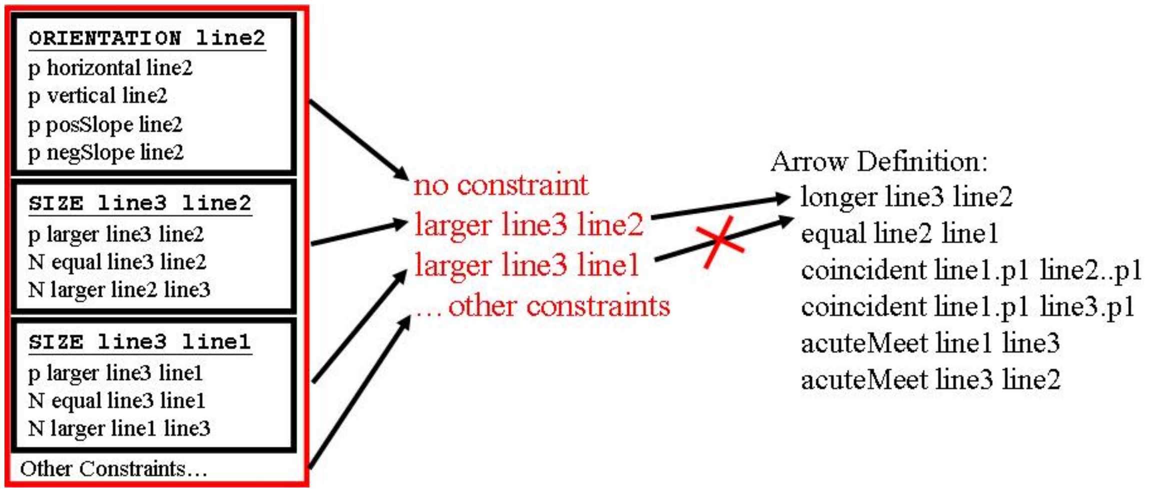


Figure 11-7: Converting the final concept to a shape description. Each bucket is first converted to a single (or no) constraint. Redundant constraints are then removed using techniques described earlier.

Chapter 12

System Interaction

Here is an overview of the process a developer would take to create a sketch interface using the techniques from this thesis. Note that this process is considerably faster than writing arecognition system from scratch.

1. Developer makes a list of all of the shapes in the domain.
2. Developer produces a description of all of the shapes in the domain:
   (a) Developer draws an example shape.
   (b) The computer automatically generates a best-guess description (or the developer can choose to manually type one herself).
   (c) The computer checks that the description is not over-constrained. For each constraint in the current description...:
      i. The computer generates one suspected near-miss example shape that tests whether that constraint is required.
      ii. The developer specifies whether or not the shape is a valid example shape.
      iii. The computer uses this knowledge to include or remove the constraint.

(d) The computer checks that the description is not under-constrained. The computer generates a manageable (less than 20) number of constraints that may be missing. For each possible new constraint:

    i. The computer generates one suspected near-miss shape that tests if that constraint is required.

    ii. The developer specifies whether or not the shape is a valid example shape.

    iii. The computer uses this knowledge to include or remove the constraint.

(e) The developer then manually specifies how the shape should be displayed and edited.

3. Each description is then translated into a recognizer for that shape, specifying how the shape should be recognized, displayed, and edited.

4. The new code is compiled into a sketch recognition user interface for that domain.

5. The sketch recognition user interface can be run, and it will recognize, display, and allow editing of the shapes described in the description.

Chapter 13

User Study: Lessons Learned about the Successes and Difficulties of Near-miss Generation

To date, ten different users have generated shapes using the automated near-miss generator. Eight of them were asked to create toy-example shapes; the rest were asked to develop a sketch system for a particular domain. All of the interactions were informal. The toy-example group was asked to generate between three and six shapes. Depending on the difficulty the user encountered, users were urged to try more or fewer examples. Problems that were readily identifiable were fixed between user sessions. To identify more serious problems and to stress-test the system, users were given both examples that this researcher had tried before and those that had not yet been tried by anyone.

13.1 Methodology

Users were first presented with the debugging system and given brief verbal instructions. They were then observed as they stepped through the system. Help was provided when needed. Users ranged from age 22-72. All were computer proficient, with computers playing a significant part in their daily work activities. Users were highly educated; all were either in or had completed some form of graduate school. Several of the users were in computer science, but others were in fields such as mathematics, business, education, and sociology. A selection of the shapes that users were asked to draw is in Figure 13-1.

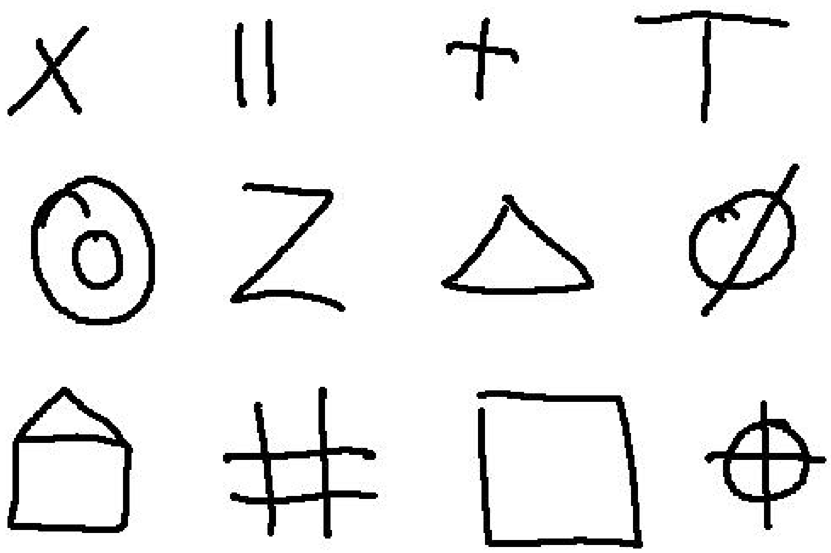


Figure 13-1: Some of the shapes that users were asked to draw.

The process occurred as follows:

1. The user draws a positive example as in Figure 13-2.

2. The system internally generates a list of true constraints, and automatically generates a shape that satisfies those constraints. The system shows this exam-

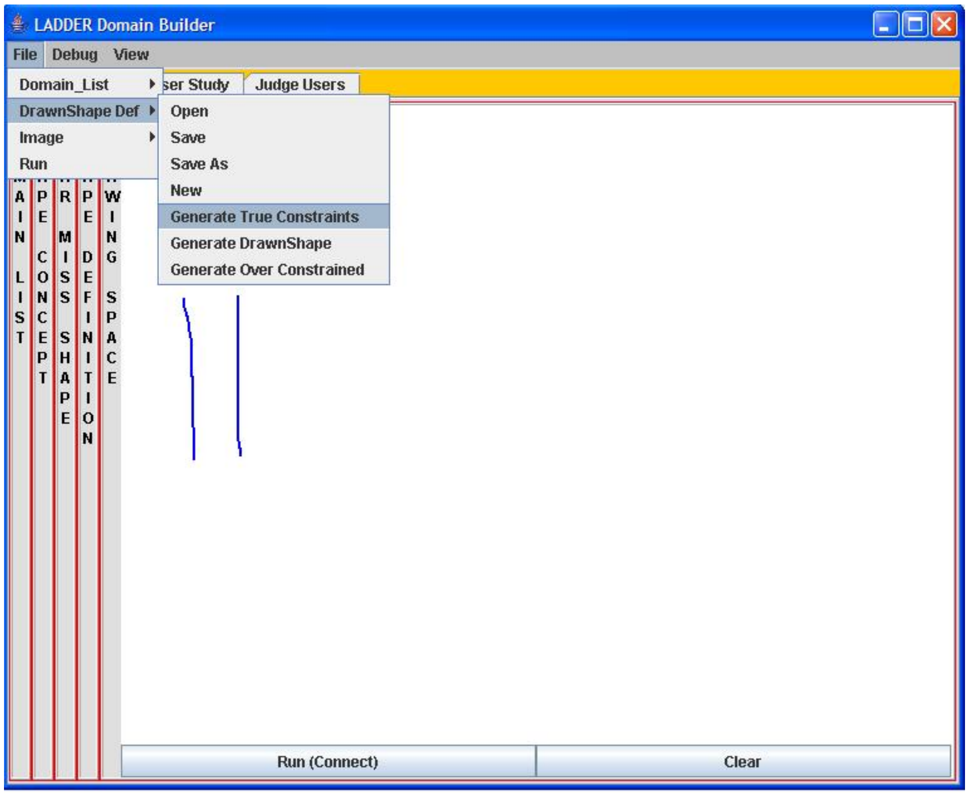


Figure 13-2: The user draws a shape.

ple to the user, as in Figure 13-3.

3. If the generated shape is incorrect, the user starts again, redrawing the shape.

4. If the generated shape is correct, the system selects a subset of all of the true constraints as a first best guess, and shows this list to the user, as shown in Figure 13-4. In particular, the system selects only the most perceptually important constraints. The list is shown to the user 1) to engender user autonomy. Our purpose is to enable the user, not to force him or her to use our near-miss debugging system. And 2) to provide transparency to the systems current shape concept and the assumptions made. By providing transparency, we hope that the system's actions make more "sense" to the user, and the user feels that there is a purpose to their actions, rather than providing information to an unresponsive system.

5. In order to allow for user autonomy, the user may stop and accept this definition

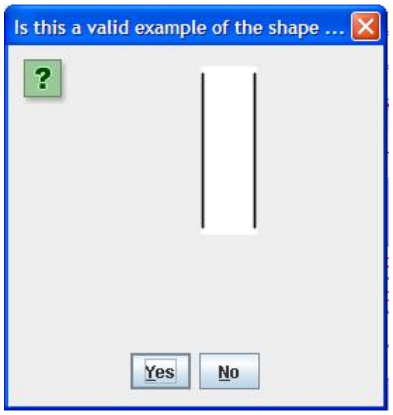


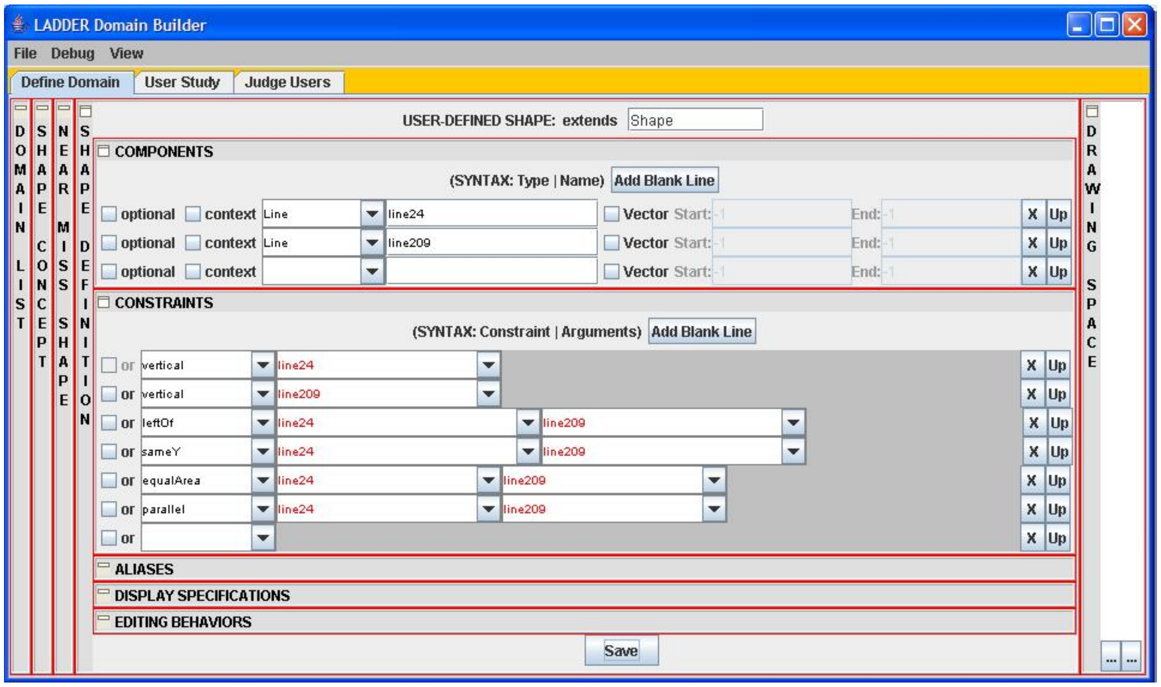
Figure 13-3: The system re-generates the drawn shape to make sure it is understood.

(now or at any time in the future).

6. Otherwise, the user can debug the description. The user is presented four near-miss examples at a time, as shown in Figure 13-5. The user clicks the examples to label them as positive (green) or negative (red) examples, as shown in Figure 13-6.

7. The system updates the description, as shown in Figure 13-7. The user can stop and accept the description at any time.

8. The system will continue to show examples (further examples shown in Figures 13-8, 13-9, 13-10, and 13-11) until the user stops and accepts the description, or when it has learned the description as best as possible, and there are no more examples to show. Using the learning algorithm described in the previous examples, the system generated all possible two line shapes (a total of 4,773 perceptually different shapes) in only 20 examples, or five pages of near-miss examples. However, the system reached its final guess (i.e., it did not change its guess) after the first page of four near-miss examples.



USER-DEFINED SHAPE: extends Shape

COMPONENTS

(SYNTAX: Type | Name)

optional	context	Line	line24	Vector Start: -1	End: -1
optional	context	Line	line209	Vector Start: -1	End: -1
optional	context			Vector Start: -1	End: -1

CONSTRAINTS

(SYNTAX: Constraint | Arguments)

or	vertical	line24	
or	vertical	line209	
or	leftOf	line24	line209
or	sameY	line24	line209
or	equalArea	line24	line209
or	parallel	line24	line209
or			

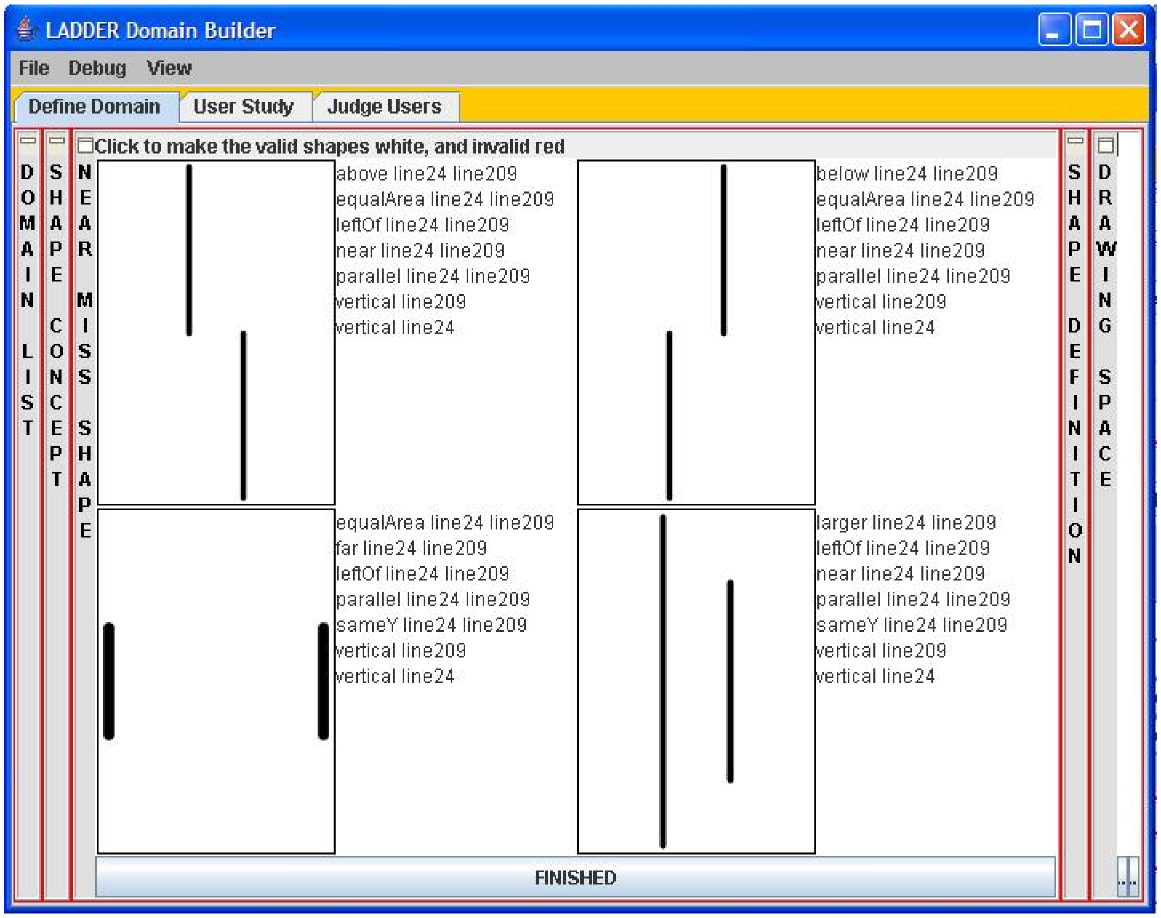
Figure 13-4: The initial description shown to the user.

13.2 Success and Failures

The results of the user study were not as expected, and the near-miss generator often failed or went un-used, as described by the following sections. However, when the system generated a correct solution, it did so quickly and effectively. As noted earlier, the system could reduce the space of 4,773 perceptually different shapes with only 20 examples, and came across the final description after only 4 examples (or even fewer, as explained below).

13.2.1 Generating All Possible Shapes

Because of the difficulties of generating specific near-miss examples, it was not clear whether the system was doing an exhaustive search of the space. Thus, to test the learning algorithm separately from the near miss generation (producing the results stated in the previous paragraph), we automatically generated all of the perceptual examples of the shapes. This was done in two ways. The first method asked Matlab



Click to make the valid shapes white, and invalid red

above line24 line209
equalArea line24 line209
leftOf line24 line209
near line24 line209
parallel line24 line209
vertical line209
vertical line24

below line24 line209
equalArea line24 line209
leftOf line24 line209
near line24 line209
parallel line24 line209
vertical line209
vertical line24

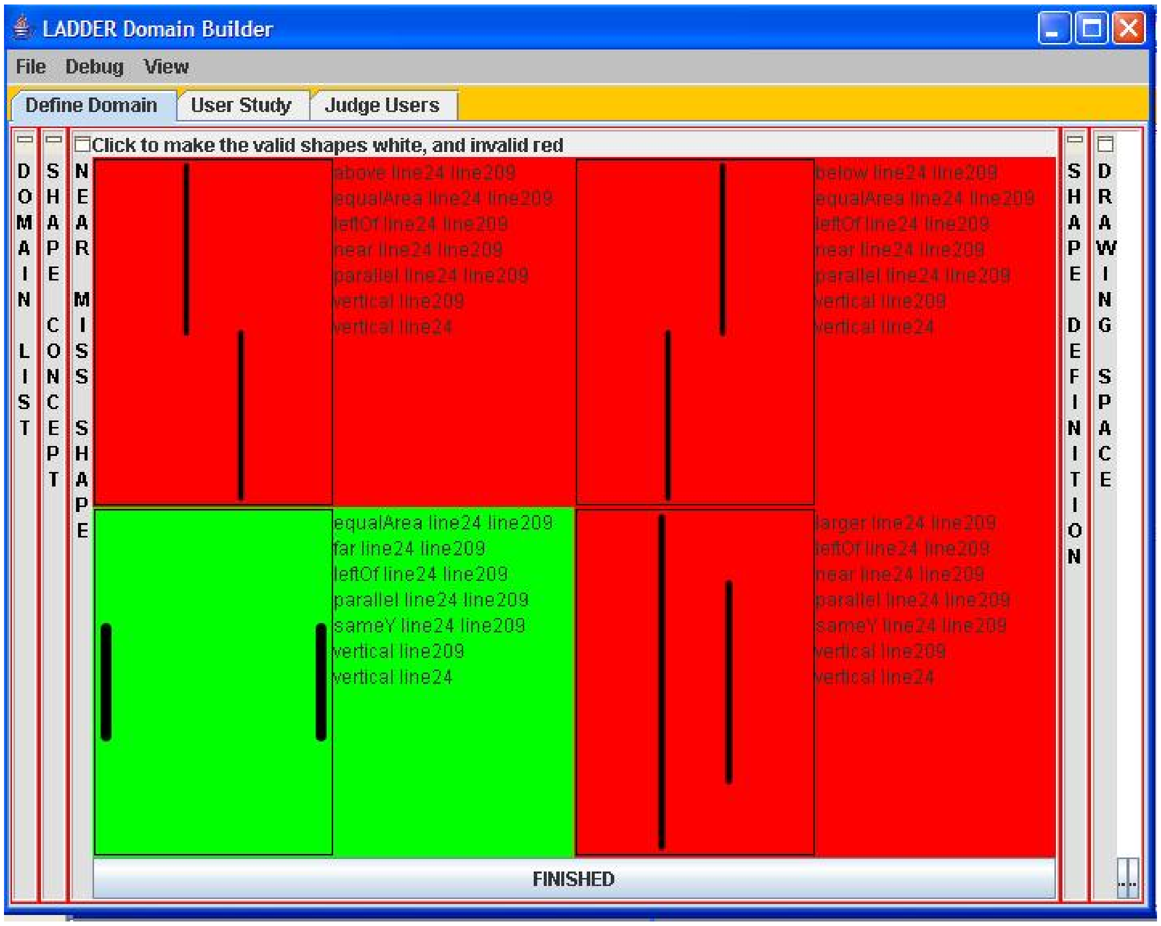
equalArea line24 line209
far line24 line209
leftOf line24 line209
parallel line24 line209
sameY line24 line209
vertical line209
vertical line24

larger line24 line209
leftOf line24 line209
near line24 line209
parallel line24 line209
sameY line24 line209
vertical line209
vertical line24

Figure 13-5: The first set of near-miss examples shown to the user.

to generate every possible constraint combination, running it 10 times with different starting conditions for each failed shape. Unfortunately, the system still could not guarantee that all possible shapes were generated - in fact, at least one such shape was found to be missing - and it took two weeks of CPU time on a supercomputer.

The second method took a ten by ten square to be used as a condensed representation of a 200 by 200 square, but since the shapes were to be perceptually different, this multiplicative factor of 20 proved useful. This researcher first created all possible two-lined shapes that could be created from the space. This involved 10 possible values for each x and y value, 10^2 possible values for each point, 10^4 values for each line, and 10^8 possible values for each two lined shape. This researcher removed lines of zero length and duplicate lines (since each line could be represented twice, once for each direction). Then, the system generated a list for each shape containing all con-



Click to make the valid shapes white, and invalid red

```
above line24 line209
equalArea line24 line209
leftOf line24 line209
near line24 line209
parallel line24 line209
vertical line209
vertical line24
```

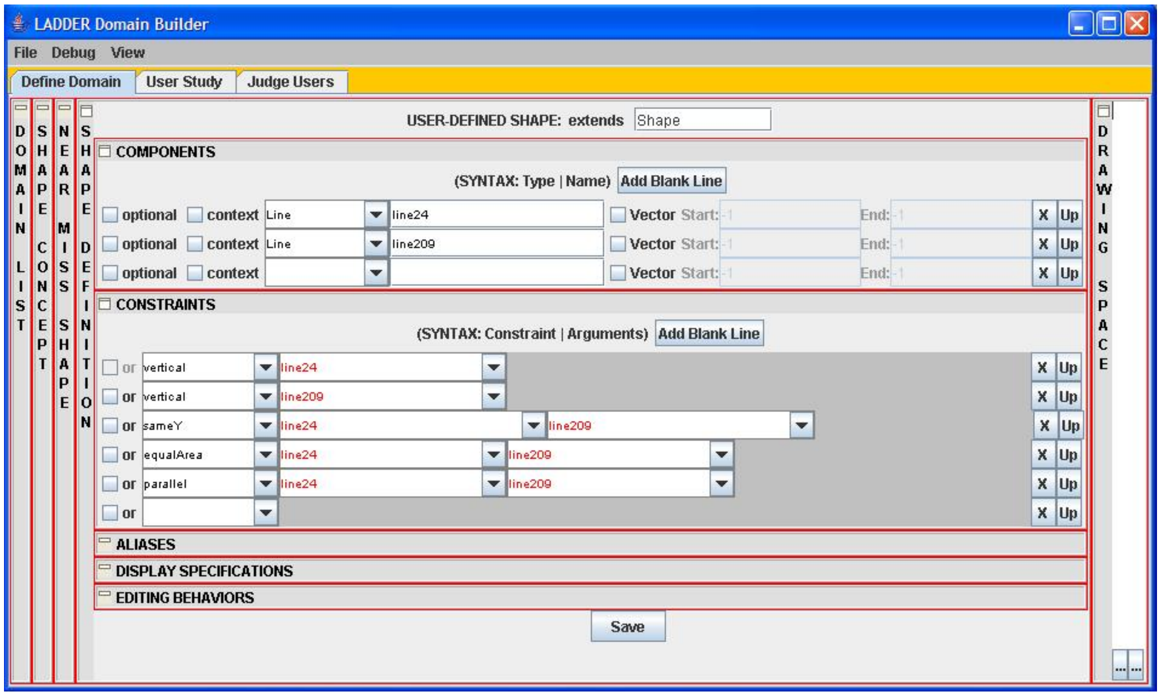
```
below line24 line209
equalArea line24 line209
leftOf line24 line209
near line24 line209
parallel line24 line209
vertical line209
vertical line24
```

```
equalArea line24 line209
far line24 line209
leftOf line24 line209
parallel line24 line209
sameY line24 line209
vertical line209
vertical line24
```

```
larger line24 line209
leftOf line24 line209
near line24 line209
parallel line24 line209
sameY line24 line209
vertical line209
vertical line24
```

Figure 13-6: The first set of the near-miss examples after the user has labeled the positive and negative examples.

straints true for that shape, and computed an error value representing their distance from the ideal representation of those constraints. The error value was computed similarly to those described in Chapter C. When multiple shapes generated the same list of perceptually true constraints, the system kept only the shape with the smallest error. This guaranteed that all possible shapes were generated, and, in fact, took a much shorter time doing it - approximately a day and a half running on a 1.73 GHz laptop, with other work processes running concurrently.



USER-DEFINED SHAPE: extends Shape

COMPONENTS

(SYNTAX: Type | Name)

optional context Line line24 Vector Start: -1 End: -1
optional context Line line209 Vector Start: -1 End: -1
optional context Vector Start: -1 End: -1

CONSTRAINTS

(SYNTAX: Constraint | Arguments)

or vertical line24
or vertical line209
or sameY line24 line209
or equalArea line24 line209
or parallel line24 line209
or

ALIASES

DISPLAY SPECIFICATIONS

EDITING BEHAVIORS

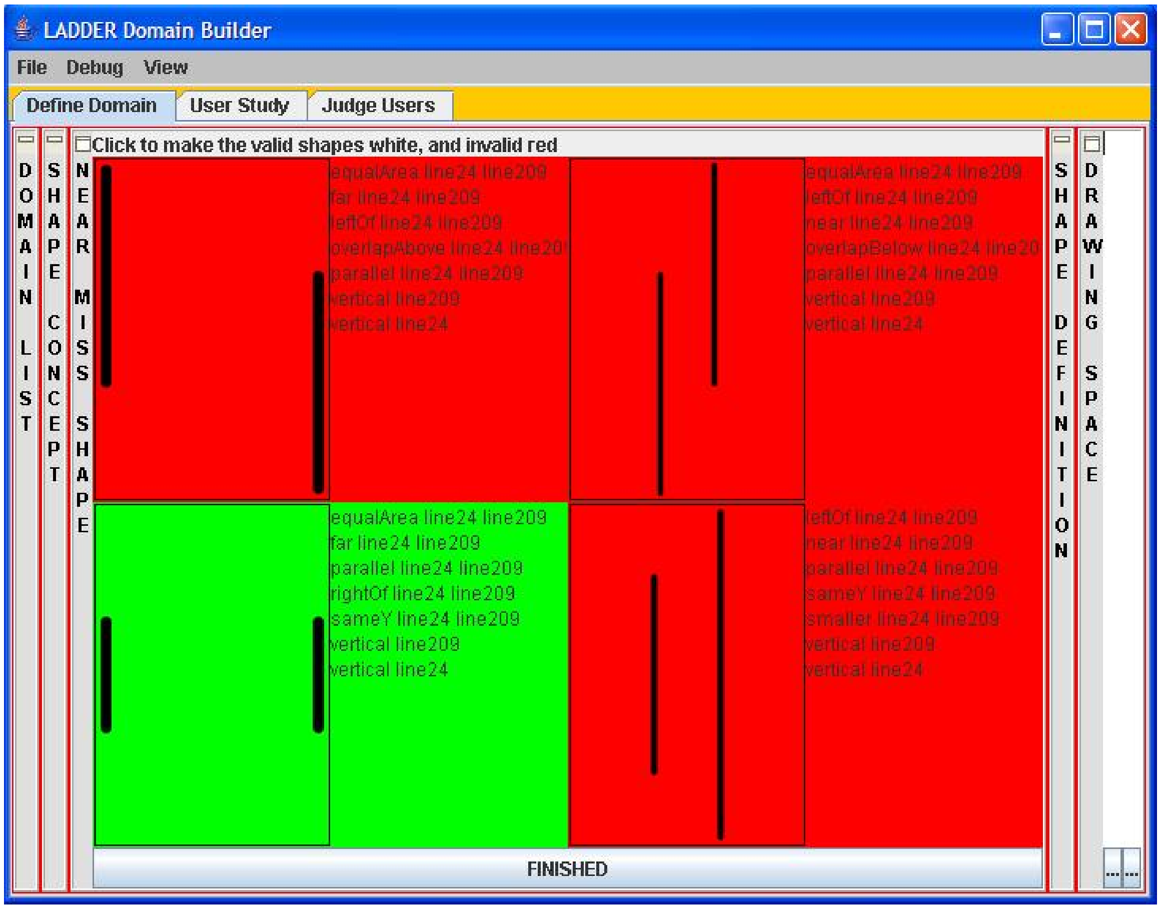
Figure 13-7: The updated description.

13.3 Lessons Learned

This section outlines some of the lessons learned from the study.

13.3.1 Lesson 1: Users Could Draw Shapes, but Needed Training

In order to generate a description, the user had to first draw an example shape. Provided the example shape was drawn relatively carefully, it was properly reproduced by the system. Users who were unfamiliar with a tablet pc had trouble initially. Their initial pen strokes sometimes did not lay a continuous stream of ink, or their strokes were rushed and ambiguous, even to the human eye. All of the users were eventually able to draw shapes that could be understood (and re-drawn) by the system.



Click to make the valid shapes white, and invalid red

```
equalArea line24 line209
far line24 line209
leftOf line24 line209
overlapAbove line24 line20
parallel line24 line209
vertical line209
vertical line24
```

```
equalArea line24 line209
leftOf line24 line209
near line24 line209
overlapBelow line24 line20
parallel line24 line209
vertical line209
vertical line24
```

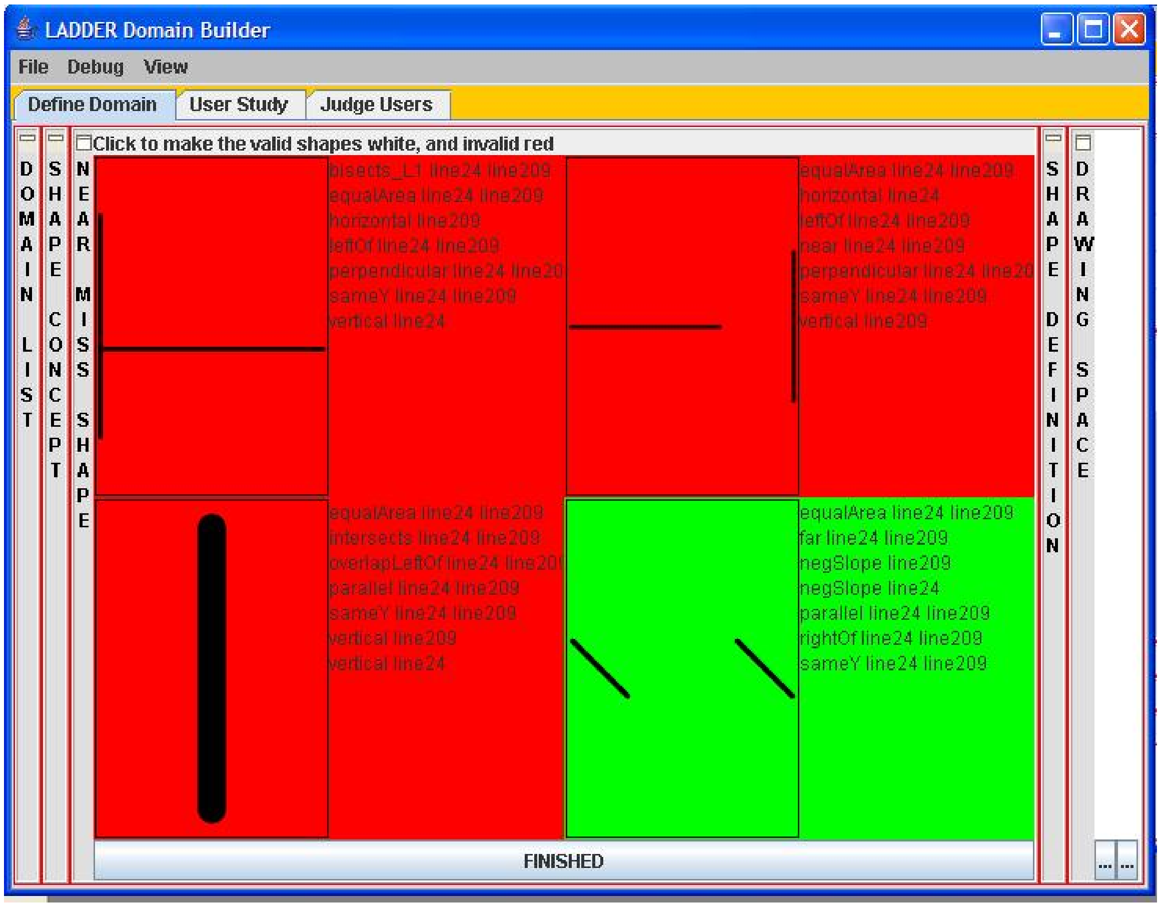
```
equalArea line24 line209
far line24 line209
parallel line24 line209
rightOf line24 line209
sameY line24 line209
vertical line209
vertical line24
```

```
leftOf line24 line209
near line24 line209
parallel line24 line209
sameY line24 line209
smaller line24 line209
vertical line209
vertical line24
```

Figure 13-8: The second set of labeled near-miss examples.

13.3.2 Lesson 2: Users Liked the Automatically-generated Description

After the system displays the understood-shape to the user, it generates a first best-guess and shows it to the user. Users were quite happy with the automatically-generated shape description. After the generated description was displayed, users would exclaim, with phrases such as: "Oh wow. Yes, that's it." or "Okay, what's the next shape?" In effect, they felt that the system had correctly described their shapes. The two users who were designing a domain system would often stop after the initial automatically-generated solution because it was either correct, or so close to what they wanted that it required minimal manual tweaking. The toy-examples contained some problems that specifically needed to be modified and were impossible to generate



bisects_L1 line24 line209
equalArea line24 line209
horizontal line209
leftOf line24 line209
perpendicular line24 line20
sameY line24 line209
vertical line24

equalArea line24 line209
horizontal line24
leftOf line24 line209
near line24 line209
perpendicular line24 line20
sameY line24 line209
vertical line209

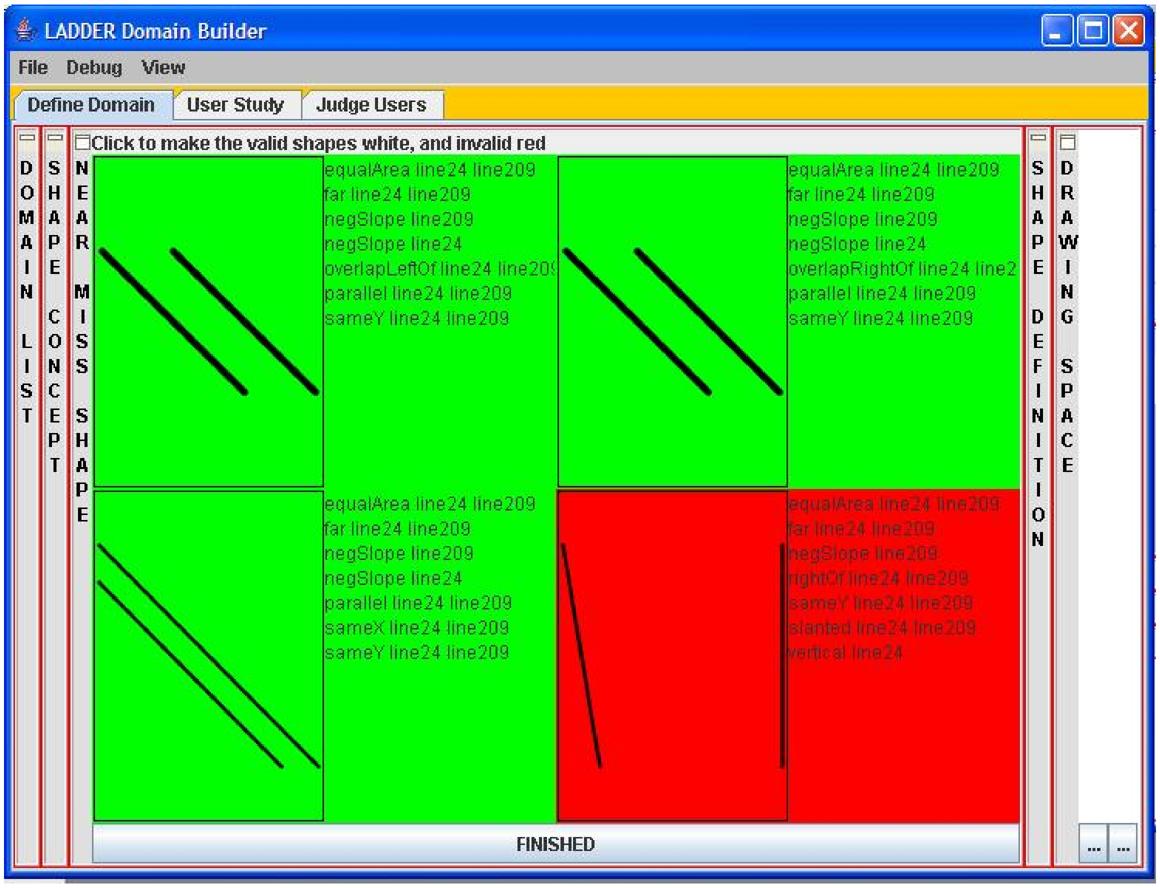
equalArea line24 line209
intersects line24 line209
overlapLeftOf line24 line20
parallel line24 line209
sameY line24 line209
vertical line209
vertical line24

equalArea line24 line209
far line24 line209
negSlope line209
negSlope line24
parallel line24 line209
rightOf line24 line209
sameY line24 line209

Figure 13-9: The third set of labeled near-miss examples.

solely from a single example. For example, one person was asked to draw two parallel equal-length lines that were rotatable, but not horizontal. Because the lines can be drawn only as positively-sloped, negatively-sloped, or vertical, it is impossible to for the computer to determine from a single example that the lines cannot be horizontal, but they can be other orientations.

The other advantage of the initial automatically generated description is that it was fast and efficient for even large shapes with many components. It was shown to be a big time-saver for generating descriptions of large shapes. Unfortunately, the users who were building domain descriptions chose, instead, to manually tweak descriptions of large shapes, rather than opt for the slow near-miss process, which was unbearably slow for large shapes.



```
equalArea line24 line209
far line24 line209
negSlope line209
negSlope line24
overlapLeftOf line24 line209
parallel line24 line209
sameY line24 line209
```

```
equalArea line24 line209
far line24 line209
negSlope line209
negSlope line24
overlapRightOf line24 line2
parallel line24 line209
sameY line24 line209
```

```
equalArea line24 line209
far line24 line209
negSlope line209
negSlope line24
parallel line24 line209
sameX line24 line209
sameY line24 line209
```

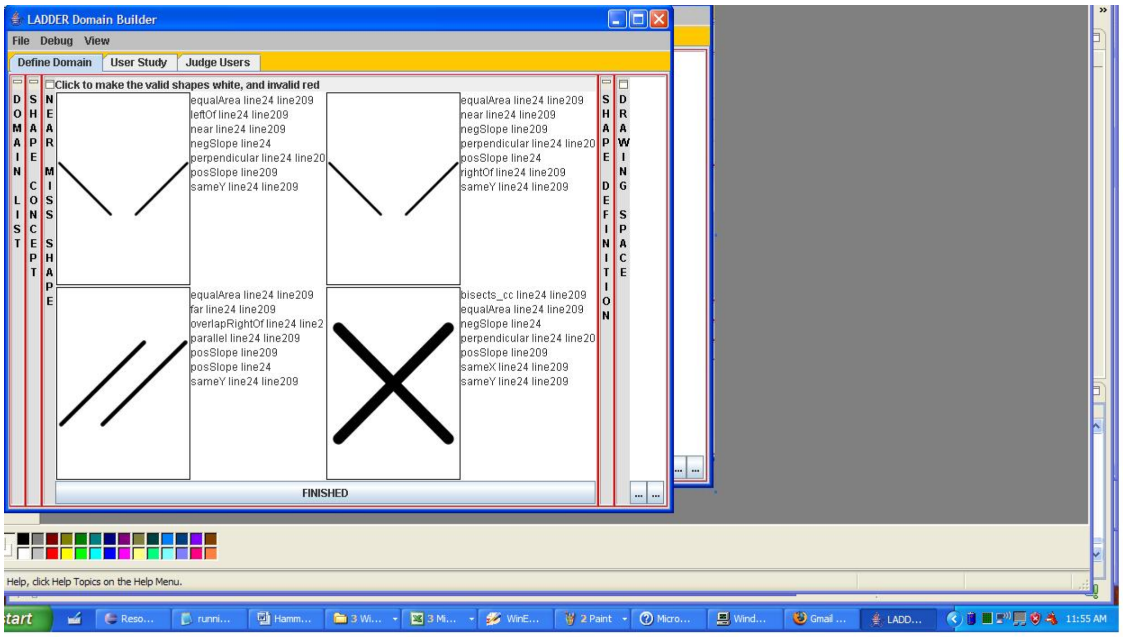
```
equalArea line24 line209
far line24 line209
negSlope line209
rightOf line24 line209
sameY line24 line209
slanted line24 line209
vertical line24
```

Figure 13-10: The fourth set of labeled near-miss examples.

Figure 13-12 shows a no smoking symbol which was quickly and correctly generated. (However, for ease of reading, the user may wish to replace SAMEX and SAMEY with a single CONCENTRIC constraint.) The list of chosen constraint is:

- posSlope line1
- equalSize line1 ellipse1
- sameX line1 ellipse1
- sameY line1 ellipse1

Figure 13-13 shows an image of a house for which a description was automatically generated. The long list of all the true constraints for the system to chose from is shown below:



Click to make the valid shapes white, and invalid red

equalArea line24 line209
leftOf line24 line209
near line24 line209
negSlope line24
perpendicular line24 line20
posSlope line209
sameY line24 line209

equalArea line24 line209
near line24 line209
negSlope line209
perpendicular line24 line20
posSlope line24
rightOf line24 line209
sameY line24 line209

equalArea line24 line209
far line24 line209
overlapRightOf line24 line2
parallel line24 line209
posSlope line209
posSlope line24
sameY line24 line209

bisects_cc line24 line209
equalArea line24 line209
negSlope line24
perpendicular line24 line20
posSlope line209
sameX line24 line209
sameY line24 line209

Figure 13-11: The fifth set of labeled near-miss examples.

1. connected_12 line2 line1
2. connected_12 line3 line2
3. connected_11 line5 line1
4. connected_21 line4 line1
5. connected_12 line4 line3
6. connected_22 line6 line3
7. connected_21 line6 line4
8. connected_12 line5 line4

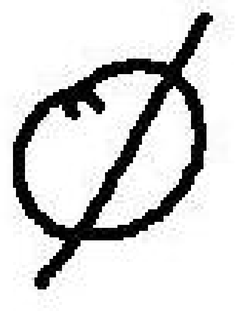


Figure 13-12: A no smoking symbol which was quickly and correctly generated.

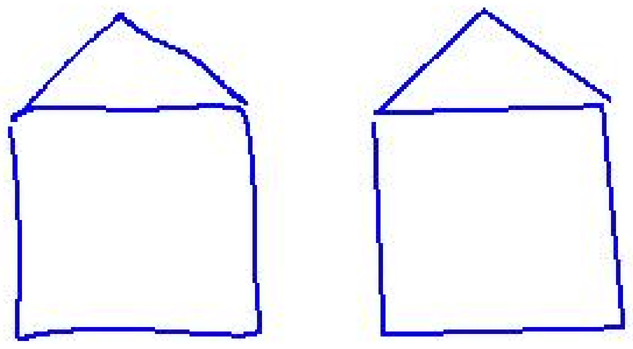


Figure 13-13: The drawn and cleaned up image of a house for which a description was automatically quickly generated for, but for which generating near miss examples for took an impractical amount of time.

9. connected_12 line6 line5
10. vertical line1
11. vertical line3
12. horizontal line2
13. horizontal line4
14. posSlope line5
15. negSlope line6
16. equalArea line2 line1
17. equalArea line3 line2
18. equalArea line3 line1
19. equalArea line4 line1
20. equalArea line4 line2
21. equalArea line4 line3
22. equalArea line6 line5
23. acuteMeet line5 line4
24. acuteMeet line6 line4
25. obtuseMeet line6 line3
26. obtuseMeet line6 line5
27. obtuseMeet line5 line1
28. smaller line5 line1
29. smaller line5 line2
30. smaller line6 line3
31. smaller line5 line3
32. smaller line5 line4

33. smaller line6 line2
34. smaller line6 line1
35. smaller line6 line4
36. above line6 line1
37. above line6 line2
38. above line4 line2
39. above line5 line2
40. leftOf line5 line3
41. rightOf line3 line1
42. rightOf line6 line1
43. sameY line3 line1
44. sameY line6 line5
45. overlapLeftOf line4 line3
46. overlapLeftOf line5 line2
47. overlapLeftOf line4 line2
48. overlapLeftOf line5 line4
49. overlapLeftOf line6 line3
50. overlapRightOf line2 line1
51. overlapRightOf line3 line2
52. overlapRightOf line4 line1
53. overlapRightOf line5 line1
54. overlapRightOf line6 line4
55. overlapRightOf line6 line5
56. overlapRightOf line6 line2
57. overlapAbove line6 line4
58. overlapAbove line4 line1
59. overlapAbove line3 line2
60. overlapAbove line4 line3
61. overlapAbove line5 line1
62. overlapAbove line5 line3
63. overlapAbove line6 line3
64. overlapAbove line5 line4
65. overlapBelow line2 line1
66. near line3 line1
67. near line4 line2
68. near line5 line2
69. near line5 line3
70. near line6 line1
71. near line6 line2

72. perpendicular line2 line1
73. perpendicular line4 line1
74. perpendicular line3 line2
75. perpendicular line4 line3
76. parallel line3 line1
77. parallel line4 line2
78. slanted line6 line2
79. slanted line5 line3
80. slanted line5 line2
81. slanted line6 line1

The list of perceptually important constraints is shown below. The list is still long, as six lines permit as many as 36 constraints to be chosen. From this list, we see the importance of creating hierarchical (and thus shorter) descriptions.

1. connected_12 line2 line1
2. connected_12 line3 line2
3. connected_11 line5 line1
4. connected_21 line4 line1
5. connected_12 line4 line3
6. connected_22 line6 line3
7. connected_21 line6 line4
8. connected_12 line5 line4
9. connected_12 line6 line5
10. vertical line1
11. vertical line3
12. horizontal line2
13. horizontal line4
14. posSlope line5
15. negSlope line6
16. equalArea line2 line1
17. equalArea line3 line1
18. equalArea line4 line1
19. equalArea line6 line5
20. acuteMeet line5 line4
21. acuteMeet line6 line4
22. obtuseMeet line6 line3
23. obtuseMeet line6 line5

24. obtuseMeet line5 line1
25. above line6 line1
26. above line6 line2
27. above line4 line2
28. above line5 line2
29. leftOf line5 line3
30. rightOf line3 line1
31. rightOf line6 line1
32. sameY line3 line1
33. sameY line6 line5
34. smaller line5 line1

13.3.3 Lesson 3: Shape Generation Can Take a Long Time

After a shape is drawn, the system presents the same shape to the user for verification. This is called shape perfecting or creating an idealized version of this shape. Shape perfecting is quite fast because 1) a solution exists and 2) the solution is near to the starting conditions. However, when trying to generate a shape which may not exist, or for which the starting condition is far from the solution, the system can take a long time. For example, shapes with more than three lines can take so long enough to compute, that is becomes impractical.

While the system would sometimes be quite fast, if a solution exists (a few seconds), a failure, or worse, several failures in a row, would undeniably frustrate the user. Because shape constraints are often interrelated, it is difficult to predict whether a shape will fail or not.

13.3.4 Lesson 4: Failure Does Not Mean the Shape Cannot Exist

When learning a shape description, it is tempting to presume that if a shape is not generated, then it cannot exist. However, shape generation may also fail simply because the starting conditions are bad. Thus, we reprogrammed the generator to try several different random starting conditions. However, this did not guarantee success, and greatly added to the wait time.

The uncertainty of the viability of a proposed list of constraints created other difficulties. It made it difficult to determine whether there were more similar shapes that should be generated, or if there were no more shapes to be generated and the near-miss generation should stop. Because of this, the system generated more near-miss examples than it should have. The learning algorithm would have benefited from the knowledge that a particular constraint could not create a shape.

13.3.5 Lesson 5: The System May Generate Perceptually Confusing Examples

Sometimes the system would return an example shape, but return an error greater than zero. Because this happened quite often, we wanted to use these generated shapes. However, sometimes these examples were good, and sometimes they were perceptually confusing. For example, in Figure 13-14, the two non-touching (the FAR constraint) parallel lines are asked to be of the same size (i.e., length), while still asking them to be horizontally and vertically aligned. This is physically impossible. But the system generates the example as best as it can and shows the example to the user. Perceptually, lines of similar lengths may appear to be of equal length to each other if they were not parallel and horizontally and vertically aligned. However, when the two lines are parallel and next to each other, the difference in length is perceptually obvious. The problem lies in that the system thinks it has created an example where the lines are *close enough* to the same length, but, in fact, it has not. This could perhaps be alleviated by more sophisticated thresholds which take into account the locality of other shapes during shape generation.

The system can also generate perceptually confusing negative examples. For in-

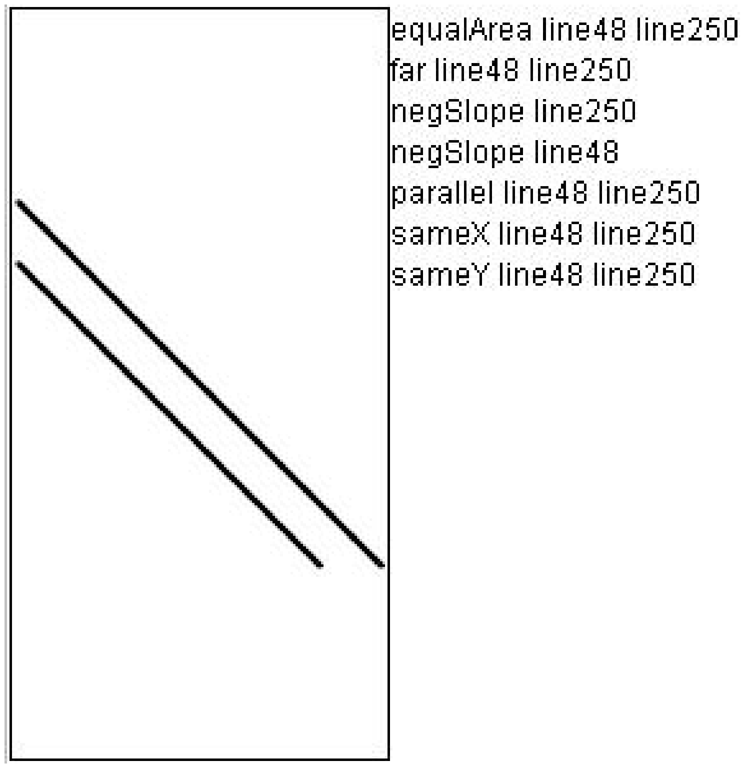


Figure 13-14: A perceptually confusing positive example.

stance, in Figure 13-15, the lines seem to be close to parallel. This example could be fixed by requiring negative-sloped lines to be closer to -45 degrees.

Displaying the description next to the generated shape solved some of the perceptual ambiguity, but we feel that this is a less than ideal solution.

13.3.6 Lesson 6: Positive Examples Were More Helpful than Negative Examples

When the system sees a positive example, it knows that all of the constraints in the example are allowed in a positive example. When the system sees a negative example, it knows only that at least one of the constraints in the examples caused the negative example. When the system does not know which example caused the negative example, it has to branch the concept space. Because constraints are inter-related, it is very common for the negative causing constraint to be unknown. Branching is costly,

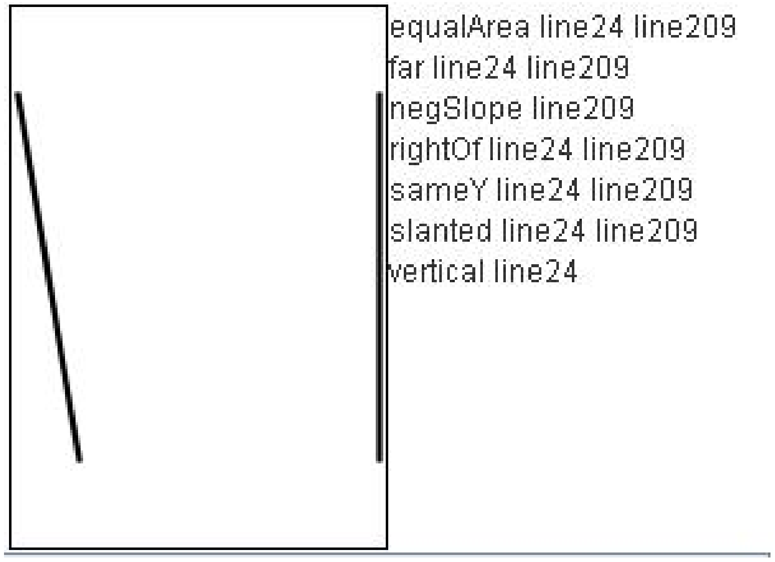

```
equalArea line24 line209
far line24 line209
negSlope line209
rightOf line24 line209
sameY line24 line209
slanted line24 line209
vertical line24
```

Figure 13-15: A perceptually confusing negative example .

and when there are many negative examples (which is common), the branching can slow down the system noticeably.

Because of the high costs of branching, the system first processes the positive examples, and, holds the negative examples which have more than one possible cause for later processing.

13.3.7 Lesson 7: Show Rotations First

Showing rotations first (which was not done in the example shown above, but is now part of the current system) has two benefits. First, many shapes can be rotated. By showing a rotated example first, if the shape is rotatable, the system will have a positive example, with many changed constraints, causing the system to learn more quickly. The other advantage of rotation is that it can be done without automatically generating the shape through Matlab, and the generated shape is guaranteed to return quickly. Thus, rotated examples can be provided quickly even for extremely large shapes.

Because of the added benefit of rotations, this researcher also had the system create other changes that could be computed without Matlab, such as scaling, single line rotations, and moving of line endpoints by shortening and elongating lines. While scaling helped moderately, the other transformations helped minimally, and, if shown first, would often only hinder the learning speed by showing unhelpful negative examples.

By showing rotations first, the system can automatically choose whether to use *orientation-dependent* or *orientation-independent* constraints, which greatly reduces the number of constraints to test. The problem with this idea, though, is that part of the shape may be rotatable, and part of the shape may not be. One can either hope that this is generally not the case, or have the system produce examples that rotate only a subset of the shape.

13.4 Overall Qualitative Results

Despite its failings, users tended to like the near-miss generator. The slowness was the main reason that people chose to stop after either the first or second generated description, and manually tweak the descriptions themselves. When asked, they did say that, if the system did not take so long for complex shapes, they would have chosen to have the system automatically construct the definition from the near-miss examples, as they found it much easier to label a shape than to examine the list of constraints to find any problems. They also thought that they probably missed certain examples and created imperfect descriptions when hand-tweaking. However, since the goal of this research is to allow users to create shape recognizers that matched their internal perceptions of a shape, it is difficult to find a definitive way to test if their final descriptions were correct.

Chapter 14

Implications for Future Research

This research has several implications for future investigations. Some of the specific implications were highlighted at the end of each chapter of this thesis. Broader issues, and those that are of a general nature, are discussed in this chapter.

14.1 Empowering Instructors to Build Sketch Systems for Immediate Feedback in the Classrooms

14.1.1 Motivation

LADDER and *GUILD* were developed to simplify development sketch interfaces to allow development by non-experts in sketch recognition. The number on application in mind for this has been integration of sketch recognition system in the classroom to improve pedagogy.

Imagine the following scenario:

Scenario: The class is Computability; the instructor is teaching finite state machines (FSMs) today. Writing on a SmartBoard behind her, she explains how an FSM works by drawing one into a sketch recognition system that she built before class in less than a half-hour. Once she has completed drawing the FSM, she adds an input string and presses "run." The students watch and learn, as each state briefly turns red as the input string passes through it. After observing various different demonstrations, the students are then asked to build an FSM of their own which satisfies a certain class of input strings. Students attempt to build the correct FSM on their own tablet PCs; they test their FSMs with several input examples, using the FSM sketch application built by the instructor. Once each student is satisfied with his or her solution, he or she submits it to the teacher. Each submitted solution is automatically classified as either correct or incorrect, and the further classified by the error that it contains. The groups are displayed to the instructor on her personal tablet PC on the podium. The teacher then selects one submission that contains a common error and displays it on the SmartBoard. She handwrites an input string that is not recognized, but should be, and the students watch the interactive display to understand where the error lies. The students leave the classroom with a thorough understanding of FSM, and the instructor receives feedback on the level of student understanding on FSMs.

This scenario emphasizes the profound effect that *LADDER* and *GUILD* could have on pedagogy and classroom learning. Sketch recognition systems can be used to:

1. explain drawn graphical content in an interactive way.
2. provide students classroom graphical practice on classroom-provided student tablets.
3. provide students at home practice.
4. automatically correct graphical homework.

5. automatically correct in-class assignments for immediate feedback.

Graphical diagrams are an important part of the teaching process. Whiteboard sketches by the instructor are used in the communication of ideas [70] [78] [79] [114] [207]. Animations are an effective way to explain material, but they currently have to be canned animations, rather than animations pertaining to real-time drawings. CAD (computer automated design) systems enable interactive graphical design, but teachers may not integrate these CAD systems into their classroom because 1) the CAD systems don't provide sketching capabilities, and their interface may be unnatural and require intricate learning, 2) teaching CAD system use may take time away from other classroom material, 3) the CAD system may be prohibitively expensive, or 4) the CAD system may not provide the functionality necessary to teach the material and is not easily modified to suit the teaching needs.

This document has already emphasized the benefits of sketch systems, allowing both drawing freedom and computer design advice and simulation capabilities. Sketches are used throughout the educational process [13]; here are a few examples: finite state machines, mechanical engineering diagrams, electrical circuit diagrams, physics diagrams, chemical symbols and reactions, flow charts, UML diagrams to design software, musical notation, tree data structures, graphs, and many others. As noted earlier in this thesis, from 2000-2002, this researcher built a UML class diagram sketch recognition system which was used in two sections of a game programming course at Columbia University to teach 60 high school students software engineering techniques [99]. The system was well received, and it helped students to comprehend difficult computing concepts without having to also teach them Rational Rose$^{\text{TM}}$. Although sketch recognition applications have been built for a variety of domains, it is impractical to build a system that will suit the needs of all instructors as there is an overabundance of academic content expressed graphically. The needs of educators are usually class, or even assignment specific, and until now, sketch recognition systems have remained inaccessible to educators who need systems specific to their classroom learning goals.

Graphical diagrams are important in many subjects, and used throughout the learning process, but correcting these diagrams proves difficult and time-consuming. Oftentimes, these graphical assignments or tests are omitted for these reasons. This is unfortunate as pedagogical studies suggest that not only does testing aid in learning, but it is more effective than testing alone [184]. Roediger explains that students remember more of what they learned when alternating only two study sessions with two testing sessions rather than by having four study sessions. Roediger also describes how early feedback after testing increases the percent of the material learned.

14.1.2 Determining Pedagogical Usefulness

Part of this future work also includes measuring this technology's effectiveness and impact on pedagogy through a case study of classroom use. The interactions of the students and the teacher will be watched, recorded, and interviewed to determine their reactions to the technology and their perceptions of the application's effectiveness in 1) aiding teacher explanations, 2) aiding student self-understanding, 3) simplifying homework correction, 4) providing student feedback, and 5) simplifying the monitoring of student understanding.

This researcher has high hopes for integrating these technologies and has already given several lectures on it, has had several instructors ask to be part of the study for use in their classroom, and has been asked to give a guest lectures at a neighboring university.

It is the hope of this researcher that work will benefit pedagogy as a whole, aiding the teaching and learning process through quicker and more effective feedback, with the sketch recognition systems ultimately built from this project being used to teach students at all levels (from kindergarten to graduate students). This researcher also hopes that this future work will bridge a gap between education and computer science, which will increase the interest of women (an under-represented group) in computer science [106].

14.1.3 Improving the Usability of *LADDER* and *GUILD*

Future work includes integrating *LADDER* and *GUILD* into the classroom, thus creating a more robust system through interviews and monitoring reactions and usage [9] [34] [124]. We hope that with the availability of tablet PCs in the classroom, this will revolutionize classroom teaching by providing understanding, animation, correction, and immediate feedback of hand-drawn graphical input. We expect that broad use will make clear how sketch user interfaces can be more simply and easily defined. This includes broadening what is expressible and examining how other forms of context can be used to improve recognition, and determining how editing, display, animations, and functionality can be more intuitively described.

Other questions to be looked at by the researcher include: What is the most effective way to specify connections to existing back-end systems? How can additional context (as described below) be specified in a simple, but yet, effective way?

14.2 Multi-modal *LADDER* and *GUILD* Development

Future work includes integrating several modes of interaction into both the development process (discussed here) and for use in recognition (discussed below). We would like to study what is the most effective way of inputting domain information; thus far domain information information is entered using only text and sketching, but other ways of describing domains include speech and hand gestures [23]. We expect that hand gestures and speech may be an effective way of inputting editing and animation information.

14.3 Recognizing a Broader Class of Shapes by Combining User-Dependent and User-Independent Methods

To provide more robust recognition, this researcher proposes to integrate user-independent recognition methods (as described in this document) and user-dependent recognition methods (such as the feature-based methods used by Rubine [185]).

We expect that the combination of these two results will provide both faster and more accurate sketch recognition, essentially capturing the benefits of the two methods. The advantage of the user-dependent feature-based recognition technique is that it is fast and robust in terms of messy drawing. The disadvantage of this technique is that a shape will not be recognized if the shape that is to be recognized is drawn with a different number or order of strokes than in the training example.

The advantage of the user-independent geometric recognition technique described in this paper is that shape can be drawn using any style as shapes are recognized by what they look like, rather than how they were drawn. The disadvantage of this technique is that shapes drawn quickly that do not look like the intended shape may be mis-recognized because the low-level stroke parse was unable to parse the stroke into the appropriate primitives.

Figure 14-1 shows the power of combining these techniques. The rectangle at the top of the figure is the initial hand-drawn example. The middle row shows rectangles that will be recognized by the geometric-based recognition algorithm, but not the feature-based recognition algorithm. The bottom row shows rectangles that will be recognized by the feature-based recognition algorithm, but not the geometric-based recognition algorithm. By combining these recognition techniques, the recognition system will recognize a larger class of shapes, while still providing users with the flexibility to draw however they like and quickly recognizing shapes.

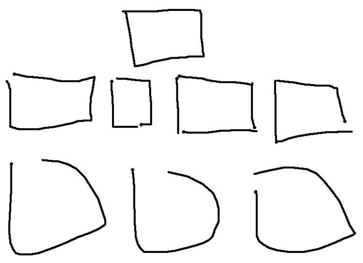


Figure 14-1: Rectangles. Row 1: The original training example. Row 2: Rectangles recognized from the training example, using geometric-based recognition. Row 3: Rectangles recognized from the training example using feature-based recognition.

On a similar note, there exist a number of user-independent vision techniques for recognizing shapes in a pixelized format [27] [18] [29] [126] [132]. We feel that integrating these techniques into the system will cause it to recognize a larger number of shapes more accurately.

14.4 Using Context Reducing Ambiguity

One important difficulty in sketch recognition is that we do not necessarily want to recognize what the user drew, but, rather, what the user intended to draw. Humans can make use of the plethora of available local and global contextual information surrounding the sketch, which provides additional clues about the sketchers intention, and can, thus, recognize many shapes that computers cannot. Sketch systems currently deal with ambiguity by constraining the drawing style [150] or by waiting for further geometrical information to disambiguate the shapes using only a minimal amount of local geometric context [102] [104] [12] [99] [157] [210].

However, humans use a myriad of forms of context when recognizing shapes, not just local geometrical context. The shape description user study described in Section 4.2.2 shows subjects utilizing a combination of geometric, similarity, cultural, and common sense contextual cues to describe objects. This study and related research [176] revealed the importance of geometrical context, but it also showed the importance of other forms of context, including the value of describing things in terms of previously defined objects, using similarity and in terms of cultural objects. This was particularly pertinent because users were specifically instructed not to include such terms in their descriptions, but to use only geometrical terms, which causes us to wonder how much more prevalent they would be if they were permitted.

Future work to be done by this researcher includes the ability to improve sketch recognition using 1) local perceptual context, such as the geometrical profiles of the shapes drawn, human perception tolerance, similarity to other shapes previously drawn, surrounding speech, and surrounding hand gestures, and 2) global context, including functional context [183] and common sense context.

14.4.1 Developing Robust Geometric Recognizers through Perception

In order to correctly interpret shapes at a higher level, the system needs to ensure that 1) our lower level stroke processors generate all of the possible shape interpretations, as well as a probabilistic measure of certainty, and 2) the geometric shape builders generate all of the possible true geometric constraints and a probabilistic measure of their certainties. This document has previously shown the need for and the usefulness of using perceptually based constraints for recognition, but there is a need to do an exhaustive study on the effect of perception of hand-drawn shapes, using various levels of context. To provide further understanding, this researcher outlines a selection of the questions she will study for the PARALLEL constraint: What is the threshold for two lines to be parallel? That is, how close to parallel do two lines have to be for them to be parallel? Are these thresholds different if they are shown perfectly clean lines versus messy hand-drawn lines? Do these thresholds change if the lines are close together, or if there are other shapes between these lines? Do the thresholds change if the lines are horizontal or vertical, compared to lines of other angles? How do the thresholds differ if subjects are asked only if the two lines are parallel versus if they are asked to give the angle between the lines? Is there a different threshold when users are asked to draw parallel lines compared to when they are shown two parallel lines?

The following studies will be performed for each of the shapes and constraints in the language. For shapes:

1. The users will be asked to draw shapes of a particular type (e.g., circle).
2. The users will be asked to identify specific values relative to the tested shape given randomly produced shapes (e.g., they may be asked to determine the ratio between the width and the height of a circle).
3. The users will be shown hand-drawn shapes alone on a page, and they will be asked to click a check box, identifying whether the sketcher intended to draw the tested shape.
4. The users will be shown hand-drawn shapes in local context (i.e., with the other shapes on the page), and they will be asked to click a check box, identifying whether each is the tested shape.
5. The users will be shown a video of the user drawing the shape, and they will be asked to click a check box, identifying whether the user intended to draw the tested shape.

For constraints:

1. The users will be asked to draw shapes abiding by a particular constraint. (For example, the users may be asked to draw parallel lines.)
2. The users will be asked to identify specific values in reference to the constraint given randomly produced shapes. (For example, the users may be asked to determine the angle between two lines.)

3. The users will be shown hand-drawn shapes alone on a page, and they will be asked to click a check box, identifying whether the sketcher intended to abide to the tested constraint (e.g., whether the two lines are parallel).

4. The users will be shown hand-drawn shapes in local context (i.e., with the other shapes on the page), and they will be asked to click a check box, identifying whether the tested constraint is held.

5. The users will be shown a video of the user drawing the shape, and they will be asked to click a check box, identifying whether the user intended for the constraint to be held.

This researcher plans to analyze the data to produce thresholds and a certainty measure for each of the shapes and constraints. When appropriate, she will produce different thresholds for different contextual situations.

The information gained above will be integrated into the system's low level and high-level recognizer. For any given hand-drawn stroke, the low level recognizer will produce all possible interpretations, along with a certainty measure. For any higher-level shape to be formed geometrically, it will produce all possible true constraints and a measure of certainty.

14.5 Multi-modal Context

Imagine a mechanical engineering instructor giving a lecture with graphical content at a white board; she is not just sketching, she is speaking, sketching, and actively moving her hands to convey and communicate the material and interactions between the elements. Each of the modes play an important part of the educational process [88] [137]. Students use all of the modes to effectively understand the diagrams and the topic. Integrating speech and gesture into the shape concept, and, thus, the recognition process, could provide contextual information to improve recognition.

Adler and Eisenstein at MIT and others have developed algorithms to aid in the understanding of speech or gesture combined with sketching [2] [3] [4] [64] [65] [63] [66] [68] [67]. Cohen shows the benefit of multi-modal systems [50]. Several multi-modal systems integrating sketch and speech have been built [35] [49] [127] [175].

This researcher plans to extend *LADDER* and *GUILD* to allow shape descriptions to include accompanying speech and hand gestures which may help disambiguate shapes.

14.5.1 Functional Context

Functional context can be used to clarify an otherwise ambiguous diagram. Humans use functional context to select a more plausible interpretation. For example, an otherwise ambiguous electrical engineering drawing may have one interpretation which causes voltage to run through the circuit, while the other does not. In this case, the clear choice should be to choose the interpretation that causes something to happen. However, this is not a simple task for the computer. In order for the computer to be able to make such an interpretation, it must know not only how to simulate the drawing, but also which simulation results are preferred to others. The example given may seem simple, but think of a circuit diagram with a myriad of possibilities for voltage flow. Or, think of a mechanical engineering diagram in which certain items must be of identical size in order to achieve the desired performance. As a first step in this problem, this researcher plans to identify several scenarios in which this may occur, as well as data for the appropriate solution. She shall then look at different ways of encoding the information in order to keep things simple for the developer. This researcher hopes to find several ways of abstracting functional domain context so that the ideas may be useable in different domains.

14.6 Describing Shapes Through Similarity

Previous studies discussed in this document have shown that humans are particularly sensitive to similarities in shape. In these studies, users described new shapes in terms of previously defined shapes, even when they were explicitly told not to. *GUILD*'s current recognizers allows users to build new shapes hierarchically, thus providing a limited amount of similarity, enabling users to build recognizers for shapes described as: "this shape is the same as the last, except that it also has...." However, based on a study described in this thesis, it seems that humans use similarity in a myriad of ways, simplifying descriptions, and also enabling users to describe and recognize shapes that would otherwise be impossible or at least impractical to distinguish. This researcher plans to develop a model for recognition based on fuzzy logic to recognize similar shapes. By allowing users to describe similar shapes recognition systems can recognize a larger class of shapes. For instance, using the new fuzzy logic model, a system can recognize the rounded rectangle as a shape that is similar to both a rectangle and a circle.

14.7 Using Common Sense to Simplify Shape Description

The shape description study described in this thesis revealed a plethora of cultural artifacts in the user's descriptions despite explicit instructions not to include any non-geometrical objects. Different users tended to use the same cultural artifact to describe the same shape, emphasizing the existence of a shared common sense library of cultural artifacts related to shape and function. For example, the McDonalds' Arch, a cultural artifact known for its shape, was commonly used to describe a side-ways curved capital-E; a bridge, a cultural artifact known for its function (a bridge connects two larger areas with a path, but bridges themselves can vary in appearance), was commonly used in the study to describe the shape"][" in our study. In order to allow developers to describe shapes in terms of everyday cultural objects, one would have to 1) define each of the objects that may be used in a description and 2) come up with a similarity metric for comparing them. Given the number of objects in our everyday lives, this is no inconsequential task. However, this researcher hopes and expects that users will use only a limited set of cultural shapes to describe these shapes, and that the set of shapes will be common among groups of people. This researcher expects that integrating common sense artifacts into the sketch recognition system will 1) ease human descriptions of shapes and 2) help systems to better recognize drawn images.

Chapter 15

Conclusion

The over-arching goal of this work is to make human-computer interaction as natural as human-human interaction. Part of this vision is to have computers understand a variety of forms of interaction that are commonly used between people, such as sketching. Computers should, for instance, be able to understand the information encoded in diagrams drawn by and for scientists and engineers, including mechanical engineering diagrams.

Sketches are used throughout the design, brainstorming, and educational process; we name here a few examples: finite state machines, mechanical engineering diagrams, electrical circuit diagrams, physics diagrams, chemical symbols and reactions, flow charts, UML diagrams to design software, musical notation, tree data structures, graphs, and many others.

Ordinary paper offers one the freedom to sketch naturally, but it does not provide the benefits of a computer-interpreted diagram, such as more powerful editing and design advice or simulation abilities. Sketch recognition systems bridge that gap by allowing users to hand-sketch their diagrams, while recognizing and interpreting these diagrams to provide the power of a computer-understood diagram.

Many sketch systems have been built for a myriad of domains. Unfortunately, these sketch systems may not fill the needs of the sketcher, and building these sketch systems requires not only a great deal of time and effort, but also an expertise in sketch recognition at a signal level. Thus, the barrier to building a sketch system is high. This researcher wants to empower user interface developers, including designers and educators, who are not experts in sketch recognition, to be able to build sketch recognition user interfaces for use in designing, brainstorming, and teaching. In response to this need, this researcher has developed the *FLUID* framework for *f*aci*l*itating *UI* *d*evelopment.

As part of the framework, this researcher has developed a perception-based sketching language, *LADDER*, for describing shapes, and a customizable recognition system, *GUILD*, that automatically generates a sketch recognition system from these shapes. In order to allow drawing freedom, shapes are recognized by what they look like, rather than by how they are drawn.

LADDER provides the ability to describe how shapes in a domain are drawn, displayed, and edited within the user interface. Because humans are naturally skilled at recognizing shapes, the system uses human perceptual rules as a guide for the constraints in the language and for recognition. These perceptual rules were reinforced through a user study of 35 people who were asked to describe approximately 30 shapes each. The language also has a number of higher-level features that simplify the task of creating a domain description, including hierarchy, abstract shapes, vectors, and context.

GUILD transforms a *LADDER* description into a user interface. Because of the importance of drawing freedom, this researcher developed a new, fast recognition algorithm based on indexing. This algorithm takes advantage of the perceptually-based constraints in *LADDER* to allow shapes to be drawn in an interspersed manner, but still to be recognized in real-time.

GUILD also provides an API to allow users to connect to existing back-end systems. Thus far, about a dozen people have used the system to build domain systems for over fifteen different domains.

Because showing the ideal shape is an important part of beautification, this researcher also built a shape generator that uses MATLAB to create an idealized version of a shape with all of the constraints solved. This system is also used to generate near-miss shapes.

As it is difficult to create a correct shape description, the research built a debugging system to correct syntactical and conceptual errors. Because it is more natural to draw a shape than to describe it, this researcher developed a system to automatically generate a description from a single drawn example, based on work by others. The generated shape description can then either be hand-tweaked or modified automatically with a concept learning algorithm developed by this researcher, using labeled near-miss examples. These near-miss examples can either be generated by the user or, because users are often unreliable at generating their own near-miss examples, they can be generated automatically by the system. This researcher had several users try the near-miss generation system, and then asked them comment on the successes and failures of the systems.

This researcher feels that the future implications of this research are large. She mentioned several examples, throughout the work. But she is most interested in seeing her work applied to classroom pedagogy. Graphical diagrams play an important part in the learning process, but they are time-consuming to grade and are often omitted from the homework and testing process. Testing and early feedback has been shown to be critical in the learning process. This researcher feels that the work in this document can be applied to in-class and out-of-class learning, enabling teachers to create their own recognition systems for class and homework use, and by 1) aiding teacher explanations through interactive animations, 2) aiding student self-understanding through student-directed interactive animations, 3) allowing for automatic graphical homework correction, 4) permitting immediate student feedback, and 5) simplifying the monitoring of student understanding.

This researcher would also like to improve sketch recognition accuracy by 1) combining user-dependent (feature-based) recognition with user-independent (geometric) recognition techniques, and by2) incorporating global and local context into the recognition system, including geometric, perceptual, functional, multi-modal, similarity, and common sense context.

Appendix A

LADDER Elements

A.1 Predefined shapes

- Shape

  The abstract shape that all shapes extend. The accessible properties for all shapes are:

  – Rectangle boundingBox: the smallest rectangle parallel to the horizon that can be placed around the shape

  – Point center: the center point of the boundingBox

  – double width: the width of the boundingBox

  – double height: the height of the boundingBox

  – double area: the area of the boundingBox

  – long time: the time at which the shape was completed. Time is included to allow constraints to specify stroke order or direction.

- Point

  A point. The accessible properties are:

  – double x: the x value of the point

  - double y: the y value of the point

- Path

  A continuous stroke, not necessarily straight. The accessible properties are:

  - Point p1: one endpoint of the stroke
  - Point p2: the other endpoint of the stroke
  - double length: the length of the stroke

- Line

  A straight line. The accessible properties are:

  - Point p1: one endpoint of the line
  - Point p2: the other endpoint of the line
  - Point midpoint: alias for center
  - double length: length of the line
  - double angle: angle of the line. The angle is between 0 and 360 degrees, and is the angle between a directional horizontal line pointing to the right and the Line directed from p1 to p2.

  All Lines are also Paths.

- Curve

  A curve defined by four points: its two endpoints and two control points. The accessible properties are:

  - Point p1: one endpoint of the curve
  - Point p2: the other endpoint of the curve
  - Point control1: one control point
  - Point control2: the other control point of the curve

- Spiral

  A spiral starting from one angle and radius, ending at another. The radius continually gets larger or smaller throughout the spiral, with the amount specified by the growFactor. The accessible properties are:

  - Point p1: one endpoint of the spiral
  - Point p2: the other endpoint of the spiral
  - Point center: center of the spiral, a.k.a. the rotation point
  - double radius1: the radius of the spiral at p1 (i.e., the distance from center to p1)
  - double radius2: the radius of the spiral at p2 (i.e., the distance from center to p2)
  - double angle1: the angle of the line from center to p1
  - double angle2: the angle of the line from center to p2
  - double degrees: the total number of degrees covered (can be larger than 360)
  - double numLoops: the number of complete rotations around the center point (equal to degrees/360)
  - double growFactor: the proportional change in the radius of the circle at each rotation

- Arc

  An arc, a portion of an ellipse

  - Point p1: one endpoint of the arc
  - Point p2: the other endpoint of the arc
  - Point center: center of the arc, as if the arc were a complete ellipse; a.k.a. the rotation point
  - double width: the width of the ellipse implied by the arc

    - double height: the height of the ellipse implied by the arc
    - double angle1: the angle of the line from center to p1
    - double angle2: the angle of the line from center to p2

- Ellipse

  An ellipse in any orientation. This is an ellipse defined by the four points of a rectangle surrounding it. The accessible properties are:

    - Point center: center of the ellipse
    - double width: the width of the ellipse
    - double height: the height of the ellipse

- Text

  Text is entered by using the keyboard or the handwriting input device provided by the tablet pc. Its accessible properties are:

    - Point center: center of the string
    - String text: the text of the string

A.2 Predefined Constraints

Note that any argument with a name *line*x can take an argument of any shape type that contains a Point p1 and Point p2, such as a curve or an arc. A line between these two points will then be used in computation.

Constraints with an underscore in them are more specific versions of other constraints. While it is permissable to use them in hand-written constraints, these are included for use by the near-miss generator to effectively reduce the possibility space.

Note that the constraints define what the shapes should look like in the ideal sense, or what the sketcher intended. Each of the constraints has a noise tolerance included in it, discussed previously in this document.

- ABOVE *shape1 shape2*

  The center of *shape1* is above the center of *shape2*.

  Orientation-Dependent

- ACUTE *line1 line2*

  The angle from *line1* to *line2*, when traveling in a counter-clockwise direction, is acute. The maximum computable angle between undirected lines is 180 degrees, as *line1* and *line2* are undirected. (Deprecated because it is hard to use. Use SLANTED instead.)

  Orientation-Independent

- ACUTEDIR *line1 line2*

  line1 and *line2* are directed from POINT P1 to POINT P2. The maximum computable angle between directed lines is 360 degrees. The angle between the two lines is acute, when measuring in the counter-clockwise direction. (Deprecated because it is hard to use. Use SLANTED instead.)

  Orientation-Independent

- ACUTEMEET *line1 line2*

  line1 and *line2* meet at one of their endpoints. The lines are directional lines pointing away from their meeting point. These lines form an acute angle when measuring in a counter-clockwise direction. The specification of P1 and P2 have nothing to do with the direction of the lines when measuring the angle.

  Orientation-Independent

- BELOW *shape1 shape2*

  The center of *shape1* is below the center of *shape2*.

  Orientation-Dependent

- BISECTS*point line*

  The *point* is located at the center (bisects) of the *line*.

  Orientation-Independent

- BISECTS_1C *line1 line2*

  Endpoint P1 of *line1* bisects *line2* (is located at the center of the *line2*). This is a more specific constraint than BISECTS.

Orientation-Independent

- BISECTS_2C *line1 line2*

  Endpoint P2 of *line1* bisects *line2* (is located at the center of the *line2*). This is a more specific constraint than BISECTS.

  Orientation-Independent

- BISECTS_C1 *line1 line2*

  Endpoint P1 of *line2* bisects *line1* (is located at the center of the *line1*). This is a more specific constraint than BISECTS.

  Orientation-Independent

- BISECTS_C2 *line1 line2*

  Endpoint P2 of *line2* bisects *line1* (is located at the center of the *line1*). This is a more specific constraint than BISECTS.

  Orientation-Independent

- COINCIDENT *point1 point2*

  point1 and *point2* are located at the same location.

  Orientation-Independent

- COLLINEAR *point1 point2 point3*

  point1, *point2*, and *point3* are located on one line.

  Orientation-Independent

- CONCENTRIC *shape1 shape2*

  The center of *shape1* and the center of *shape2* are coincident.

  Orientation-Independent

- CONNECTED *shape1 shape2*

  shape1 and *shape2* have P1, P2, or PORTx defined as endpoints. (Ports are used to allow complex shapes to have more than one endpoint. This is used in domains such as electrical engineering.) One endpoint from *shape1* and one endpoint from *shape2* are coincident.

  Orientation-Independent

- CONNECTED_11 *line1 line2*

  Endpoint P1 of *line1* is coincident to endpoint P1 of *line2*. This is a more specific constraint than CONNECTED.

  Orientation-Independent

- CONNECTED_12 *line1 line2*

  Endpoint P1 of *line1* is coincident to endpoint P2 of *line2*. This is a more specific constraint than CONNECTED.

  Orientation-Independent

- CONNECTED_21 *line1 line2*

  Endpoint P2 of *line1* is coincident to endpoint P1 of *line2*. This is a more specific constraint than CONNECTED.

  Orientation-Independent

- CONNECTED_22 *line1 line2*

  Endpoint P2 of *line1* is coincident to endpoint P2 of *line2*. This is a more specific constraint than CONNECTED.

  Orientation-Independent

- CONTAINS *shape1 shape2*

  The BOUNDINGBOX of *shape1* contains the BOUNDINGBOX of *shape2*.

  Orientation-Independent

- DIAGONAL *line1 line2*

  The line has a positive or negative slope.

  Orientation-Dependent

- DRAWORDER *shape1 shape2*

  shape1 was drawn before *shape2*.

  Orientation-Dependent

- EQUAL *value1 value2*

  value1 is equal to *value2*. Deprecated. This is not suggested for use, use a more specific constraint like EQUALSIZE.

  Orientation-Independent

- equalAngle *line1 line2 line3 line4*

  The angle between *line1* and *line2* is equal to the angle between *line3* and *line4*.

  Orientation-Independent

- equalLength *line1 line2*

  line1 and *line2* are of equal length.

  Orientation-Independent

- equalSize *shape1 shape2*

  shape1 and *shape2* are of equal size. The size of the object is measured by the length of the boundingBox. This allows lines and other objects to be compared appropriately.

  Orientation-Independent

- far *shape1 shape2*

  shape1 and *shape2* are far from each other.

  Orientation-Independent

- horizontal *line*

  The *line* is horizontal. (The slope is zero.).

  Orientation-Dependent

- intersects *shape1 shape2*

  If *shape1* and *shape2* are lines, then they intersect; else, their boundingBoxes overlap.

  Orientation-Independent

- larger *shape1 shape2*

  The diagonal of the boundingBox of *shape1* is longer than the diagonal of the boundingBox of *shape2*

  Orientation-Independent

- leftOf *shape1 shape2*

  All of *shape1* is to the left of all of *shape2*. The x values do not overlap.

  Orientation-Dependent

- LONGER *line1 line2*

  The length of *line1* is longer than the length of *line2*. This is syntactic sugar for LARGER

  Orientation-Independent

- LONGER *shape1 shape2*

  The length of *line1* is longer than the length of *line2*. This is syntactic sugar for LARGER

  Orientation-Independent

- NEAR *shape1 shape2*

  shape1 is near to *shape2*.

  Orientation-Independent

- NEGSLOPE *line*

  The *line* has a negative slope. It is pointing from bottom right to upper left. The *line* is undirected.

  Orientation-Dependent

- NOT *constraint*

  This constraint confirms that the *constraint* is not true.

  Meta-constraint

- OBTUSE *line1 line2*

  The angle from *line1* to *line2* when traveling in a counter-clockwise direction is obtuse. *line1* and *line2* are undirected. The maximum computable angle between undirected lines is 180 degrees. (Deprecated because it is hard to use. Use SLANTED instead.)

  Orientation-Independent

- OBTUSEDIR *line1 line2*

  line1 and *line2* are directed from POINT P1 to POINT P2. The maximum computable angle between directed lines is 360 degrees. The angle between the two lines is obtuse, when measuring in the counter-clockwise direction. (Deprecated because it is hard

to use. Use SLANTED instead.)
Orientation-Independent

- OBTUSEMEET *line1 line2*

  line1 and *line2* meet at one of their endpoints. The lines are directional lines pointing away from their meeting point. These lines form an obtuse angle when measuring in a counter-clockwise direction. The specification of P1 and P2 have nothing to do with the direction of the lines when measuring the angle.
  Orientation-Independent

- ONONESIDE *line shape*

  The bounding box of the *shape* is on a single side of the line. This is syntactic sugar for INTERSECTS.
  Orientation-Independent

- OPPOSITESIDE *line shape1 shape2*

  The center of *shape1* and the center of *shape2* are on the opposite sides of the *line*. If you draw a line between the two center points, that line will intersect the *line* when the two lines are extended to infinity.
  Orientation-Independent

- OR *constraint1 constraint2*

  This constraint checks that at least one of the two constraints listed is true.
  Meta-constraint.

- OVERLAPABOVE *shape1 shape2*

  The y-values overlap, but the center of *shape1* is above the center of *shape2*.
  Orientation-Dependent

- OVERLAPBELOW *shape1 shape2*

  The y-values overlap, but the center of *shape1* is below the center of *shape2*.
  Orientation-Dependent

- OVERLAPLEFTOF *shape1 shape2*

  The x-values overlap, but the center of *shape1* is to the left of the center of *shape2*.
  Orientation-Dependent

- overlapRightOf *shape1 shape2*

  The x-values overlap, but the center of *shape1* is to the right of the center of *shape2*.

  Orientation-Dependent

- parallel *line1 line2*

  line1 is parallel to *line2*. They are both undirected lines. They have the same slope.

  Orientation-Independent

- perpendicular *line1 line2*

  line1 is perpendicular to *line2*. They are both undirected lines. The slopes are reciprocal inverses of each other (-1/m).

  Orientation-Independent

- pointsDown *arc*

  The direction of the arc has an angle of 270. The bump is at the bottom.

  Orientation-Dependent

- pointsLeft *arc*

  The direction of the arc has an angle of 180. The bump is at the left.

  Orientation-Dependent

- pointsRight *arc*

  The direction of the arc has an angle of 0. The bump is at the right.

  Orientation-Dependent

- pointsUp *arc*

  The direction of the arc has an angle of 90. The bump is at the top.

  Orientation-Dependent

- posSlope *line*

  The *line* has a positive slope. It is pointing from bottom left to upper right. The *line* is undirected.

  Orientation-Dependent

- rightOf *shape1 shape2*

  All of *shape1* is to the right of all of *shape2*. The x values do not overlap.

  Orientation-Dependent

- sameSide *line shape1 shape2*

  The center of *shape1* and the center of *shape2* are on the same side of the *line*. If you draw a line between the two center points, that line will not intersect the *line* when the two lines are extended to infinity.

  Orientation-Independent

- sameX *line shape1 shape2*

  The center of *shape1* and the center of *shape2* are at the same x-value. The shapes are vertically aligned.

  Orientation-Dependent

- sameY *line shape1 shape2*

  The center of *shape1* and the center of *shape2* are at the same y-value. The shapes are horizontally aligned.

  Orientation-Dependent

- slanted *line1 line2*

  The angle between *line1* and *line2* is either acute or obtuse. The angle is not parallel or perpendicular. The lines are undirected.

  Orientation-Independent

- smaller *shape1 shape2*

  shape1 is smaller than *shape2*. The diagonal of the boundingBox of *shape1* is shorter than the diagonal of the boundingBox of *shape2*.

  Orientation-Independent

- smaller *line shape1*

  An endpoint of the *line* touches the *shape*.

  Orientation-Independent

- vertical *line*

  The *line* is vertical. The slope is infinite.

  Orientation-Dependent

- vertPosSlope*line*

  The *line* is either vertical or has a positive slope.

Orientation-Dependent

A.3 Predefined Editing Behaviors

A.3.1 Predefined Triggers

The possible triggers include all of those listed here as well as all of the actions listed in Appendix A.3.2, allowing for "chain-reaction" editing.

- CLICK *shape/selection*

  Click the mouse on a *shape* or *selection*.

- DOUBLECLICK *shape/selection*

  Double click the mouse on a *shape* or *selection*.

- CLICKHOLD *shape/selection*

  Click and hold down the mouse over a *shape* or *selection* for a time greater than 0.4 seconds.

- CLICKHOLDDRAG *shape/selection*

  Click and hold down the mouse over a *shape* or *selection* for a time greater than 0.4 seconds, then move the mouse with the mouse button held down.

- DRAW *shape/shape-composition*

  Draw a particular *shape* or *shape-composition*.

- PENOVER *shape/selection*

  Hold the pen over of a *shape* or *selection*. For instance, this constraint may be used to show a special cursor handle to imply that an object can be scaled when the pen rests over one of the corners of the bounding box.

- DRAWOVER *new-shape old-shape/selection*

  Draw a *new-shape* on top of an *old-shape* or *selection*. For instance, one may wish to draw an X over an object to signify deletion.

- SCRIBBLEOVER *shape/selection*

  Draw a scribble over a *shape* or *selection*. A scribble is defined as a back and forth motion repeatedly crossing over an object.

- ENCIRCLE *shapes*

  Draw a closed path around a group of *shapes*. This trigger may be used to select a collection of shapes.

- Any editing action

  Any action (listed in Section A.3.2) also can be used as a trigger.

A.3.2 Predefined Actions

- WAIT *milliseconds*

  Wait for a certain number of *milliseconds* before performing the next action.

- SELECT *shapes*

  Select the collection of *shapes* specified.

- DESELECT *selection*

  Deselect the collection of shapes specified.

- COLOR *shape/selection color*

  Color the *shape* or *selection* the *color* specified.

- DELETE *shape/selection*

  Delete the specified *shape* or *selection*.

- MOVE *shape/selection [x-shift y-shift]*

  If the *x-shift* and *y-shift* are not specified, then translate the specified *shape* or *selection* according to the motion of the mouse. If *x-shift* and the *y-shift* are specified, translate according to the amount specified.

- ROTATE *shape/selection fixed-point [amount]*

  Rotate the specified *shape* or *selection* in reference to the *amount* specified. Rotation occurs around the *fixed-point*. If the *amount* is not specified, then rotate according to the motion of the mouse.

- SCALE *shape/selection fixed-point [amount]*

  Scale the specified *shape* or *selection* in terms of the *amount* specified. The *fixed-point* remains fixed, and the other points move to adjust to the scaling. For instance, when dragging the bottom corner of a square, the *fixed-point* could be the upper corner of the square. If the *amount* is not specified, then scale according to the motion of the mouse.

- RESIZE *shape/selection width height*

  Resize the bounding box of the *shape* or *selection* specified to the *width* and *height* specified. This is done by a combination of scale and translate commands.

- RUBBERBAND *shape/selection fixed-point move-point*

  Translate, scale, and rotate the *shape* or *selection* specified so that the *fixed-point* remains in the same spot, while the *move-point* translates according to the movement of the mouse, and the entire shape or selection remains solid.

- RUBBERBAND *shape/selection fixed-point old-point new-point*

  Translate, scale and rotate the shape or selection specified so that the *fixed-point* remains in the same spot, while the *old-point* translates according to the location of the *new-point*, and the entire shape or selection remains solid.

- SETCURSOR *type point*

  Shows a specialized cursor handle at a particular *point*. The *type* can be NORMAL, MOVE, SCALE, ROTATE, DRAG, PAINT, or TEXT.

- SHOWHANDLE *type point*

  Place an object at a particular *point* specifying that the object can be edited. The *type* can be MOVE, SCALE, or ROTATE.

A.3.3 Predefined Display Methods

The predefined display methods are listed below. The arguments in square brackets are optional.

If only color is specified, then the object is drawn as normal. If a display method other than color is specified, the shape is drawn only as specified. E.g., if the display method PAINTSTRING is listed, then only the string will be displayed unless accompanied by a PAINTCLEANED or other such display command to display the entire drawn shape.

- COLOR *color1* [*shape1*] [*shape2*] [*shape3*]

  Draw the shapes in the color specified. All three shape arguments are optional. If no shapes are included, draw the entire shape in the specified color; else, draw only the shapes listed in *color1*.

- ORIGINALSTROKES [*shape1*] [*shape2*] [*shape3*] [*shape4*]

  All arguments are optional. If no shapes are specified, draw the entire shape using the original strokes. Otherwise, draw only the subshapes specified using the original strokes.

- CLEANEDSTROKES [*shape1*] [*shape2*] [*shape3*] [*shape4*]

  All arguments are optional. This command specifies that the entire shape should be drawn using cleaned strokes using no arguments. Or, it specifies that only the shapes specified in the arguments should be drawn using cleaned strokes. Cleaned strokes is when the primitives are drawn neatly, but nothing more.

- IDEALSTROKES [*shape1*] [*shape2*] [*shape3*] [*shape4*]

Note that Matlab is needed on your machine to get this to work properly. All arguments are optional. This command specifies that the entire shape should be drawn using ideal strokes using no arguments. Or it specifies that only the shapes specified in the arguments should be drawn using ideal strokes. Ideal strokes is when all the constraints are solved before displaying. For instance, two lines may be constrained to meet at their endpoints. When displaying the ideal strokes, *shape*x will be drawn such that these lines actually do meet at their endpoints.

- PAINTPOINT *locationPoint* [*size*] [*color*]
  Draws a point of the *size* specified at the *locationPoint* specified. Note that the last argument is optional, in which case the point is drawn to be of size 2. The color can also be specified. If no *color* is specified, the point will be drawn using the color specified.

- PAINTLINE *start-point end-point [color]*
  This draws a line from the *start-point* to the *end-point.* The color can also be specified. If no *color* is specified, the line will be drawn using the color specified.

- PAINTELLIPSE *center-point width height* [*color*]
  Draws an ellipse specified at the *center-point* with the specified *width* and *height.* The color can also be specified. If no *color* is specified, the ellipse will be drawn using the color specified.

- PAINTRECTANGLE *upper-left-corner-point lower-right-corner-point* [*color*]
  This draws a horizontal rectangle from the *upper-left-corner-point* to the *lower-right-corner-point*. Note that this will still work if the points are reversed; it simply draws the smallest rectangle that surrounds both points. The color can also be specified. If no *color* is specified, the rectangle will be drawn using the color specified.

- PAINTSTRING *string start-point* [*size*] [*color*]
  This draws a text *string* at the specified *start-point. size* specifies the size of

the font, else the default size will be used. The string should not contain any spaces. The color can also be specified. If no *color* is specified, the string will be drawn using the color specified.

- PAINTTEXT *textObject start-point* [*size*] [*color*]
  Draws the text in the *textObject* specified (which had been entered in by the keyboard as part of the drawing and saved as such in the definition) at the specified location. The color can also be specified. If no *color* is specified, the string will be drawn using the color specified.

- PAINTIMAGE *filename centerPoint* [*width*] [*height*]
  This draws the image (.gif or .jpg) in the specified *filename* located in the images director where this shapedef is defined. You can have subdirectories, just replace the slash with a dot. The image is drawn with then center of the image at the *centerPoint*. The image can be scaled to the *width* and *height* specified. If no width or height is specified, the image will be displayed the original size. If just the width is specified, the image will be scaled the shape keeping the original aspect ratio.

Appendix B

Domain Descriptions

This appendix provides sample domain descriptions, including Tic Tac Toe, UML class diagrams, Finite State Machines, and Course of Action Diagrams to aid in understanding, and for use with *LADDER* and *GUILD*. Images of these domains are shown in Chapter 6.2.

B.1 Tic Tac Toe

B.1.1 Domain List

The contents of TicTacToe.ldl are as follows:

```
sketch.shapes.geom.ellipse.Circle
sketch.shapes.tictactoe.Cross
sketch.shapes.tictactoe.Board
sketch.shapes.tictactoe.CircleWin
sketch.shapes.tictactoe.CrossWin
```

B.1.2 Circle Shape Description

The contents of CIRCLE.LSD are as follows:

```
(define shape Circle
  (isA Body)
  (components
    (Ellipse e)
  )
  (display
    (color blue)
    (paintEllipse e.center e.width e.width)
  )
  (editing
    ((drag this) (move this))
  )
)
```

B.1.3 Cross Shape Description

The contents of CROSS.LSD are as follows:

```
(define shape Cross
  (isA Shape)
  (components
    (Line pos)
    (Line neg)
  )
  (constraints
    (posSlope pos)
    (negSlope neg)
    (equalLength pos neg)
    (intersects pos neg)
  )
  (display
    (color green)
  )
  (editing
    ((drag this) (move this))
  )
)
```

B.1.4 Board Shape Description

The contents of BOARD.LSD are as follows:

```
define shape Board
  (isA Shape)
  (components
    (Line top)
    (Line bottom)
    (Line left)
    (Line right)
  )
  (constraints
    (intersects left bottom)
    (intersects left top)
    (intersects right bottom)
    (intersects right top)
    (equalLength left right)
    (equalLength top bottom)
    (vertical left)
    (vertical right)
    (horizontal bottom)
    (horizontal top)
    (not (intersects left right))
    (not (intersects top bottom))
  )
  (display
    (color black)
  )
)
```

B.1.5 CircleWin Shape Description

The contents of CircleWin.lsd are as follows:

```
(define shape CircleWin
  (isA Shape)
  (components
    (Circle c1)
    (Circle c2)
    (Circle c3)
    (Board board)
  )
  (constraints
    (collinear c1.center c2.center c3.center)
    (intersects c1 board)
    (intersects c2 board)
    (intersects c3 board)
  )
  (display
    (color black c1 c2 c3)
    (paintString CIRCLE_WINS board.center)
    (cleanedStrokes)
  )
)
```

B.1.6 CrossWin Shape Description

The contents of CrossWin.lsd are as follows:

```
(define shape CrossWin
  (isA Shape)
  (components
    (Cross c1)
    (Cross c2)
    (Cross c3)
    (context Board board)
  )
  (constraints
    (collinear c1.center c2.center c3.center)
    (intersects c1 board)
    (intersects c2 board)
    (intersects c2 board)
  )
  (display
    (color red)
    (paintString CROSS_WINS board.center)
    (cleanedStrokes)
  )
)
```

B.2 UML Class Diagrams

B.2.1 Domain List

The contents of UML.LDL are as follows:

```
sketch.shapes.uml.Circle
sketch.shapes.uml.Arrow
sketch.shapes.uml.TriangleArrow
sketch.shapes.uml.DiamondArrow
sketch.shapes.uml.Rectangle
sketch.shapes.uml.Interface
sketch.shapes.uml.Class
sketch.shapes.uml.AbstractClass
sketch.shapes.uml.DependencyAssociation
sketch.shapes.uml.GeneralizationAssociation
sketch.shapes.uml.CompositionAssociation
```

B.2.2 Circle Shape Description

The contents of CIRCLE.LSD are as follows:

```
(define shape Circle
  (isA Shape)
  (components
    (Ellipse e)
  )
  (display
    (color magenta)
    (paintEllipse e.center e.width e.height)
  )
  (editing
    ((drag center) (move this))
    ((drag boundBottomRight)
       (scale this boundBottomRight boundTopLeft))
  )
)
```

B.2.3 Arrow Shape Description

The contents of ARROW.LSD are as follows:

```
(define shape Arrow
  (isA AbstractArrow)
  (components
    (Line head1)
    (Line head2)
    (Line shaft)
  )
  (constraints
    (coincident shaft.p1 head1.p1)
    (coincident shaft.p1 head2.p1)
    (longer shaft head2)
    (equalLength head1 head2)
    (acuteMeet shaft head1)
    (acuteMeet head2 shaft)
  )
  (aliases
    (Point head shaft.p1)
    (Point tail shaft.p2)
    (Point pointHead1 head1.p2)
    (Point pointHead2 head2.p2)
  )
  (display
    (color red)
    (cleanedStrokes)
  )
  (editing
    ((drag center) (move this))
    ((drag head) (rubberband this tail head))
    ((drag tail) (rubberband this head tail))
  )
)
```

B.2.4 TriangleArrow Shape Description

The contents of TriangleArrow.lsd are as follows:

```
(define shape TriangleArrow
  (isA AbstractArrow)
  (components
    (Arrow oa)
    (Line head3)
  )
  (constraints
    (coincident head3.p1 oa.pointHead1)
    (coincident head3.p2 oa.pointHead2)
  )
  (aliases
    (Line shaft oa.shaft)
    (Line head1 oa.head1)
    (Line head2 oa.head2)
    (Point head oa.head)
    (Point tail oa.tail)
  )
  (display
    (color pink)
    (cleanedStrokes)
  )
  (editing
    ((drag center) (move this))
    ((drag head) (rubberband this tail head))
    ((drag tail) (rubberband this head tail))
  )
)
```

B.2.5 DiamondArrow Shape Description

The contents of DiamondArrow.lsd are as follows:

```
(define shape DiamondArrow
  (isA AbstractArrow)
  (components
    (Arrow oa)
    (Line d1)
    (Line d2)
  )
  (constraints
    (coincident d1.p1 d2.p1)
    (coincident d1.p2 oa.pointHead1)
    (coincident d2.p2 oa.pointHead2)
  )
  (aliases
    (Line shaft oa.shaft)
    (Line head1 oa.head1)
    (Line head2 oa.head2)
    (Point head oa.head)
    (Point tail oa.tail)
  )
  (display
    (color magenta)
    (cleanedStrokes)
  )
  (editing
    ((drag center) (move this))
    ((drag head) (rubberband this tail head))
    ((drag tail) (rubberband this head tail))
  )
)
```

B.2.6 Rectangle Shape Description

The contents of RECTANGLE.LSD are as follows:

```
(define shape Rectangle
  (isA Shape)
  (components
    (Line top)
    (Line bottom)
    (Line left)
    (Line right)
  )
  (constraints
    (horizontal top)
    (horizontal bottom)
    (vertical left)
    (vertical right)
    (coincident top.p2 right.p1)
    (coincident right.p2 bottom.p1)
    (coincident bottom.p2 left.p1)
    (coincident left.p2 top.p1)
    (above top bottom)
    (leftOf left right)
  )
  (display
    (color magenta)
    (paintRectangle top.p1 bottom.p1)
  )
  (editing
    ((drag center) (move this))
    ((drag boundBottomRight)
       (scale this boundBottomRight boundTopLeft))
  )
)
```

B.2.7 Interface Shape Description

The contents of INTERFACE.LSD are as follows:

```
(define shape Interface
  (isA AbstractClass)
  (components
    (Circle circle)
    (Text text)
  )
  (constraints
    (contains circle text)
  )
  (display
    (color blue)
  )
  (editing
    ((drag center) (move this))
    ((drag text) (move text))
    ((drag boundBottomRight)
       (scale this boundBottomRight boundTopLeft))
  )
)
```

B.2.8 Class Shape Description

The contents of CLASS.LSD are as follows:

```
(define shape Class
  (isA AbstractClass)
  (components
    (Rectangle rect)
    (Text text)
  )
  (constraints
    (contains rect text)
  )
  (display
    (color blue)
    (cleanedStrokes)
  )
  (editing
    ((drag center) (move this))
    ((drag boundBottomRight)
      (scale this boundBottomRight boundTopLeft))
    ((drag text) (move text))
  )
)
```

B.2.9 AbstractClass Shape Description

The contents of AbstractClass.lsd are as follows:

```
(define shape AbstractClass
  (isA Shape)
)
```

B.2.10 DependencyAssociation Shape Description

The contents of DependencyAssociation.lsd are as follows:

```
(define shape DependencyAssociation
  (isA Shape)
  (components
    (Arrow arrow)
    (context AbstractClass classhead)
    (context Class classtail)
  )
  (constraints
    (contains classhead arrow.head)
    (contains classtail arrow.tail)
  )
  (display
    (paintString uses arrow.center)
    (cleanedStrokes)
    (paintString uses arrow.center)
  )
)
```

B.2.11 GeneralizationAssociationShape Description

The contents of GeneralizationAssociation.lsd are as follows:

```
(define shape GeneralizationAssociation
  (isA Shape)
  (components
    (TriangleArrow arrow)
    (context Class headclass)
    (context Class tailclass)
  )
  (constraints
    (contains headclass arrow.head)
    (contains tailclass arrow.tail)
  )
  (display
    (paintString extends arrow.center)
    (cleanedStrokes)
    (paintString extends arrow.center)
  )
)
```

B.2.12 CompositionAssociation Shape Description

The contents of CompositionAssociation.lsd are as follows:

```
(define shape CompositionAssociation
  (isA Shape)
  (components
    (DiamondArrow arrow)
    (context Class headclass)
    (context Class tailclass)
  )
  (constraints
    (contains headclass arrow.head)
    (contains tailclass arrow.tail)
  )
  (display
    (paintString composes arrow.center)
    (cleanedStrokes)
    (paintString composes arrow.center)
  )
)
```

B.3 Finite State Machines

B.3.1 Domain List

The contents of FiniteState.ldl are as follows:

```
sketch.shapes.finitestate.State sketch.shapes.geom.arrow.Arrow
sketch.shapes.finitestate.AbstractState
sketch.shapes.finitestate.State
sketch.shapes.finitestate.InputString
sketch.shapes.finitestate.StartState
sketch.shapes.finitestate.Transition
sketch.shapes.finitestate.AcceptState
```

B.3.2 Arrow Shape Description

The contents of Arrow.lsd are as follows:

```
(define shape Arrow
  (isA Shape)
  (components
    (Line head1)
    (Line head2)
    (Line shaft)
  )
  (constraints
    (coincident shaft.p1 head1.p1)
    (coincident shaft.p1 head2.p1)
    (longer shaft head2)
    (equalLength head1 head2)
    (acuteMeet shaft head1)
    (acuteMeet head2 shaft)
  )
  (aliases
    (Point head shaft.p1)
    (Point tail shaft.p2)
  )
  (display
    (color green)
    (cleanedStrokes)
  )
  (editing
    ((drag center) (move this))
    ((drag head) (rubberband this tail head))
    ((drag tail) (rubberband this head tail))
  )
)
```

B.3.3 AbstractState Shape Description

The contents of AbstractState.lsd are as follows:

```
(define shape AbstractState
  (isA Shape)
)
```

B.3.4 State Shape Description

The contents of STATE.LSD are as follows:

```
(define shape State
  (isA AbstractState)
  (components
    (Ellipse c)
    (Text t)
  )
  (constraints
    (intersects c t)
  )
  (display
    (paintEllipse c.center c.height c.height)
    (paintText t c.center)
    (color blue)
  )
  (editing
    ((drag this) (move this))
  )
)
```

B.3.5 AcceptState Shape Description

The contents of ACCEPTSTATE.LSD are as follows:

```
(define shape AcceptState
  (isA AbstractState)
  (components
    (Ellipse c)
    (State s)
  )
  (constraints
    (intersects c s)
  )
  (display
    (paintEllipse s.center s.height s.height)
    (paintEllipse s.center c.height c.height)
    (color blue)
    (paintText s.t s.center)
  )
  (editing
    ((drag this) (move this))
  )
)
```

B.3.6 StartState Shape Description

The contents of StartState.lsd are as follows:

```
(define shape StartState
  (isA AbstractState)
  (components
    (AbstractState state)
    (Line top)
    (Line bottom)
    (context Text text)
  )
  (constraints
    (posSlope bottom)
    (negSlope top)
    (coincident bottom.p2 top.p2)
    (intersects state top.p2)
    (near text.location top.p1)
  )
  (aliases
    (Point textTop top.p1)
    (Point textBottom bottom.p1)
    (Point textRight top.p2)
  )
  (display
    (color green)
  )
  (editing
    ((drag this) (move this))
  )
)
```

B.3.7 InputState Shape Description

The contents of InputState.lsd are as follows:

```
(define shape InputString
  (isA AbstractState)
  (components
    (context StartState state)
    (Text input)
  )
  (constraints
    (intersects state input)
  )
  (display
    (color red)
  )
  (editing
    ((drag this) (move this))
  )
)
```

B.3.8 Transition Shape Description

The contents of TRANSITION.LSD are as follows:

```
(define shape Transition
  (isA Shape)
  (components
    (Arrow arrow)
    (Text text)
  )
  (constraints
    (intersects arrow text)
  )
  (aliases
    (Point head arrow.head)
    (Point tail arrow.tail)
  )
  (display
    (cleanedStrokes arrow)
    (paintText text arrow.center)
    (color black arrow)
    (color red text)
  )
  (editing
    ((drag head) (rubberband this tail head))
  )
)
```

B.3.9 Backend Code

The finite state machine application checks to see if an input string is accepted by the drawn system. It highlights each state that it passes on the way to be red.

The backend code is as follows:

```
package edu.mit.sketch.language.applink;

import java.awt.Color; import java.util.ArrayList; import
java.util.List;

import edu.mit.sketch.language.shapes.DrawnShape; import
edu.tamu.hammond.sketch.shapes.TText;

public class Finitestate extends AppLink {

  @Override
  public void connect() {
    this.start();
  }
```

```
public void run(){
  DrawnShape startState = null;
  List<DrawnShape> stateList = new ArrayList<DrawnShape>();
  List<DrawnShape> transitionList = new ArrayList<DrawnShape>();
  String inputString = null;

  //classify shapes
  for(DrawnShape s : getViewableShapes()){
    if(s.isOfType("InputString")){
      inputString = ((TText)s.get("input")).getText();}
    if(s.isOfType("StartState")){
      startState = s;}
    if(s.isOfType("AbstractState")){
      stateList.add(s);}
    if(s.isOfType("Transition")){
      transitionList.add(s);}
  }
  //check diagram correctness
  if(inputString == null){ popUp(true, "Please add input string");
    return;}
  if(startState == null){ popUp(true, "can’t find input state");
    return;}

  //highlight states as they are passed
  setPauseColor(startState, Color.red, 3000);

  DrawnShape currentState = startState;
  for(int i = 0; i < inputString.length(); i++){
    DrawnShape nextState = null;
    String letter = inputString.substring(i, i+1);
    for(DrawnShape transition: transitionList){
      if(currentState.contains(transition.get("tail"))){
        if(((TText)transition.get("text")).getText().
             equals(letter)){
          for(DrawnShape state : stateList){
            if(state.contains(transition.get("head"))){
              nextState = state;
              setPauseColor(transition.get("arrow"),
              Color.red, 5000);
              break;
            }
          }
        }
      }
      if(nextState != null){break;}
    }
    if(nextState == null){
      popUp(true, "No Next state for " + letter +
            " transition!") ;
      return;}
    setPauseColor(nextState, Color.red, 3000);
    currentState = nextState;
  }

  //check if final state is an accept state
  if(currentState.isOfType("AcceptState")){
    popUp(false, inputString + " String Accepted");
  } else {
    popUp(false, inputString + " String Rejected");
  }
```

```
  }
}
```

B.4 Course of Action

B.4.1 Domain List

The contents of CourseOfAction.ldl are as follows:

```
sketch.shapes.courseOfAction.Unit
sketch.shapes.courseOfAction.FriendlyUnit
sketch.shapes.courseOfAction.UnitType
sketch.shapes.courseOfAction.EnemyUnit
sketch.shapes.courseOfAction.Command
sketch.shapes.courseOfAction.Supply
sketch.shapes.courseOfAction.Armored
sketch.shapes.courseOfAction.Reconnaissance
sketch.shapes.courseOfAction.Signals
sketch.shapes.courseOfAction.Infantry
sketch.shapes.courseOfAction.BridgeIcon
sketch.shapes.courseOfAction.Bridging
sketch.shapes.courseOfAction.Antitank
sketch.shapes.courseOfAction.Artillery
sketch.shapes.courseOfAction.Motorized
sketch.shapes.courseOfAction.TransportIcon
sketch.shapes.courseOfAction.Transport
sketch.shapes.courseOfAction.Medical
sketch.shapes.courseOfAction.RocketIcon
sketch.shapes.courseOfAction.Rocket
sketch.shapes.courseOfAction.EngineeringIcon
sketch.shapes.courseOfAction.Engineering
sketch.shapes.courseOfAction.Triangle
sketch.shapes.courseOfAction.Observation
```

B.4.2 Unit Shape Description

The contents of UNIT.LSD are as follows:

```
(define shape Unit
  (isA FriendlyUnit)
  (components
    (Line top)
    (Line bottom)
    (Line left)
    (Line right)
  )
  (constraints
    (coincident top.p2 right.p1)
    (coincident right.p2 bottom.p1)
    (coincident bottom.p2 left.p1)
    (coincident left.p2 top.p1)
    (parallel left right)
    (parallel top bottom)
    (perpendicular left bottom)
    (equalLength left right)
    (equalLength top bottom)
    (longer top left)
    (leftOf left.center right.center)
    (above top.center bottom.center)
  )
  (aliases
    (Point topleft top.p1)
    (Point topright right.p1)
    (Point bottomright bottom.p1)
    (Point bottomleft left.p1)
    (Point topmiddle top.center)
    (Point bottommiddle bottom.center)
    (Point rightmiddle right.center)
    (Point leftmiddle left.center)
  )
  (display
    (color green)
  )
  (editing
    ((drag this) (move this))
  )
)
```

B.4.3 FriendlyUnit Shape Description

The contents of FriendlyUnit.lsd are as follows:

```
(define shape FriendlyUnit
  (isA Shape)
)
```

B.4.4 UnitType Shape Description

The contents of UnitType.lsd are as follows:

```
(define shape UnitType
  (isA Shape)
)
```

B.4.5 EnemyUnit Shape Description

The contents of EnemyUnit.lsd are as follows:

```
(define shape EnemyUnit
  (isA Shape)
  (components
    (Unit enemy)
    (context Unit unit)
  )
  (constraints
    (contains enemy unit)
  )
  (display
    (color red)
  )
  (editing
    ((drag this) (move this))
  )
)
```

B.4.6 Command Shape Description

The contents of Command.lsd are as follows:

```
(define shape Command
  (isA Shape)
  (components
    (Line left)
    (Line right)
    (Line top)
    (Line bottom)
  )
  (constraints
    (coincident left.p1 top.p2)
    (coincident top.p1 right.p2)
    (coincident right.p1 bottom.p2)
    (larger left right)
    (parallel left right)
    (leftOf left right)
    (above top bottom)
    (parallel bottom top)
    (perpendicular left top)
    (equalLength top bottom)
  )
  (display
    (color cyan)
  )
)
```

B.4.7 Supply Shape Description

The contents of SUPPLY.LSD are as follows:

```
(define shape Supply
  (isA UnitType)
  (components
    (context Unit unit)
    (Line line)
  )
  (constraints
    (touches line unit.left)
    (touches line unit.right)
    (parallel line unit.bottom)
    (above unit.center line.center)
    (above line.center unit.bottommiddle)
  )
  (display
    (color red)
  )
  (editing
    ((drag this) (move this))
  )
)
```

B.4.8 Armored Shape Description

The contents of ARMORED.LSD are as follows:

```
(define shape Armored
  (isA UnitType)
  (components
    (context Unit unit)
    (Ellipse ellipse)
  )
  (constraints
    (contains unit ellipse)
    (larger ellipse unit.top)
  )
  (display
    (color orange)
  )
  (editing
    ((drag this) (move this))
  )
)
```

B.4.9 Reconnaissance Shape Description

The contents of RECONNAISSANCE.LSD are as follows:

```
(define shape Reconnaissance
  (isA UnitType)
  (components
    (context Unit unit)
    (Line cross)
    (Armored ellipse)
  )
  (constraints
    (coincident unit.bottomleft cross.p1)
    (contains unit ellipse)
    (coincident unit.topright cross.p2)
  )
  (display
    (color cyan)
  )
  (editing
    ((drag this) (move this))
  )
)
```

B.4.10 Signals Shape Description

The contents of SIGNALS.LSD are as follows:

```
(define shape Signals
  (isA UnitType)
  (components
    (context Unit unit)
    (Line top)
    (Line middle)
    (Line bottom)
  )
  (constraints
    (coincident unit.topleft top.p1)
    (coincident top.p2 middle.p1)
    (coincident middle.p2 bottom.p1)
    (coincident bottom.p2 unit.bottomright)
    (parallel middle unit.left)
  )
  (display
    (color green)
  )
)
```

B.4.11 Infantry Shape Description

The contents of INFANTRY.LSD are as follows:

```
(define shape Infantry
  (isA UnitType)
  (components
    (context Unit unit)
    (Line cross1)
    (Line cross2)
  )
  (constraints
    (coincident unit.topleft cross1.p1)
    (coincident unit.bottomright cross1.p2)
    (coincident unit.topright cross2.p1)
    (coincident unit.bottomleft cross2.p2)
  )
  (display
    (color red)
  )
  (editing
    ((drag this) (move this))
  )
)
```

B.4.12 BridgeIcon Shape Description

The contents of BRIDGEICON.LSD are as follows:

```
(define shape BridgeIcon
(isA Shape)
(components
  (Line top)
  (Line bottom)
  (Line leftup)
  (Line leftdown)
  (Line rightup)
  (Line rightdown)
)
(constraints
  (equalLength top bottom)
  (longer top leftdown)
  (longer top rightup)
  (longer top rightdown)
  (longer top leftup)
  (coincident top.p1 leftup.p1)
  (coincident top.p2 rightup.p1)
  (coincident bottom.p1 leftdown.p1)
  (coincident bottom.p2 rightdown.p1)
  (parallel top bottom)
  (parallel leftup rightdown)
  (parallel leftdown rightup)
  (above leftup.p2 leftdown.p2)
  (above rightup.p2 rightdown.p2)
  (above top bottom)
  (leftOf leftup rightdown)
  (near top.center bottom.center)
)
(display
  (color cyan)
  )
)
```

B.4.13 Bridging Shape Description

The contents of BRIDGING.LSD are as follows:

```
(define shape Bridging
  (isA UnitType)
  (components
    (context Unit unit)
    (BridgeIcon bridge)
  )
  (constraints
    (contains unit bridge)
    (parallel unit.top bridge.top)
  )
  (display
    (color green)
  )
)
```

B.4.14 Antitank Shape Description

The contents of ANTITANK.LSD are as follows:

```
(define shape Antitank
  (isA UnitType)
  (components
    (context Unit unit)
    (Line pos)
    (Line neg)
  )
  (constraints
    (coincident unit.topmiddle pos.p1)
    (coincident unit.topmiddle neg.p1)
    (coincident unit.bottomleft pos.p2)
    (coincident unit.bottomright neg.p2)
  )
  (display
    (color orange)
  )
  (editing
    ((drag this) (move this))
  )
)
```

B.4.15 Artillery Shape Description

The contents of ARTILLERY.LSD are as follows:

```
(define shape Artillery
  (isA UnitType)
  (components
    (Ellipse ellipse)
    (context Unit unit)
  )
  (constraints
    (larger unit.left ellipse.boundTop)
    (contains unit ellipse)
  )
  (display
    (color blue)
  )
)
```

B.4.16 Motorized Shape Description

The contents of MOTORIZED.LSD are as follows:

```
(define shape Motorized
  (isA FriendlyUnit)
  (components
    (context Unit unit)
    (Infantry infantry)
    (Line line)
  )
  (constraints
    (coincident unit.topmiddle line.p1)
    (coincident unit.bottommiddle line.p2)
    (concentric unit infantry)
  )
  (display
    (color orange)
  )
  (editing
    ((drag this) (move this))
  )
)
```

B.4.17 TransportIcon Shape Description

The contents of TransportIcon.lsd are as follows:

```
(define shape TransportIcon
  (isA Shape)
  (components
    (Ellipse circle)
    (Line line1)
    (Line line2)
    (Line line3)
    (Line line4)
  )
  (constraints
    (intersects circle line1)
    (intersects circle line2)
    (intersects circle line3)
    (intersects circle line4)
    (equalLength line1 line2)
    (equalLength line1 line3)
    (equalLength line1 line4)
    (concentric line1 circle)
    (concentric line2 circle)
    (concentric line3 circle)
    (concentric line4 circle)
    (perpendicular line1 line3)
    (perpendicular line2 line4)
  )
  (display
    (color green)
  )
  (editing
    ((drag this) (move this))
  )
)
```

B.4.18 Transport Shape Description

The contents of TRANSPORT.LSD are as follows:

```
(define shape Transport
  (isA UnitType)
  (components
    (context Unit unit)
    (TransportIcon transport)
  )
  (constraints
    (contains unit transport)
  )
  (display
    (color green)
  )
  (editing
    ((drag this) (move this))
  )
)
```

B.4.19 Medical Shape Description

The contents of MEDICAL.LSD are as follows:

```
(define shape Medical
  (isA UnitType)
  (components
    (context Unit unit)
    (Line vert)
    (Line hor)
  )
  (constraints
    (coincident unit.topmiddle vert.p1)
    (coincident unit.bottommiddle vert.p2)
    (coincident unit.leftmiddle hor.p1)
    (coincident unit.rightmiddle hor.p2)
  )
  (display
    (color green)
  )
)
```

B.4.20 RocketIcon Shape Description

The contents of ROCKETICON.LSD are as follows:

```
(define shape RocketIcon
  (isA Shape)
  (components
    (Line left)
    (Line middle)
    (Line right)
    (Line pos)
    (Line neg)
  )
  (constraints
    (parallel left middle)
    (parallel middle right)
    (collinear left.center middle.center right.center)
    (equalLength left middle)
    (equalLength middle right)
    (leftOf left middle)
    (leftOf middle right)
    (coincident left.p1 pos.p1)
    (coincident pos.p2 neg.p1)
    (coincident neg.p2 right.p1)
    (collinear middle.p1 middle.p2 neg.p1)
    (above neg.p1 middle)
  )
  (display
    (color cyan)
  )
)
```

B.4.21 Rocket Shape Description

The contents of ROCKET.LSD are as follows:

```
(define shape Rocket
  (isA UnitType)
  (components
    (RocketIcon rocket)
    (context Unit unit)
  )
  (constraints
    (contains unit rocket)
    (parallel unit.left rocket.left)
  )
  (display
    (color green)
  )
)
```

B.4.22 EngineeringIcon Shape Description

The contents of EngineeringIcon.lsd are as follows:

```
(define shape EngineeringIcon
  (isA Shape)
  (components
    (Line top)
    (Line left)
    (Line middle)
    (Line right)
  )
  (constraints
    (larger top left)
    (equalLength left middle)
    (equalLength middle right)
    (coincident top.p1 left.p1)
    (coincident top.center middle.p1)
    (coincident top.p2 right.p1)
    (parallel left middle)
    (parallel right middle)
    (perpendicular top left)
  )
  (display
    (color cyan)
  )
)
```

B.4.23 Engineering Shape Description

The contents of ENGINEERING.LSD are as follows:

```
(define shape Engineering
  (isA UnitType)
  (components
    (context Unit unit)
    (EngineeringIcon engineering)
  )
  (constraints
    (contains unit engineering)
    (parallel unit.top engineering.top)
  )
  (display
    (color green)
  )
)
```

B.4.24 Triangle Shape Description

The contents of TRIANGLE.LSD are as follows:

```
(define shape Triangle
  (isA Shape)
  (components
    (Line line1)
    (Line line2)
    (Line line3)
  )
  (constraints
    (coincident line1.p2 line2.p1)
    (coincident line2.p2 line3.p1)
    (coincident line3.p2 line1.p1)
  )
  (display
    (color cyan)
  )
)
```

B.4.25 Observation Shape Description

The contents of OBSERVATION.LSD are as follows:

```
(define shape Observation
  (isA UnitType)
  (components
    (context Unit unit)
    (Triangle triangle)
  )
  (constraints
    (contains unit triangle)
  )
  (display
    (color orange)
  )
)
```

B.4.26 Back-end Code

The course of action application simply prints out a string description of each shape below it, waits 6 seconds, and then removes the string.

The backend code is as follows:

```
package edu.mit.sketch.language.applink;

import java.util.ArrayList; import java.util.List;

import edu.mit.sketch.language.shapes.DrawnShape; import
edu.tamu.hammond.sketch.shapes.TPoint; import
edu.tamu.hammond.sketch.shapes.TText;

public class CourseOfAction extends AppLink {
  @Override
  public void connect() {
    List<DrawnShape> unitList = new ArrayList<DrawnShape>();
    List<DrawnShape> typeList = new ArrayList<DrawnShape>();
    List<DrawnShape> commandList = new ArrayList<DrawnShape>();
    List<DrawnShape> enemyList = new ArrayList<DrawnShape>();
    for(DrawnShape s : getViewableShapes()){
      if(s.isOfType("Unit")){
        unitList.add(s);}
      if(s.isOfType("UnitType")){
        typeList.add(s);}
      if(s.isOfType("Command")){
        commandList.add(s);}
      if(s.isOfType("EnemyUnit")){
        enemyList.add(s);}
    }

    System.out.println("Commands at: ");
    for(DrawnShape command : commandList){
      System.out.println("  Command : " + command.getCenter());
      TText t = new TText(command.getCenter(), "Command");
      t.setCenter(new TPoint(
          (command.getMinX() + command.getMaxX())/2,
          command.get("bottom").getMaxY() + 10));
      t.setName("added");
      command.addComponent(t);
    }

    for(DrawnShape enemy : enemyList){
      unitList.remove(enemy.get("unit"));
    }

    System.out.println("Friendly Units at: ");
    for(DrawnShape friend : unitList){
      String s = "Friendly ";
      for(DrawnShape type : typeList){
        if(type.get("unit").equals(friend)){
          s += type.getType() + " ";
        }
      }
      s += "Unit";
      TText t = new TText(friend.getCenter(), s);
      t.setCenter(new TPoint(
              (friend.getMinX() + friend.getMaxX())/2,
              friend.getMaxY() + 10));
      t.setName("added");
      friend.addComponent(t);
      System.out.println("  " + s + ": " + friend.getCenter());
    }
    System.out.println("Enemy Units at: ");
    for(DrawnShape enemy : enemyList){
      String s = "Enemy ";
      for(DrawnShape type : typeList){
        if(type.get("unit").equals(enemy.get("unit"))){
          s += type.getType() + " ";
        }
      }
      s += "Unit";
      TText t = new TText(enemy.getCenter(), s);
      t.setCenter(new TPoint(
          (enemy.getMinX() + enemy.getMaxX())/2,
           enemy.getMaxY() + 10));
      t.setName("added");
      enemy.addComponent(t);
      System.out.println("  " + s  + ": " + enemy.getCenter());
    }

    repaint();
    wait(6000);
```

```
    //remove the text strings
    for(DrawnShape s : getViewableShapes()){
      TText added = (TText) s.get("added");
      if(added != null){
        s.removeComponent(added);
        s.moved();
      }
    }
    repaint();
  }
}
```

B.4.27 Images

Because the characteristics of a unit can be combined, many different shapes can be composed from the above shapes. This section shows a variety of different shapes drawn and recognized from the above descriptions.

The Original Hand-drawn Sketches

The Cleaned-up Drawings with System Generated Labels

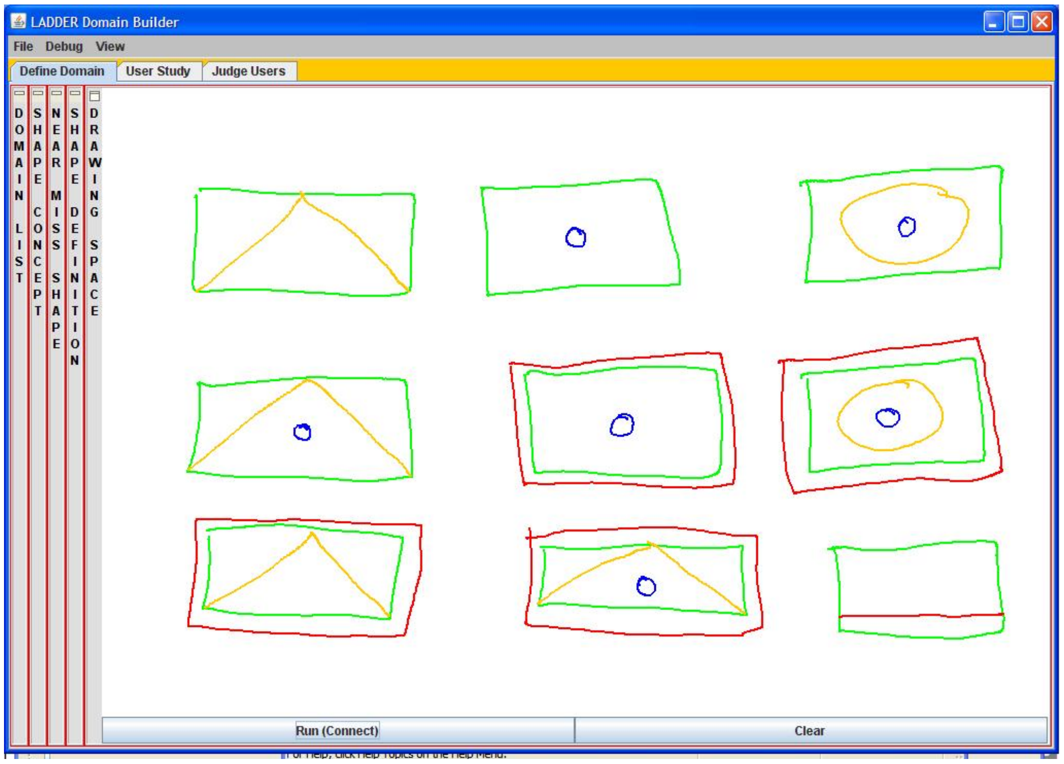


Figure B-1: Example 1 of hand-drawn Course of Action symbols.

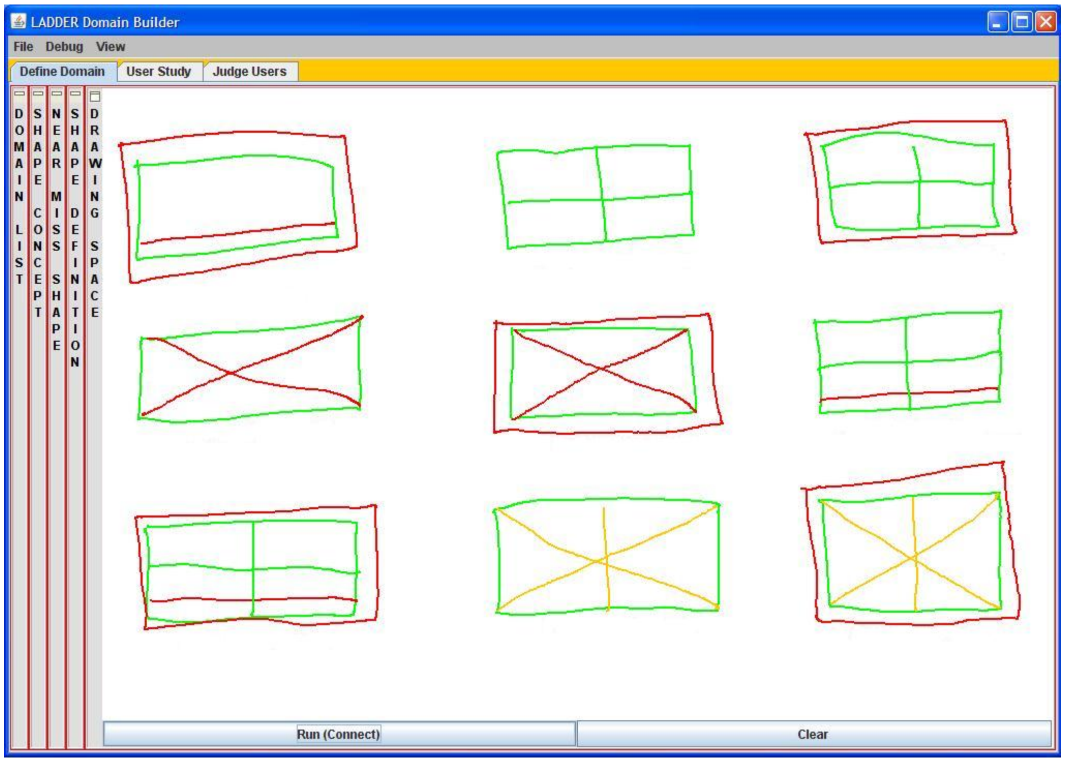


Figure B-2: Example 2 of hand-drawn Course of Action symbols.

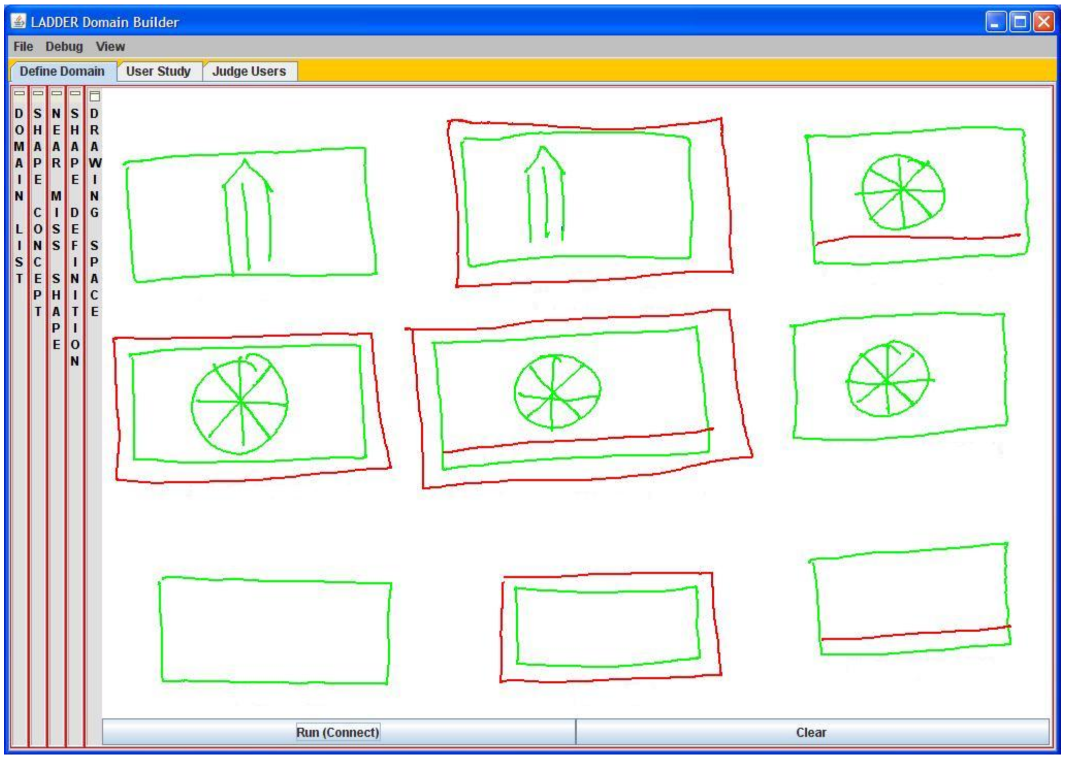


Figure B-3: Example 3 of hand-drawn Course of Action symbols.

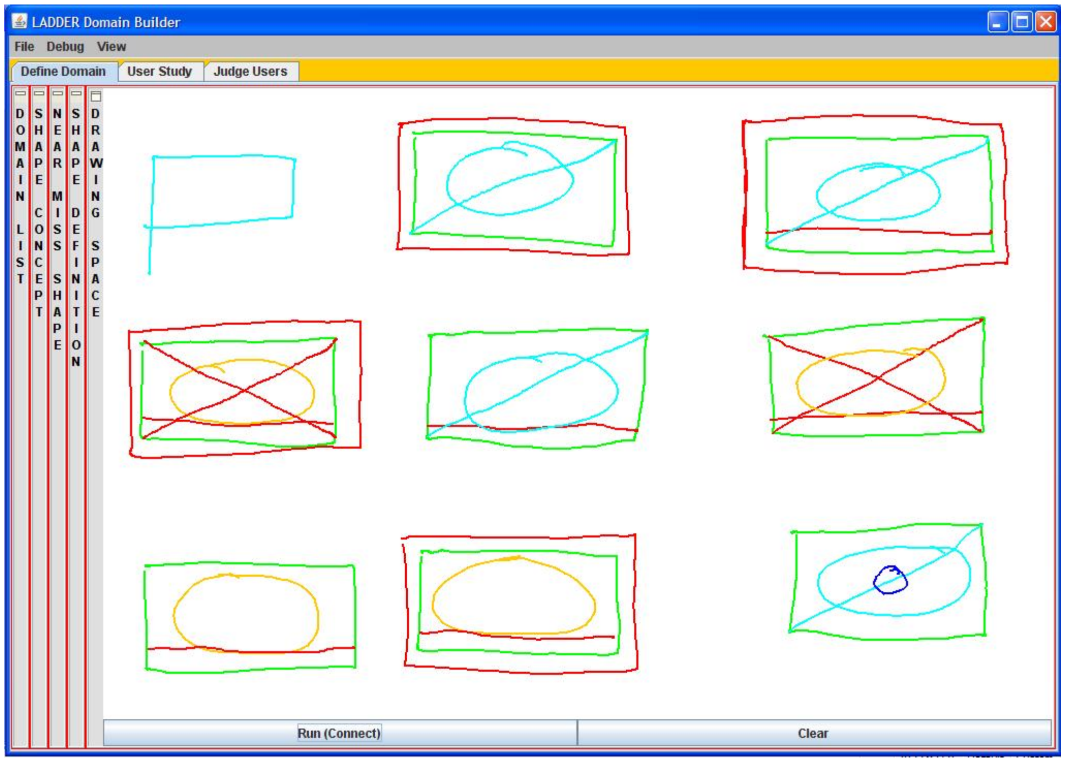


Figure B-4: Example of 4 of hand-drawn Course of Action symbols.

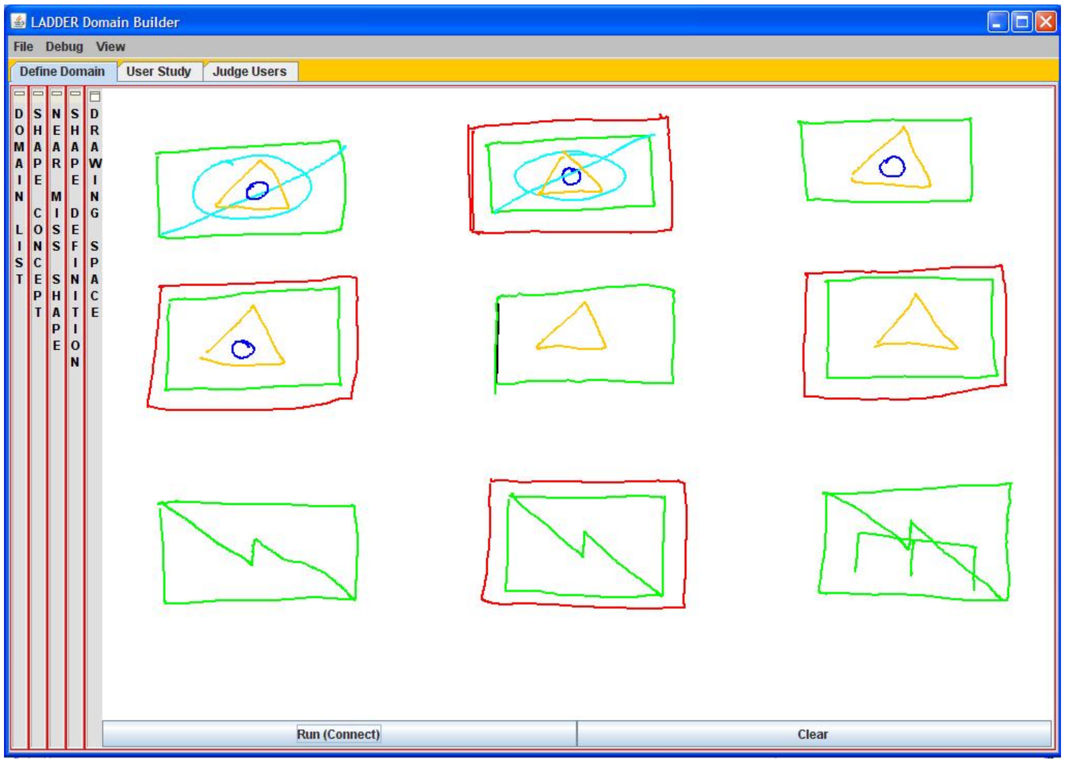


Figure B-5: Example 5 of hand-drawn Course of Action symbols.

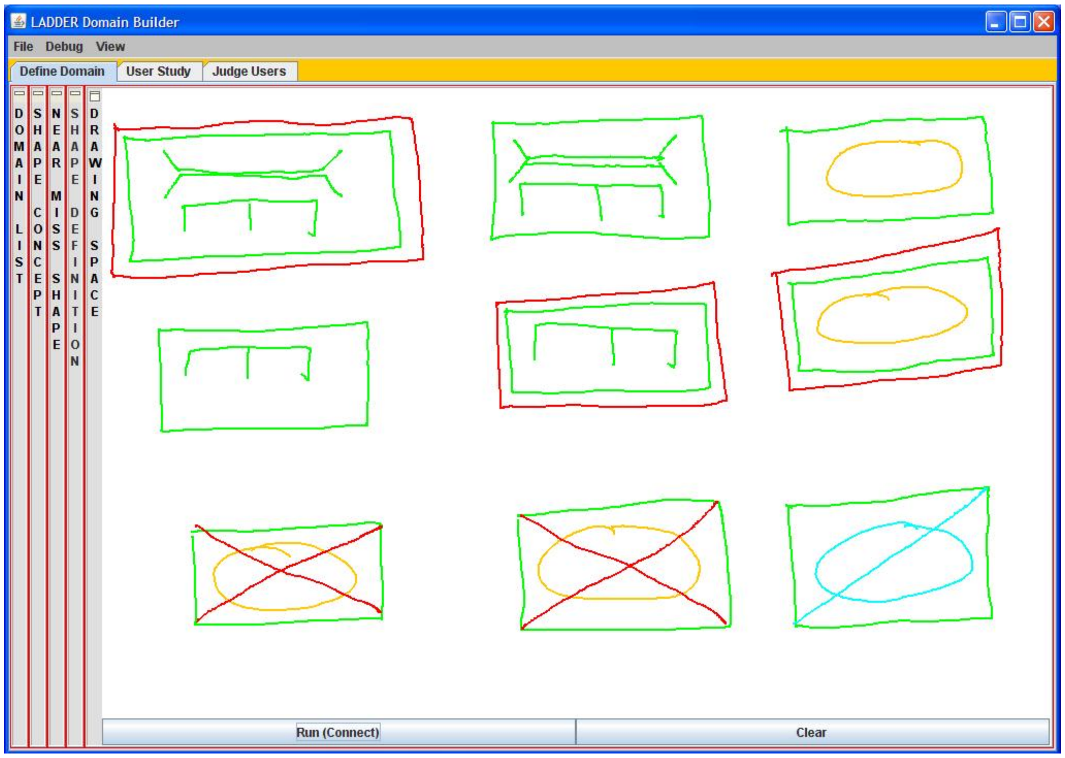


Figure B-6: Example 6 of hand-drawn Course of Action symbols.

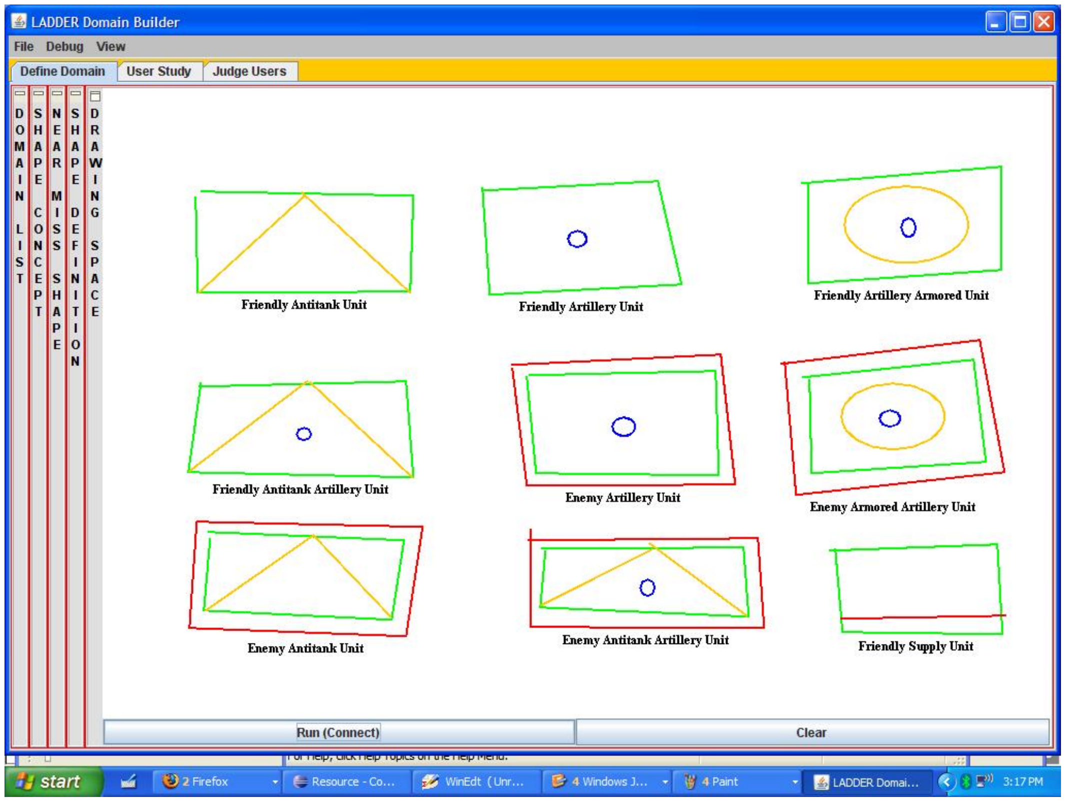


Figure B-7: Example 1 of recognized hand-drawn Course of Action symbols from Figure B-1..

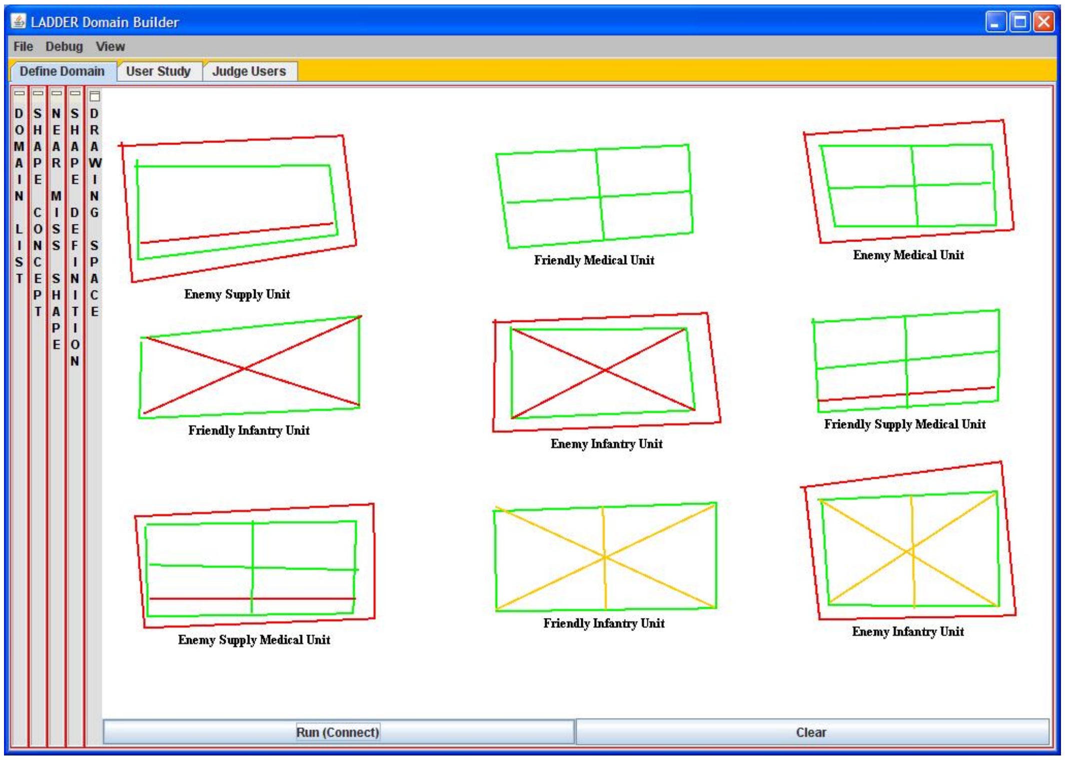


Figure B-8: Example 2 of recognized Course of Action symbols from Figure B-2..

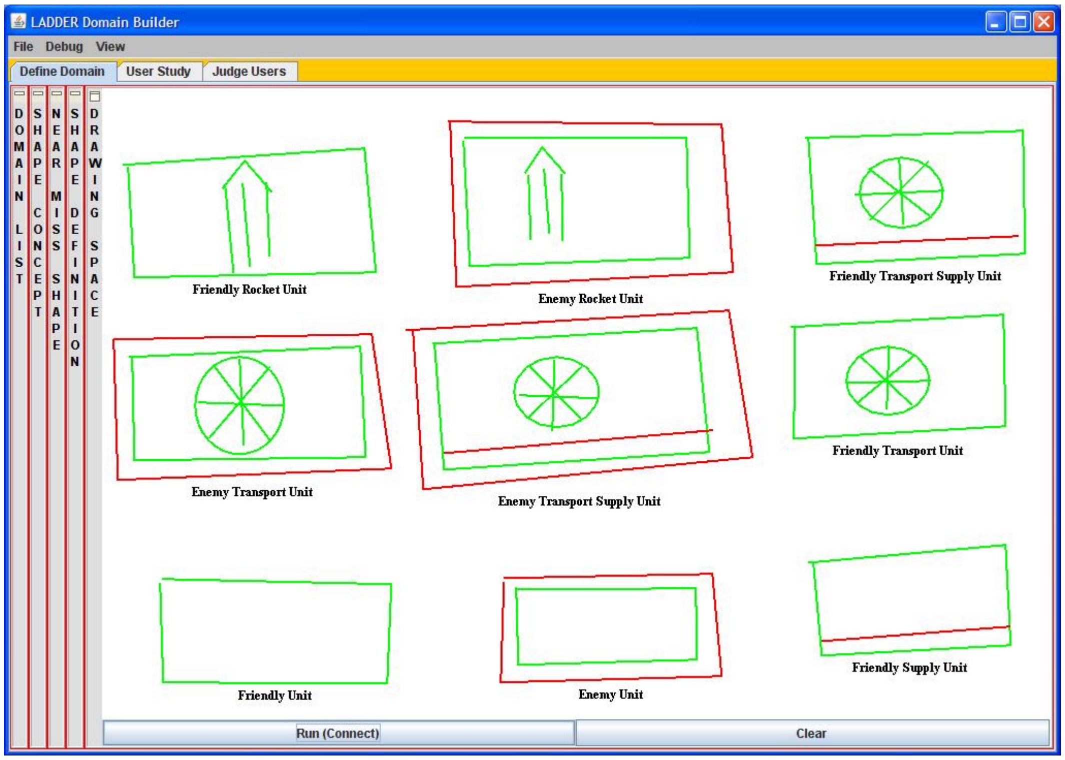


Figure B-9: Example 3 of recognized Course of Action symbols from Figure B-3..

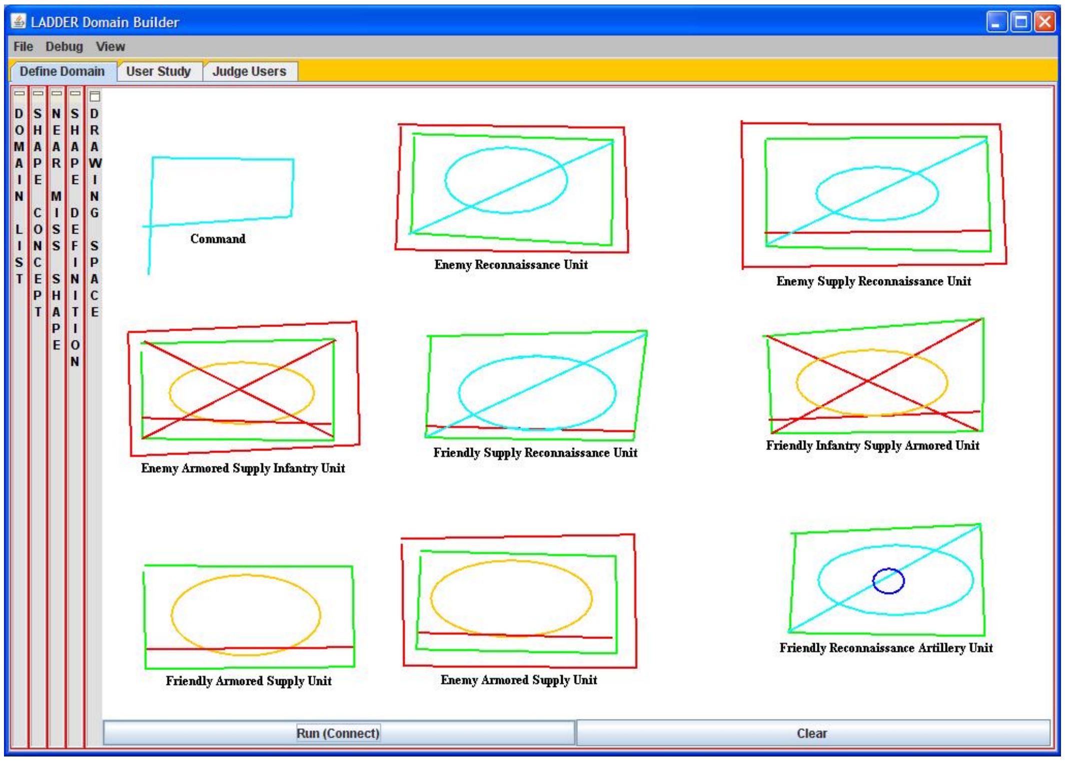

LADDER Domain Builder
File Debug View
Define Domain
User Study
Judge Users
DOMAIN LIST
SHAPE CONCEPT
NEAR MISS SHAPE
SHAPE DEFINITION
DRAWING SPACE
Command
Enemy Reconnaissance Unit
Enemy Supply Reconnaissance Unit
Enemy Armored Supply Infantry Unit
Friendly Supply Reconnaissance Unit
Friendly Infantry Supply Armored Unit
Friendly Armored Supply Unit
Enemy Armored Supply Unit
Friendly Reconnaissance Artillery Unit
Run (Connect)
Clear

Figure B-10: Example 4 of recognized Course of Action symbols from Figure B-4..

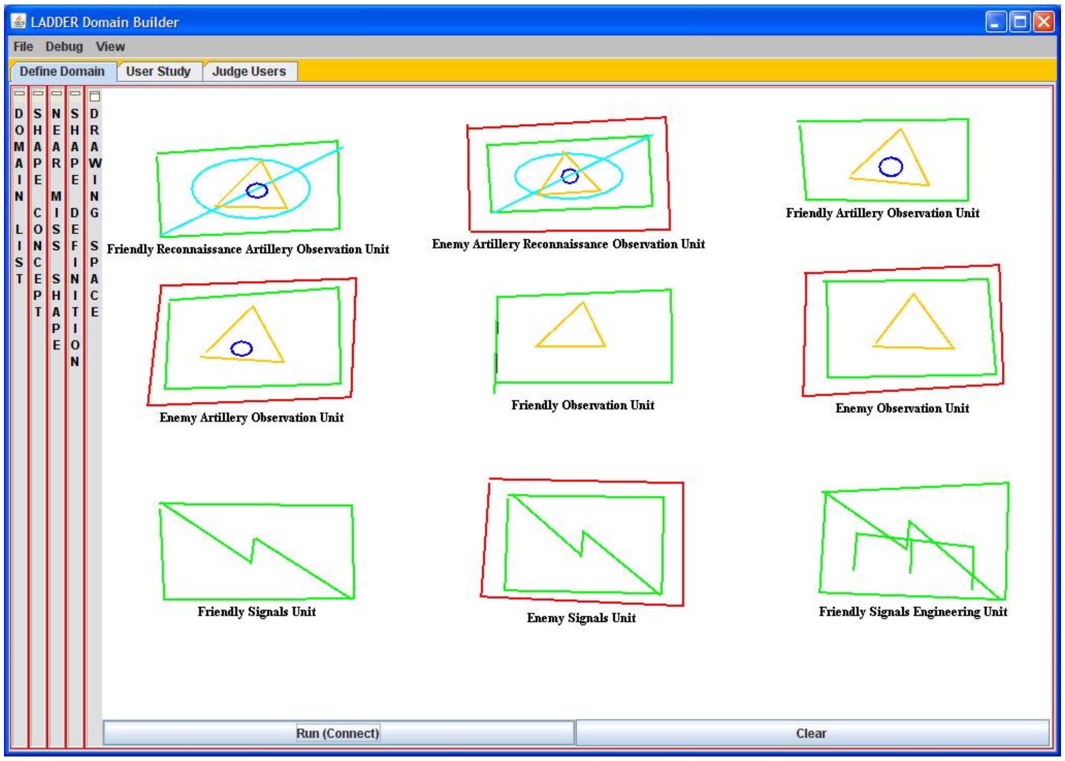

LADDER Domain Builder
File Debug View
Define Domain
User Study
Judge Users
DOMAIN LIST
SHAPE CONCEPT
NEAR MISS SHAPE
SHAPE DEFINITION
DRAWING SPACE
Friendly Reconnaissance Artillery Observation Unit
Enemy Artillery Reconnaissance Observation Unit
Friendly Artillery Observation Unit
Enemy Artillery Observation Unit
Friendly Observation Unit
Enemy Observation Unit
Friendly Signals Unit
Enemy Signals Unit
Friendly Signals Engineering Unit
Run (Connect)
Clear

Figure B-11: Example 5 of recognized Course of Action symbols from Figure B-5..

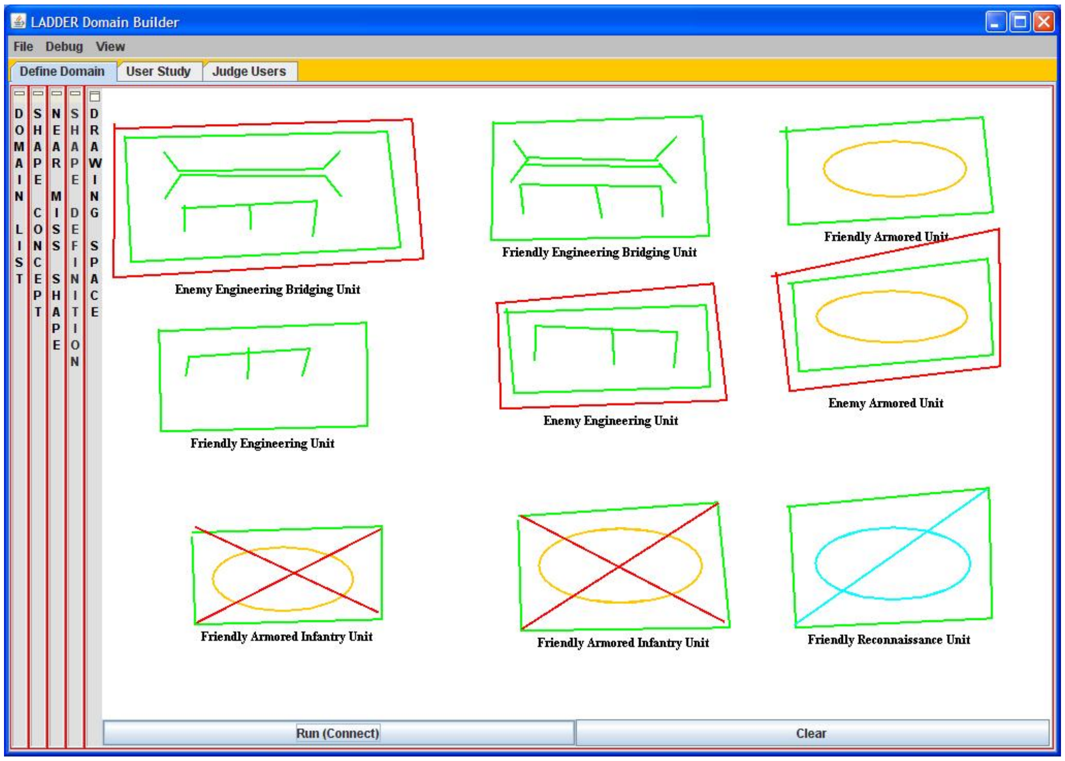


Figure B-12: Example 6 of recognized Course of Action symbols from Figure B-6.

Appendix C

Generating Ideal Shapes with MATLAB: “You’re Getting Warmer.”

One overarching goal of this thesis has been to explain how shapes can be automatically generated from a list of constraints. Chapters 4 and 5 suggested that developers may choose to display the ideal shape with all the constraints solved. In Chapter 10 explains how near miss shapes can be generated automatically by altering a shape description. This chapter describes how that is done, using minimization and optimization in MATLAB.

C.1 Input: Drawn Shape and Description

To generate a shape based on a description, the system needs two things: 1) a description that includes all of the constraints that should be true in our generated shape, and 2) a starting shape that is close to the final solution and provides the initial starting points for the constraint solver. The starting shape should be as close as possible to the final shape, as it makes finding a solution faster and easier than trying to find a solution from a random initial starting point because the algorithm uses hill-climbing to find the solution. When generating the ideal shape for display of a recognized shape, the originally drawn shape (which should be very close to the final shape because it was recognized as an example of the ideal shape with signal error) is chosen as the starting shape. When generating a near-miss, the initial hand-drawn example is chosen as the starting shape. This should be *mathematically* (most of the constraints are already solved) and *perceptually* (thus, the shape seems similar to others shown) close to the final shape, as the goal is to provide a near-miss example that is close to the initial shape, altering the initial shape as little as possible, while testing the chosen constraints.

C.2 Why the Problem Is Difficult

The LADDER constraint set includes both nonlinear and disjunct constraints, which are quite difficult to solve. EqualLength is an example of a nonlinear constraint: the formula for the distance between two points $(x1, y1)$ and $(x2, y2)$ is

$$\sqrt{(x2 - x1)^2 + (y2 - y1)^2}.$$

Disjunct functions include even simple functions such as posSlope, as there exist two separate solutions, one solution where endpoint p1 is in the upper right and endpoint p2 is in the lower left, and another solution where endpoint p2 is in the upper right and endpoint p1 is in the lower left. Because there exists a requirement that lines have a length of greater than 30 pixels (to prevent lines of imperceptible length), there is no way to transition between the two solutions.

Also, finding the minimum value of a nonlinear objective function requires an exhaustive search of the space, which takes an impractical amount of time for large integer-based graphs, and is impossible on a real-number-based graph.

C.3 Choosing the Appropriate MATLAB Function

MATLAB has several functions for solving nonlinear constraints:

fmincon : Find a minimum of constrained nonlinear multivariable function

fminunc : Find minimum of unconstrained multivariable function

fminsearch : Find minimum of unconstrained multivariable function using derivative-free method

All three functions minimize a nonlinear multivariable function, taking as inputs a function to be minimized and initial starting values for the variables to be determined. In our case, the variables to be determined are the $(x1, y1), (x2, y2)$ endpoints of a line, as well as the top left corner and bottom right corner of the smallest rectangle enclosing an ellipse (so that $x2 - x1$ represents the width of the ellipse, and $y2 - y1$ represents the height of the ellipse). [1]

fmincon not only minimizes an objective function, but also allows the user to specify *less than* and *equality* constraints on the variables to be determined. Thus, both inequality constraints, including the requirement that $x1$ must lie on the screen $(0 < x1 < 500)$, and equality constraints, including specifying that two lines must be of equal length or parallel, can be specified in this matter. However, constraints are either true or false, and *fmincon* often has no way of knowing if it is getting closer to a value that solves the constraint or not, especially for nonlinear disjunct constraints. Thus, in order to guarantee that *fmincon* produces a solution to the constraints, *it must be given an initial feasible solution as the starting condition.* While *fmincon* can find a solution for simple equations, moderately difficult systems of equations fail to produce a solution. (E.g., a system of equations constraining three lines succeeds less than 10% of the time, and a system of equations constraining

[1]When necessary the system appropriately translates between the differing coordinate systems in Java and Matlab.

four lines almost never produces a feasible solution, when not given a correct initial starting position.) Thus, for *fmincon* to be useful, the function needs a solution that solves the constraints before trying to solve the constraints. This is impractical, since the solution is unknown, and if it was known, there would be no need of a constraint solver. However, a constraint for which initial starting variables that solve the constraint can easily be computed can still be specified in the constraint section of *fmincon*. For example, the upper and lower bound requirements ($0 < x1 < 500$) can be included in the list of constraints to be solved by the numerical solver, as the system can easily produce initial values that abide by this constraint, (Any shape drawn on the page will conform to this constraint.) The remaining constraints will be moved into the objective function, and, instead, of trying to solve these constraints, the system will try to minimize the value of the objective function (which translates into the error of the constraints to be solved). This does not present a problem as it is possible to translate all of the *LADDER* constraints into a function which produces an error to be added to the output of the objective function (described below). It is still possible to generate the proposed shape even when all of the constraints (including the simple boundary conditions) are moved to the objective function. With this method, MATLAB generate the proposed shape faster and with more reliability.

fminunc solves unconstrained minimization problems. If all constraints are moved into the objective function, then *fmincon* and *fminunc* perform similarly. Because all constraints can be moved to the objective function with similar results, *fmincon* and *fminunc* are functionally equivalent for this problem.

The third function, *fminsearch*, solves unconstrained minimization problems, but is different from *fminunc* in that it does not use a derivative-based search method, which is used by both *fmincon* and *fminunc*. *fmincon* and *fminunc* require that the constraint (in *fmincon* only) and objective (in both) functions are continuous, since they use a gradient-based method that is designed to work on problems where the objective and constraint functions are both continuous and have continuous first derivatives. However, *fminsearch* does not require that the function be continuous, and solves nondifferentiable problems and can often handle discontinuity, particularly if it does not occur near the solution.

Since the generated objective function is often discontinuous (as explained above) and all of the constraints can be moved to the generated objective function, this researcher choose to use *fminsearch* function predominately.

However, even though the MATLAB manual states that "*fminsearch* solves nondifferentiable problems and can often handle discontinuity," one must note the operative use of the word "often." *fminsearch* still does not guarantee that it will find a solution, if one exists. All three functions, including *fminsearch*, might give only locally optimal solutions.

C.4 Generating the Objective Function

Our goal in creating a successful objective function is to create a function that continuously lets the program know that it is getting closer to the solution. Each function returns an error associated with that constraint, returning zero when the constraint is solved, or a positive number representing the error relative to the distance to the solution.

The equation below explains the objective function $O(\mathbf{x})$; $\mathbf{x}$ represents a solution vector, O represents the objective function, and $C_i(\mathbf{x})$ represents a numerical constraint function for the i^{th} shape description constraint that takes in the solution vector $\mathbf{x}$ and returns a scalar value representing the error for that constraint.

$$O(\mathbf{x}) = C_1(\mathbf{x}) + C_2(\mathbf{x}) + ... + C_n(\mathbf{x})$$

For an example of a numerical constraint function, look at the equalLength constraint. One could create a function that would return 0 if the two lines were of equal length and 1 if they are not of equal length, but that would not let the computer know if it was approaching the solution or deviating from it. One could, instead, create a function that returns the absolute value of the difference between the two line lengths.[2] In this way, as the line lengths become closer, the error decreases.

Another example is the constraint posSlope. In this case, one could use a similar technique and return the absolute value of the actual line angle minus the ideal angle of a positively-sloped line of 45 degrees. However, while that error function does lead the computer to a correct solution, it does not give the computer the allowed flexibility; to the computer, the transformation of a line from 0 to 20 degrees is the same as a transformation of a line from 25 to 45 degrees, as the change in error is the same. However, as shown and discussed earlier in this thesis, humans are much more perceptually clued into changes from 0 to 20 than from 25 to 45. A change of a line from 0 to 20 degrees changes the line from horizontal to positively sloped, whereas a change from 25 to 45 degrees does not change its perceptual description. Therefore, the chosen error function to test positively-sloped lines returns 0, if the line is between 25 and 75 degrees; otherwise, it returns the distance from the closer of the two angles.

Appendix D lists the objective function used for each constraint.

C.5 Finding a Solution

MATLAB is not guaranteed to find a solution. It may fail because the presented geometric shape constraints are impossible, or it may fail simply because it got stuck in a local minimum.

My system first wants to be able to ascertain whether MATLAB found a solution that solved all of the given geometric shape constraints. If MATLAB did find a solution, then the numerical solver should have found a solution with values for the vector $\mathbf{x}$ such that, ideally, $O(\mathbf{x}) = 0$, or, rather, practically, such that $O(\mathbf{x}) < \epsilon$,

[2]The system computes the absolute value so that the difference approaches zero. If the system were to subtract without the absolute value, it would end up with the minimum difference approaching negative infinity, with one line much larger than the other.

where ϵ is some small error bound. (The system uses $\epsilon = .05$.)

On the first attempt, the system sets the initial starting values for $\mathbf{x}$ to be the values in the initial drawn shape. Since MATLAB may not find a solution on the first try, the numerical solver is run several times. The numerical solver is run using both *fminsearch* and *fmincon*, since *fmincon* uses a different hill climbing algorithm and may produce different results than *fminsearch*. (Five is the current maximum number of times each is tried.) If any solution $\mathbf{x}$, presented by the solver, returns $O(\mathbf{x}) < \epsilon$, the system halts and returns that solution for $\mathbf{x}$. Otherwise, the system returns the solution for $\mathbf{x}$ that produced the minimum value for $O(\mathbf{x})$, along with the $O(\mathbf{x})$ value which represents the error. The application then decides whether or not to use the imperfect solution.

Appendix D

MATLAB Code to Generate an Ideal Shape

This appendix describes and provides the MATLAB code used to generate an ideal shape. Some of the inputs may appear to be strange, as MATLAB places constraints on input values of the functions *fmincon* and *fminsearch*, as well as the format of the inputs of the objective function. Also, several functions convert values to numerical form or to matrices in order to speed up MATLAB computations. Because this code also works with circles and arcs (which is not shown in the code here, in an effort to make things as easy as possible to understand, and because the handling of them is still in flux), there are many functions that may appear to perform trivial operations. These are explained in the text below.

This appendix includes the access function, the objective function, the constraint functions, which compute the error values for each of the constraints, and some, but not all of the helper functions. Each of the functions is described in detail for ease of implementation.

D.1 Access Function: createshape

The CREATESHAPE function is the initial function that is called by the Java program to create a shape.

D.1.1 Inputs

The function takes in:

- X0: a vector of the initial values. X0 is a vector of double values representing each of the x- and y-values of each of the lines; e.g., for a single lined shape, X0 = [X1 Y1 X2 Y2]. For a two-lined shape, X0 = [X1 Y1 X2 Y2 X3 Y3 X4 Y4], where X1 and X2 represent the endpoints of the first line, and X3 and X4 represent the endpoints of the second line. Multiple lines are represented similarly.
- STRCONSTRAINTS: a vector of the list of the string constraints to be solved. The elements in the list are in the form PARALLEL LINE1 LINE2.
- ATTEMPTS: the number of times to try to create the shape divided by two. For each ATTEMPT, the function tries to solve the constraints with FMINSEARCH and FMINCON.
- UPPERB: the upper bound on the values of the X solution vector. This corresponds to the maximum width and height of the screen display.

D.1.2 Outputs

The function returns:

- X: a vector of the solution values of the shape. These values have the same form as in X0. The solution values are each between 0 and UPPERB.

- FVAL: the error of the solution chosen. This is the value returned from the objective function.
- EXITFLAG: a flag stating if operation terminated normally. A 1 signifies normal completion.
- OUTPUT: a debug output of the results.

D.1.3 Explanation of Internals

The code process is as follows:

1. Translate the constraints into numerical values using the function TRANSLATE-CONSTRAINTS. The code first checks that the constraints have not already been translated.
2. Sets the display options to display fewer warnings.
3. Creates a vector of the upper UPPERB and lower 0 bound for each of the values in the solution matrix, X.
4. Makes a line matrix. Changes X0 from a vector of x- and y-values to a matrix of lines, where each line is of the form [X1 Y1 X2 Y2].
5. For each ATTEMPT,
   (a) if it is the first ATTEMPT, then the initial values are as set in X0. Otherwise, the function sets random initial values within the upper and lower bounds.
   (b) the function attempts to solve the constraint list using first FMINSEARCH, then FMINCON. If ever a solution is returned with a error value (FVAL) less than .001, then that solution is returned immediately. Otherwise, all ATTEMPTS are tried and the function returns the solution with the smallest error value (FVAL). The error value is the value returned by the

objectiveFunction. If the exitFlag is not 1, then it is probable that the solution is not valid, and does not hold true the constraints.

6. The returned shape is plotted using the function. plotLine

D.1.4 Code

```
function [x, fval, exitflag, output] = ...
  createshape (x0, strConstraints, attempts, upperb)

%translates constraints into numerical values
if isnumeric(strConstraints(1,1))
  constraints = strConstraints;
else
 constraints = translateConstraints(strConstraints);
end

%sets the debug display options
options = optimset('LargeScale','off','Display','off',
'MaxFunEvals', ...
  500, 'MaxIter', 500); %, 'Display', 'Iter');

%sets the upper and lower bound of each of the values
for i = 1 : length(x0)
    lb(i) = 0;
    ub(i) = upperb;
end

%changes x0 from a vector of x and y values to a
%matrix of lines, where each line is of the form
```

```
%[x1 y1 x2 y2]
lineMatrix = makeLineMatrix(x0);

%initial value so all value are smaller
minfval = 10000;

for i=1:attempts
   if i==1
       xstart=x0;
   else
       %upperb should be 500 -
       %else created invalid first shape
       xstart=upperb*rand(1,length(x0));
   end
   [x,fval, exitflag, output] = ...
     fminsearch(@(x)objectiveFunction(x, lineMatrix, ...
                constraints, 0), xstart,options);
   if exitflag > 0 && fval < .01
       break;
   end
   if fval < minfval
       minfval = fval;
       minx = x;
       minexit = exitflag;
       minoutput = output;
   end
   [x,fval, exitflag, output, lambda, grad, hessian ] = ...
     fmincon(@(x)objectiveFunction(x, lineMatrix, ...
        constraints, 0), ...
        xstart, [], [], [], [], lb, ub, ...
     @(x)nonlinearinequalities(x, lineMatrix, ...
         constraints), options);
   if exitflag > 0 && fval < .01
       break
   end
   if fval < minfval
       minfval = fval;
       minx = x;
       minexit = exitflag;
       minoutput = output;
   end
   fval = minfval;
   x = minx;
   exitflag = minexit;
   output = minoutput;
end

%plots the shape created
plotLine(x);
```

D.2 Objective Function: objectiveFunction

This function computes the error value of the constraints for a particular set of x and y-values. This is the most important function of the code.

D.2.1 Inputs

- `x`: A vector of solution values to be tested

- LINEMATRIX: A matrix, where each vector contains the X indices pertaining to a line.
- CONSTRAINTS: A listing of the constraints to be solved in numerical form.
- DEBUG: A flag for debug output.

D.2.2 Output

This function returns the error value for the CONSTRAINTS on that input vector X.

D.2.3 Explanation of Internals

1. The function creates a matrix holding all of the current values for each of the lines.
2. If any of those lines are shorter than 30 pixels, a penalty is placed by adding to the variable NUM which represents the total error of the constraints.
3. If any of the x- or y-values is less than 0, or greater than 500, (i.e., the shape is off the screen), a penalty is given by adding to the error value, NUM.
4. For each of the constraints:
   (a) Each of the arguments of the constraints is loaded so the values are easy to access.
   (b) The error of the constraint is computed for those argument values, and the error added to the error value, NUM.

D.2.4 Code

```
function num = objectiveFunction(
    x, lineMatrix, constraints, debug)

global m_horizontal m_vertical m_posSlope m_negSlope ...
  m_near m_far m_intersects m_bisects_cc m_bisects_c1  ...
  m_bisects_c2 m_bisects_1c m_bisects_2c m_connected_11  ...
  m_connected_12 m_connected_21 m_connected_22 ...
  m_bisects_Lc m_bisects_cL m_meets_L1 m_meets_L2 ...
  m_meets_1L m_meets_2L m_sameX  m_leftOf m_rightOf ...
  m_overlapLeftOf m_overlapRightOf m_sameY m_above ...
  m_below m_overlapAbove m_overlapBelow  m_parallel ...
  m_perpendicular m_slanted m_acuteMeet m_obtuseMeet ...
  m_equalArea m_larger m_smaller m_equalAngle m_mapping

num = 0;

%creates a matrix holding all of the current line values
[a,b] = size(lineMatrix);
for i = 1 : a
  lines(i,1) = x(lineMatrix(i,1));
  lines(i,2) = x(lineMatrix(i,2));
  lines(i,3) = x(lineMatrix(i,3));
  lines(i,4) = x(lineMatrix(i,4));
  line1 = lineToPoints(lines(i,:));
  if getLineLength(line1) < 30
      num = num + 30 - getLineLength(line1);
  end
  for j = 1:4
  if lines(i, j) < 0
      num = num - lines(i,j);
  end
  if lines(i,j) > 500
      num = num + lines(i, j) - 500;
  end
  end
end

[a,b] = size(constraints);
for i = 1 : a
    row = constraints(i,:);
    constraint = row(1);
    if row(2) > 0
        line1 = lineToPoints(lines(row(2),:));
        l1x1 = line1(1,1);
        l1y1 = line1(1,2);
        l1x2 = line1(2,1);
        l1y2 = line1(2,2);
    end
    if row(3) > 0
        line2 = lineToPoints(lines(row(3),:));
        l2x1 = line2(1,1);
        l2y1 = line2(1,2);
        l2x2 = line2(2,1);
        l2y2 = line2(2,2);
    end
    if row{4} > 0
      line3 = lineToPoints(lines(row{3}, :));
    end
    if row{5} > 0
        line4 = lineToPoints(lines(row{4}, :));
    end
```

```
if constraint == m_horizontal
    num = num + abs(l1y1 - l1y2);
    %penalty = the difference between the y values
elseif constraint == m_vertical
    num = num + abs(l1x1 - l1x2);
    %penalty = the difference between the x values
elseif constraint == m_posSlope
    num = num + posSlope(line1);
    %penalty = the distance from 45
    %(unless between 15 and 75)
elseif constraint == m_negSlope
    num = num + negSlope(line1);
    %penalty = the distance from -45
    %(unless between -15 and -75)

elseif constraint == m_equalArea
    num = num + abs(getLineLength(line1)
      - getLineLength(line2));
    %penalty = the difference between the line lengths
elseif constraint == m_larger
    num = num + larger(line1, line2);
    %penalty equals how much larger line1 must be
    %to be more than twice the length of line2
elseif constraint == m_smaller
    num = num + larger(line2, line1);
    %penalty equals how much larger line2 must be
    %to be more than twice the length of line1

elseif constraint == m_sameX
```

```
    num = num + abs(l1x1 + l1x2 - l2x1 - l2x2);
    %penalty equals the distance between the
    %x values of the midpoint
elseif constraint == m_sameY
    num = num + abs(l1y1 + l1y2 - l2y1 - l2y2);
    %penalty equals the distance between the
    %y values of the midpoint
elseif constraint == m_leftOf
    num = num + leftOf(line1, line2);
    %penalty equals the how much more to the left
    %line1 needs to go to be completely
    %to the left of line2
elseif constraint == m_rightOf
    num = num + leftOf(line2, line2);
    %penalty equals the how much more to the right
    %line1 needs to go to be completely
    %to the right of line2
elseif constraint == m_above
    num = num + above(line1, line2);
    %penalty equals the how much more up
    %line1 needs to go to be completely above line2
elseif constraint == m_below
    num = num + above(line2, line1);
    %penalty equals the how much more down
    %line1 needs to go to be completely below line2
elseif constraint == m_overlapLeftOf
    num = num + overlapLeftOf(line1, line2);
    %penalty equals how much more to have the center
    %of line1 to the left of the center of line2 or
    %to have line1's x-values overlap line2's x-values
```

```
 elseif constraint == m_overlapRightOf
     num = num + overlapLeftOf(line2, line1);
     %penalty equals how much more to have the center
     %of line1 to the right of the center of line2 or
     %to have line1's x-values overlap line2's x-values
 elseif constraint == m_overlapAbove
     num = num + overlapAbove(line1, line2);
     %penalty equals how much more to have the center
     %of line1 above the center of line2 or
     %to have line1's y-values overlap line2's y-values
 elseif constraint == m_overlapBelow
     num = num + overlapAbove(line2, line1);
     %penalty equals how much more to have the center
     %of line1 below the center of line2 or
     %to have line1's y-values overlap line2's y-values

elseif constraint == m_parallel
     [v1x, v1y] = getDirectionVector(line1);
     [v2x, v2y] = getDirectionVector(line2);
     num = num + abs(v1x - v2x) + abs(v1y - v2y);
     %penalty equals the difference in the change in
     %x's, plus the difference in the change in y's
 elseif constraint == m_perpendicular
     num = num + perpendicular(line1, line2);
     %penalty equals the difference from zero of the
     %change in x's times the change in y's, using the
     %slope formula for perpendicular lines
 elseif constraint == m_acuteMeet
     num = num + acuteMeet(line1, line2);
     %penalty equals the distance between the closest
```

```
    %endpoints plus a penalty if they are not
    %perceptually acute
elseif constraint == m_obtuseMeet
    num = num + obtuseMeet(line1, line2);
    %penalty equals the distance between the closest
    %endpoints plus a penalty if they are not
    %perceptually obtuse
elseif constraint == m_slanted
    num = num + slanted(line1, line2);
    %penalty equals the minimum of the posSlope and
    %negSlope penalties

elseif constraint == m_near
    num = num + near(line1, line2);
    %penalty equals the distance from the near boundaries
elseif constraint == m_far
    num = num  + far(line1, line2);
    %penalty equals the distance from the far boundary
elseif constraint == m_intersects ||
       constraint == m_bisects_cc
   num = num + getLineLength([getLineMidpoint(line1); ...
     getLineMidpoint(line2)]);
   %penalty equals the distance between
   %the two line centers

elseif constraint == m_bisects_c1
  num = num + getLineLength(
        [getLineMidpoint(line1); line2(1,:)]);
  %penalty equals the distance between the center of
  %line1 and endpoint 1 of line2
```

```
elseif constraint == m_bisects_c2
  num = num + getLineLength(
         [getLineMidpoint(line1); line2(2,:)]);
  %penalty equals the distance between the center of
  %line1 and the second endpoint of line2
elseif constraint == m_bisects_1c
  num = num + getLineLength(
         [getLineMidpoint(line2); line1(1,:)]);
  %penalty equals the distance between the first endpoint
  %of line1 and the center of line2
elseif constraint == m_bisects_2c
  num = num + getLineLength(
         [getLineMidpoint(line2); line1(2,:)]);
  %penalty equals the distance between the second
  %endpoint of line1 and the center of line2

elseif constraint == m_connected_11
  num = num + 10*getLineLength([line1(1,:); line2(1,:)]);
  %penalty equals the distance between the first endpoint
  %of line1 and the first endpoint of line2
elseif constraint == m_connected_12
   num = num + 10 *
         getLineLength([line1(1,:); line2(2,:)]);
   %penalty equals the distance between the first
   %endpoint of line1 and the second endpoint of line2
elseif constraint == m_connected_21
   num = num + 10 *
         getLineLength([line1(2,:); line2(1,:)]);
   %penalty equals the distance between the second
   %endpoint of line1 and the first endpoint of line2
```

```
elseif constraint == m_connected_22
    num = num + 10 *
          getLineLength([line1(2,:); line2(2,:)]);
    %penalty equals the distance between the second
    %endpoint of line1 and the second endpoint of line2
elseif constraint == m_bisects_Lc
    num = num + getDistanceToLine(
          getLineMidpoint(line2), line1);
    %penalty equals the distance between the line1
    %and the center of line2
elseif constraint == m_bisects_cL
    num = num + getDistanceToLine(
          getLineMidpoint(line1), line2);
    %penalty equals the distance between the center of
    %line1 and the line2
elseif constraint == m_meets_L1
    num = num + getDistanceToLine(line2(1,:), line1);
    %penalty equals the distance between line1 and
    %the first endpoint of line2
elseif constraint == m_meets_L2
    num = num + getDistanceToLine(line2(2,:), line1);
    %penalty equals the distance between line1 and
    %the second endpoint of line2
elseif constraint == m_meets_1L
    num = num + getDistanceToLine(line1(1,:), line2);
    %penalty equals the distance between the first
    %endpoint of line1 and line2
elseif constraint == m_meets_2L
    num = num + getDistanceToLine(line1(2,:), line2);
    %penalty equals the distance between the second
```

```
        %endpoint of line1 and line2

    elseif constraint == m_equalAngle
        num = num + abs(abs(getLineAngle(line1) -
          getLineAngle(line2)) - ...
          abs(getLineAngle(line3) - getLineAngle(line4)));
        %penalty equals the difference between the
        %difference between the two angles

    else
        s = ['error can not find ' constraint]
    end
    if debug
      m_mapping(row(1))
      num
    end
end

num;
```

D.3 Constraint Mapping

This file is not a function; it simply assigns numerical values to variables. This also creates a vector m_mapping that contains a listing of all of the string constraints so that it is easy to go back to the string constraint when necessary. (This is very helpful in debugging.)

D.3.1 Code

```
%mapping file

global m_horizontal m_vertical m_posSlope m_negSlope ...
  m_near m_far m_intersects m_bisects_cc m_bisects_c1 ...
  m_bisects_c2 m_bisects_1c m_bisects_2c m_connected_11 ...
  m_connected_12 m_connected_21 m_connected_22 ...
  m_bisects_Lc m_bisects_cL m_meets_L1 m_meets_L2 ...
  m_meets_1L m_meets_2L m_sameX m_leftOf m_rightOf ...
  m_overlapLeftOf m_overlapRightOf m_sameY m_above ...
  m_below m_overlapAbove m_overlapBelow m_parallel ...
  m_perpendicular m_slanted m_acuteMeet m_obtuseMeet ...
  m_equalArea m_larger m_smaller m_equalAngle m_mapping

m_horizontal = 1;
m_vertical =2;
m_posSlope =3;
m_negSlope =4;

m_intersects =7;
m_bisects_cc =8;
m_bisects_c1 =9;
m_bisects_c2 =10;
m_bisects_1c =11;
m_bisects_2c =12;
m_connected_11 =13;
m_connected_12 =14;
m_connected_21 =15;
m_connected_22 =16;
```

```
m_bisects_Lc =17;
m_bisects_cL =18;
m_meets_L1 =19;
m_meets_L2 =20;
m_meets_1L =21;
m_meets_2L = 22;
m_near =5;
m_far =6;

m_parallel =33;
m_perpendicular =34;
m_slanted =35;
m_acuteMeet =36;
m_obtuseMeet =37;

m_sameX =23;
m_leftOf =24;
m_rightOf =25;
m_overlapLeftOf =26;
m_overlapRightOf = 27;
m_sameY =28;
m_above =29;
m_below =30;
m_overlapAbove =31;
m_overlapBelow =32;

m_equalArea =38;
m_larger =39;
m_smaller =40;
m_equalAngle =41;
```

```
m_mapping{m_horizontal}='horizontal';
m_mapping{m_vertical}='vertical';
m_mapping{m_posSlope}='posSlope';
m_mapping{m_negSlope}='negSlope';
m_mapping{m_near}='near';
m_mapping{m_far}='far';
m_mapping{m_intersects}='intersects';
m_mapping{m_bisects_cc}='bisects_cc';
m_mapping{m_bisects_c1}='bisects_c1';
m_mapping{m_bisects_c2}='bisects_c2';
m_mapping{m_bisects_1c}='bisects_1c';
m_mapping{m_bisects_2c}='bisects_2c';
m_mapping{m_connected_11}='connected_11';
m_mapping{m_connected_12}='connected_12';
m_mapping{m_connected_21}='connected_21';
m_mapping{m_connected_22}='connected_22';
m_mapping{m_bisects_Lc}='bisects_Lc';
m_mapping{m_bisects_cL}='bisects_cL';
m_mapping{m_meets_L1}='meets_L1';
m_mapping{m_meets_L2}='meets_L2';
m_mapping{m_meets_1L}='meets_1L';
m_mapping{m_meets_2L}='meets_2L';
m_mapping{m_sameX}='sameX';
m_mapping{m_leftOf}='leftOf';
m_mapping{m_rightOf}='rightOf';
m_mapping{m_overlapLeftOf}='overlapLeftOf';
m_mapping{m_overlapRightOf}='overlapRightOf';
m_mapping{m_sameY}='sameY';
m_mapping{m_above}='above';
```

```
m_mapping{m_below}='below';
m_mapping{m_overlapAbove}='overlapAbove';
m_mapping{m_overlapBelow}='overlapBelow';
m_mapping{m_parallel}='parallel';
m_mapping{m_perpendicular}='perpendicular';
m_mapping{m_slanted}='slanted';
m_mapping{m_acuteMeet}='acuteMeet';
m_mapping{m_obtuseMeet}='obtuseMeet';
m_mapping{m_equalArea}='equalArea';
m_mapping{m_larger}='larger';
m_mapping{m_smaller}='smaller';
m_mapping{m_equalAngle}='equalAngle';
```

D.4 Constraint Function: posSlope

This function returns zero if the angle is between 15 and 75 degrees (or .26 and 1.3 radians). Else, it returns an error relative to the distance from 45 degrees.

D.4.1 Code

```
function error = posSlope(line1)

angle = getLineAngle(line1);

if angle > .26 && angle < 1.3
    error = 0;
else
    l1x1 = line1(1,1);
    l1y1 = line1(1,2);
    l1x2 = line1(2,1);
    l1y2 = line1(2,2);
    error = abs((l1x2 - l1x1) - (l1y2 - l1y1));
end
```

D.5 Constraint Function: negSlope

This function returns zero if the angle is between -15 and -75 degrees (or $-.26$ and -1.3 radians). Else, it returns an error relative to the distance from -45 degrees.

D.5.1 Code

```
function error = negSlope(line1)

angle = getLineAngle(line1);

if angle > -1.3 && angle < -.26
    error = 0;
else
   error = abs(angle - .76)*10;
end
```

D.6 Constraint Function: larger

This function returns 0 if LINE1 is more than twice the length of the LINE2, else, it returns a penalty stating how much longer the LINE1 must be to be twice the length of the LINE2.

D.6.1 Code

```
function error = larger(line1, line2)


len1 = getLineLength(line1);
len2 = getLineLength(line2);


if len1 >  2 * len2
    error = 0;
else
    error = 2*getLineLength(line2) - getLineLength(line1);
end
```

D.7 Constraint Function: leftOf

If all of LINE1 is to the left of all of LINE2, this function returns 0. Otherwise, this function returns the difference between the maximum x-value of LINE1 and the minimum x-value of LINE2.

D.7.1 Code

```
function error = leftOf(line1, line2)
line1Max = getLineMaxX(line1);
line2Min = getLineMinX(line2);
error = max(0,line1Max - line2Min);
```

D.8 Constraint Function: above

If all of LINE1 is above all of LINE2, this function returns 0. Otherwise, this function returns the difference between the maximum y-value of LINE1 and the minimum y-value of LINE2.

D.8.1 Code

```
function error = above(line1, line2)
line1Min = getLineMinY(line1);
line2Max = getLineMaxY(line2);
error = max(0,line2Max - line1Min);
```

D.9 Constraint Function: overlapLeftOf

If the center of LINE1 is to the right of the center of LINE2, then this returns the difference between the x-values of the two centers. If this LINE1 is completely to the left of LINE2, (i.e., their bounding boxes do not overlap), then this function returns the difference between the maximum x-value of LINE1 and the minimum x-value of LINE2. Otherwise, this function returns 0 because the two x-values overlap, but the center of LINE1 is to the left of the center of LINE2.

D.9.1 Code

```
function error = overlapLeftOf(line1, line2)

point1 = getLineMidpoint(line1);
point2 = getLineMidpoint(line2);
```

```
leftCenter = point1(1);
rightCenter = point2(1);
leftMax = getLineMaxX(line1);
rightMin = getLineMinX(line2);

if leftMax < rightMin
    error = rightMin - leftMax;
elseif rightCenter < leftCenter
    error = leftCenter - rightCenter;
else
    error = 0;
end
```

D.10 Constraint Function: overlapAbove

If the center of LINE1 is above the center of LINE2, then this returns the difference between the y-values of the two centers. If this LINE1 is completely above LINE2, (i.e., their bounding boxes do not overlap), then this function returns the difference between the maximum y-value of LINE1 and the minimum y-value of LINE2. Otherwise, this function returns 0 because the two y-values overlap, but the center of LINE1 is above the center of LINE2.

D.10.1 Code

```
function error = overlapAbove(line1, line2)

point1 = getLineMidpoint(line1);
point2 = getLineMidpoint(line2);
```

```
topCenter = point1(2);
bottomCenter =point2(2);
topMax = getLineMaxY(line1);
bottomMin = getLineMinY(line2);

if topMax < bottomMin
    error = bottomMin - topMax;
elseif topCenter < bottomCenter
    error = bottomCenter - topCenter;
else
    error = 0;
end
```

D.11 Constraint Function: perpendicular

This function returns an error value based on how far away the two lines are from perpendicular. It uses the equality, $m1 == -1/m2$, which implies that $dy1/dx1 == -dx2/dy2$, to compute the error.

D.11.1 Code

```
function error = perpendicular(line1, line2)
%m1 = -1/m2
%dy1/dx1 = -1/(dy2/dx2)
%dy1/dx1 = -dx2/dy2
%dy1*dy2 = -dx2*dx1;
[dx1, dy1] = getDirectionVector(line1);
[dx2, dy2] = getDirectionVector(line2);
error = (dx2*dx1 + dy2*dy1)^2;
```

D.12 Constraint Function: acuteMeet

This function finds the distance between each of the endpoints. It finds the distance between the two closest endpoints. This value is part of the returned error penalty. It then compute how far away the other two points are. The distance between the other two points should be less than the length of the longest line, as the three points of an acute angle form a triangle. The function also ensures that the angle is not 0 degrees, such that the two lines lie flat on each other. In this case, the function requires that the sum of the shortest lines of the triangle formed by the points acute angle are longer than the longest line of this triangle.

D.12.1 Code

```
function error = acuteMeet(line1, line2)

distance11 = getLineLength([line1(1,:); line2(1,:)]);
distance12 = getLineLength([line1(1,:); line2(2,:)]);
distance21 = getLineLength([line1(2,:); line2(1,:)]);
distance22 = getLineLength([line1(2,:); line2(2,:)]);

distance = getLineEndpointDistance(line1, line2);
%distanceLong = getLongEndpointDistance(line1, line2);
maxlen = max(getLineLength(line1),getLineLength(line2));
minlen = min(getLineLength(line1),getLineLength(line2));

if distance == distance11
    otherdist = distance22;
elseif distance == distance12
    otherdist = distance21;
elseif distance == distance21
```

```
    otherdist = distance12;
else
    otherdist = distance11;
end


if otherdist + minlen < maxlen * 1.1
    error = maxlen * 1.1 - otherdist - minlen;
elseif otherdist > maxlen + 1;
    error = otherdist - maxlen - 1;
else
    error = 0;
end


error = error + distance;
```

D.13 Constraint Function: obtuseMeet

This function finds the distance between each of the endpoints. It finds the distance between the two closest endpoints. This value is part of the returned error penalty. It then compute how far away the other two points are. In the case of the triangle formed by an obtuse angle, the longest line of the triangle is abstract line connecting the two endpoints of the angle. This function ensures that distance between the two endpoints is longer than the maximum of the two input lines, and returns an appropriate error penalty if they are not. This function also ensures that the distance between the two lines is not equal to the sum of the two lines, in which case the two lines would be flat, and it returns the appropriate penalty if they are.

D.13.1 Code

```
function error = obtuseMeet(line1, line2)

distance11 = getLineLength([line1(1,:); line2(1,:)]);
distance12 = getLineLength([line1(1,:); line2(2,:)]);
distance21 = getLineLength([line1(2,:); line2(1,:)]);
distance22 = getLineLength([line1(2,:); line2(2,:)]);

distance = getLineEndpointDistance(line1, line2);
%distanceLong = getLongEndpointDistance(line1, line2);
maxlen = getLineLength(line1) + getLineLength(line2);
minlen = sqrt(getLineLength(line1)^2 +
              getLineLength(line2)^2);

if distance == distance11
    otherdist = distance22;
elseif distance == distance12
    otherdist = distance21;
elseif distance == distance21
    otherdist = distance12;
else
    otherdist = distance11;
end

if otherdist < minlen * 1.1
    error = minlen*1.1 - otherdist;
elseif otherdist > maxlen * .9
    error = otherdist - maxlen*.9;
else
```

```
    error = 0;
end

error = error + distance;
```

D.14 Constraint Function: slanted

This function computes the angle between the two lines modulus 90 degrees. If the angle is between 15 and 75 degrees, it returns 0, else it returns the distance to 45 degrees.

D.14.1 Code

```
function error = slanted(line1, line2)

angle1 = getLineAngle(line1);
angle2 = getLineAngle(line2);

dif = angle1 - angle2;
mpi = mod(dif, pi/2);

if mpi > .38 && mpi < 1.18
    error = 0;
else
    error = mod(dif, .78);
end

error = error + notConnected(line1, line2);
```

D.15 Constraint Function: near

In order for two lines to be considered near, they must be at least 10 pixels apart, and at least a quarter of the length of the shortest line away from each other. They must also be a maximum of 40 pixels apart, and the distance between them must be less than the length of the shortest line. If within these boundaries, the function returns 0; else, the function returns the distance to these boundaries.

D.15.1 Code

```
function error = near(line1, line2)

distance = getLineDistance(line1, line2);
len1 = getLineLength(line1);
len2 = getLineLength(line2);
mindistance = max(10, min(len1/4,len2/4));
maxdistance = max(40, min(len1, len2));

if distance < mindistance
    error = mindistance - distance;
elseif distance > maxdistance;
    error = distance - maxdistance;
else
    error = 0;
end
```

D.16 Constraint Function: far

In order for two lines to be considered far apart, the distance between them must be greater than 40 pixels, or it must be greater than the length of the minimum line length. If this is not the case, than the system returns the distance from these boundaries.

D.16.1 Code

```
function error = far(line1, line2)

distance = getLineDistance(line1, line2);
mindistance = max(40, min(getLineLength(line1),
                          getLineLength(line2)));

if distance > mindistance
    error = 0;
else
    error = mindistance - distance;
end
```

D.17 Helper Function: translateConstraints

This function translates a vector of string constraints, such as PARALLEL 1 2 (where 1 and 2 represent the order of the incoming lines), into a matrix of numerical constraints. Each constraint has a numerical value, and each line of the returned numerical constraints contains the value and the argument numbers (e.g., 33 1 2).

D.17.1 Inputs

The input is the matrix of string constraints. Each line of the matrix is a vector representing one constraint. The vector representing the constraint contains the constraint, followed by the argument numbers. For example, a line could consist of [PARALLEL 1 2] which means that the lines 1 and 2 (in the order they are listed in the X and X0 vectors) are parallel.

D.17.2 Outputs

The output is a matrix of numerical constraints. Each constraint string is translated to its numerical representation as specified by the MAPPING function. For example, in the mapping function, PARALLEL is represented by the number 33, and thus the line in the input matrix displaying [PARALLEL 1 2] will be translated to [33 1 2].

D.17.3 Explanation of Internals

The MAPPING function is first called to set all of the internal numerical values for each of the constraints. Then each string constraint name is replaced one by one with the numerical value representing that constraint.

D.17.4 Code

```
function constraints = translateConstraints(strConstraints)

mapping;

global m_horizontal m_vertical m_posSlope m_negSlope ...
  m_near m_far m_intersects m_bisects_cc ...
  m_bisects_c1 m_bisects_c2 m_bisects_1c ...
  m_bisects_2c m_connected_11 m_connected_12 ...
  m_connected_21 m_connected_22 m_bisects_Lc m_bisects_cL ...
  m_meets_L1 m_meets_L2  m_meets_1L m_meets_2L m_sameX ...
  m_leftOf m_rightOf  m_overlapLeftOf m_overlapRightOf ...
  m_sameY m_above m_below m_overlapAbove m_overlapBelow ...
  m_parallel m_perpendicular m_slanted m_acuteMeet  ...
  m_obtuseMeet m_equalArea m_larger m_smaller ...
  m_equalAngle m_mapping

[a,b] = size(strConstraints); for i = 1 : a
    constraints(i,1) = 0;
    row = strConstraints(i,:)
    for c = 1 : length(m_mapping)
        if  strcmp(row(1), m_mapping(c))
            constraints(i,1) = c;
        end
    end
    if constraints(i,1) == 0
        ['ERROR! ' row(1) ' not found!']
    end
    constraints(i,2) = row{2};
    constraints(i,3) = row{3};
end
```

D.18 Helper Function: makeLineMatrix

This function takes makes a line matrix for easy access to each of the lines. It changes x0 from a vector of x- and y-values to a matrix of lines that specify the index of the x solution vector that the value pertains to. For example, [X1 Y1 X2 Y2 X3 Y3 X4 Y4] is changed to [1 2 3 4, 5 6 7 8]. This may seem trivial, but is important when other values are included in the matrix, such as circles, which contain a different number of properties, and other constraints.

D.18.1 Input

This function takes, as input, X0, a vector of the initial values. X0 is a vector of double values representing each of the x- and y-values of each of the lines; e.g., for a single lined shape, X0 = [X1 Y1 X2 Y2]. For a two-lined shape, X0 = [X1 Y1 X2 Y2 X3 Y3 X4 Y4], where X1 and X2 represent the endpoints of the first line, and X3 and X4 represent the endpoints of the second line. Multiple lines are represented similarly.

D.18.2 Output

This function returns, as output, a line matrix representing the index of the line x or y-value in the X solution vector. For example, [X1 Y1 X2 Y2 X3 Y3 X4 Y4] is changed to [1 2 3 4; 5 6 7 8].

D.18.3 Code

```
function lineMatrix = makeLineMatrix(x0)

lineMatrix = 0;
count = 1;
for i = 1 : length(x0) / 4
  lineMatrix(i, 1) = count;
  count = count+ 1;
  lineMatrix(i, 2) = count;
  count = count + 1;
  lineMatrix(i, 3) = count;
  count = count + 1;
  lineMatrix(i, 4) = count;
  count = count + 1;
end
```

D.19 Helper Function: lineToPoints

This function converts a vector of point values of a line, such as [1 2 3 4], to a matrix representation of a line, with the points separated, such as [1 2; 3 4].

D.19.1 Code

```
function points = lineToPoints(line)
points(1,:) = line(1:2);
points(2,:) = line(3:4);
```

D.20 Helper Function: getLineLength

This function computes the length of a line of the form [X1 Y1 ; X2 Y2].

D.20.1 Code

```
function l = getLineLength(line)

x1 = line(1,1);
```

```
y1 = line(1,2);
x2 = line(2,1);
y2 = line(2,2);

l = norm([x2-x1, y2-y1]);
```

D.21 Helper Function: getDistanceToLine

This function takes in a point and a line and computes the distance between the point and the line segment.

D.21.1 Explanation of Internals

1. The function checks if the point is on the line segment, using isPointOnLine. If it is, the function returns a distance of 0.
2. The function checks if the point is on the line extended to infinity, using distToHPlane. If so, it returns the distance to the closest endpoint.
3. It create a line that is perpendicular to this line that passes through the point, using getPerpendicularLine.
4. It converts both the original line and the perpendicular line to the form of $Ax + By = C$.
5. It finds the intersection point of the original line and the perpendicular line, using getLineAxByC.
6. If the intersection point is on the line segment, it returns the distance from the point to the intersection point.
7. Otherwise, it returns the distance from the point to the closest endpoint.

D.21.2 Code

```
function theDistance = getDistanceToLine(point, line)

if isPointOnLine(point, line)
    theDistance = 0;
else
  if  distToHPlane(line, point) < .001
    dist(1) = getLineLength([line(1,:); point]);
    dist(2) = getLineLength([line(2,:); point]);
    theDistance = min(dist);
  else
    if length(line) == 3
      array1 = line;
    else
      array1 = getLineAxByC(line);
    end
    perpline = getPerpendicularLine(line, point);
    array2 = getLineAxByC(perpline);

    A = [array1(1), array1(2); array2(1), array2(2)];
    b = [array1(3); array2(3)];

    intersectsPoint = linsolve(A, b)';
    if isPointOnLine(intersectsPoint, line)
       theDistance = getLineLength([intersectsPoint; point]);
    else
        dist(1) = getLineLength([line(1,:); point]);
        dist(2) = getLineLength([line(2,:); point]);
        theDistance = min(dist);
    end
  end
end
```

D.22 Helper Function: isPointOnLine

This function checks if the point is on the line segment. It returns 1 if the point is on the line segment; otherwise, it returns 0.

D.22.1 Explanation of Internals

1. The function checks to see if the point is within the bounding box of the line. If not, the function returns 0.
2. The function computes the distance from the point to the line extended to infinity using distToHPlane. If the distance is 0, the point is on the line, and the function returns 1. Otherwise, the function returns 0.

D.22.2 Code

```
function bool = isPointOnLine(point, line)

x = point(1);
y = point(2);

x1 = line(1,1);
y1 = line(1,2);
x2 = line(2,1);
y2 = line(2,2);
```

```
if x > x1 && x > x2

    bool = 0;

elseif x < x1 && x < x2

    bool = 0;

elseif y > y1 && y > y2

    bool = 0;

elseif y < y1 && y < y2

    bool = 0;

elseif distToHPlane(line, point) < .001

    bool = 1;

else

    bool = 0;

end
```

D.23 Helper Function: distToHPlane

This function finds the distance from a point, z, to a hyperplane, hplanePts.

D.23.1 Code

```
function theDistance = distToHPlane(hplanePts,z)

%this function assumes the hyperplane is given by hplanePts

%you want to find the distance from point z to this

%hyperplane


%form hplane equation a'x=k

a = computeNormal(hplanePts);
```

```
k = a'*hplanePts(1,:)'; %use any point on hplane to get rhs k

if norm(a) == 0
    theDistance = Inf;
else
    theDistance = abs((dot(a,z)-k)/norm(a));
end
```

D.24 Helper Function: getLineABCArray

The function takes a line and puts it into the form $[abc]$, where $ax + by = c$ represents the formula of the line.

D.24.1 Code

```
function array = getLineAxByC(line)
% puts the formula into the form [a b c], where ax + by = c
% y = mx + b => - mx + y = b ...

x1 = line(1,1);
y1 = line(1,2);
x2 = line(2,1);
y2 = line(2,2);

if abs(x2-x1) < .001
    y_val = 0;
    x_val = 1;
    c_val = x1;
elseif abs(y2-y1) < .001
```

```
    y_val = 1;
    x_val = 0;
    c_val = y1;
else
    y_val = 1;
    x_val = -getLineSlope(line);
    c_val = getLineYIntercept(line);
end

array = [x_val, y_val, c_val];
```

D.25 Helper Function: getPerpendicularLine

This function computes the perpendicular line, PERPLINE, perpendicular to the input LINE and passing through a POINT. The output line, PERPLINE, is the same length as the input line, LINE.

D.25.1 Code

```
function perpline = getPerpendicularLine(line, point)

if length(line) == 3
  line = ABCLineToSegment(line);
end
angle = getLineAngle(line);
len = getLineLength(line);
newangle = angle + pi/2;

p1 = point;
```

```
p2 = [p1(1) + cos(newangle) * len, p1(2)
           + sin(newangle) * len];
perpline = [p1;p2];
```

D.26 Helper Function: getLineDistance

This function computes the distance between the two lines. If the two lines intersect, then the distance is returned as zero. Otherwise, the distance from each endpoint to the other line is computed, and the shortest distance is returned.

D.26.1 Code

```
function theDistance = getLineDistance(line1, line2)

if isLineIntersecting(line1, line2)
    theDistance = 0;
else
    dist(1) = getDistanceToLine(line2(1,:), line1);
    dist(2) = getDistanceToLine(line2(2,:), line1);
    dist(3) = getDistanceToLine(line1(1,:), line2);
    dist(4) = getDistanceToLine(line1(2,:), line2);
    theDistance = min(dist);
end
```

D.27 Helper Function: isLineIntersecting

This function determines if two line segments are intersecting. It does this by first determining if the two lines overlap (i.e., lie on top of each other with the same slope).

If they do not, if finds out if there exists an intersection point that lies on both line segments.

D.27.1 Code

```
function bool = isLineIntersecting(line1, line2)


if isLineOverlapping(line1, line2)
    bool = 1;
else
    point = getInfLineIntersectingPoint(line1, line2);
    if isPointOnLine(point, line1) &&
       isPointOnLine(point, line2)
        bool = 1;
    else
        bool = 0;
    end
end
```

D.28 Helper Function: getInfLineIntersectingPoint

This function computes the intersection point of the two lines when extended to infinity. It does this by converting both lines to the form $Ax + By = C$, and solving the linear equation to find the point (x, y) that solves both equations.

D.28.1 Code

```
function point = getInfLineIntersectingPoint(line1, line2)
```

```
if length(line1) == 3
  array1 = line1;
else
  array1 = getLineAxByC(line1);
end
if length(line2) == 3
  array2 = line2;
else
  array2 = getLineAxByC(line2);
end

A = [array1(1), array1(2); array2(1), array2(2)];
b = [array1(3); array2(3)];

[X, R] = linsolve(A, b);
point = X';
```

D.29 Helper Function: isLineOverlapping

This function determines if two lines are overlapping, i.e., they intersect and share the same slope. We check for this condition separately since this case causes difficulties when solving linear constraints.

D.29.1 Code

```
function bool = isLineOverlapping(line1, line2)

if isInfLineOverlapping(line1, line2) == 0
    bool = 0;
```

```
elseif isLineBoundingBoxOverlapping(line1, line2)
    bool = 1;
else
    bool = 0;
end
```

D.30 Helper Function: isInfLineOverlapping

This function checks to see if the two lines when extended to infinity overlapping, i.e., they have the same slope and y-intercept.

D.30.1 Code

```
function bool = isInfLineOverlapping(line1, line2)

point = getInfLineIntersectingPoint(line1, line2);
if isnan(point(1,1)) %&& isnan(point(1,2))
    bool = 1;
else
    bool = 0;
end
```

D.31 Helper Function: isLineBoundingBoxOverlapping

This function checks that the bounding boxes of the two lines do not overlap.

D.31.1 Code

```
function bool = isLineBoundingBoxOverlapping(line1, line2)

if getLineMinX(line1) > getLineMaxX(line2)
    bool = 0;
elseif getLineMaxX(line1) < getLineMinX(line2)
    bool = 0;
elseif getLineMinY(line1) > getLineMaxY(line2)
    bool = 0;
elseif getLineMaxY(line1) < getLineMinY(line2)
    bool = 0;
else
    bool = 1;
end
```

D.32 Helper Function: getLineAngle

This function returns the angle of the line in radians, returning a value between 0 and $2 * pi$.

D.32.1 Code

```
function angle = getLineAngle(line)

x1 = line(1,1);
y1 = line(1,2);
x2 = line(2,1);
y2 = line(2,2);
```

```
%soh cah toa
angle = atan2((y2-y1),(x2-x1));
```

D.33 Helper Function: getLineSlope

This function computes the slope of a line.

```
function m = getLineSlope(line)
%computes the slope of a line.

x1 = line(1,1);
y1 = line(1,2);
x2 = line(2,1);
y2 = line(2,2);

m = (y2-y1)/(x2-x1);
```

D.34 Helper Function: getLineYIntercept

This function returns the y-intercept of a line.

D.34.1 Code

```
function b = getLineYIntercept(line)
% returns the yIntercept b
% y = mx + b
% b = y - mx
% if line is vertical, returns NaN
```

```
x1 = line(1,1);
y1 = line(1,2);

if isLineVertical(line)
    b = NaN;
else
    b = y1 - getLineSlope(line) * x1;
end
```

Appendix E

Indexing Data

E.1 Indexing of a Line

This section shows the indexing data after a line and all related created data have been indexed. (Point data lists x, y, and time.)

E.1.1 Name Index

This lists all of the possible components for each shape name.

```
Shape Names:
  line:
    RLine line8 RPoint p1 (177, 408, 1170046781488)
                RPoint p2 (277, 323, 1170046781706)
    RLine line11 RPoint p1 (277, 323, 1170046781706)
                 RPoint p2 (177, 408, 1170046781488)
Subshape Names:
  boundRight:
    RLine line32 RPoint p1 (277, 323, 1170046782831)
```

```
                RPoint p2 (277, 408, 1170046782831)
  RLine line81 RPoint p1 (277, 323, 1170046782831)
               RPoint p2 (277, 408, 1170046782831)
boundTopRight:
  RPoint point36 (277, 323, 1170046782831)
  RPoint point85 (277, 323, 1170046782846)
p2:
  RPoint p2 (277, 323, 1170046781706)
  RPoint p2 (177, 408, 1170046781488)
boundBottomRight:
  RPoint point38 (277, 408, 1170046782831)
  RPoint point87 (277, 408, 1170046782846)
p1:
  RPoint p1 (177, 408, 1170046781488)
  RPoint p1 (277, 323, 1170046781706)
boundTop:
  RLine line17 RPoint p1 (177, 323, 1170046782815)
               RPoint p2 (277, 323, 1170046782815)
  RLine line66 RPoint p1 (177, 323, 1170046782831)
               RPoint p2 (277, 323, 1170046782831)
boundBottomMiddle:
  RPoint point50 (227, 408, 1170046782831)
  RPoint point99 (227, 408, 1170046782846)
boundTopMiddle:
  RPoint point44 (227, 323, 1170046782831)
  RPoint point93 (227, 323, 1170046782846)
boundLeft:
  RLine line27 RPoint p1 (177, 323, 1170046782831)
               RPoint p2 (177, 408, 1170046782831)
  RLine line76 RPoint p1 (177, 323, 1170046782831)
```

```
                RPoint p2 (177, 408, 1170046782831)
  boundRightMiddle:
    RPoint point62 (277, 365, 1170046782831)
    RPoint point111 (277, 365, 1170046782846)
  boundBottomLeft:
    RPoint point37 (177, 408, 1170046782831)
    RPoint point86 (177, 408, 1170046782846)
  center:
    RPoint point14 (227, 365, 1170046782815)
    RPoint point63 (227, 365, 1170046782815)
  boundLeftMiddle:
    RPoint point56 (177, 365, 1170046782831)
    RPoint point105 (177, 365, 1170046782846)
  boundBottom:
    RLine line22 RPoint p1 (177, 408, 1170046782831)
                 RPoint p2 (277, 408, 1170046782831)
    RLine line71 RPoint p1 (177, 408, 1170046782831)
                 RPoint p2 (277, 408, 1170046782831)
  boundTopLeft:
    RPoint point35 (177, 323, 1170046782831)
    RPoint point84 (177, 323, 1170046782846)
pos:
  RLine line8 RPoint p1 (177, 408, 1170046781488)
              RPoint p2 (277, 323, 1170046781706)
  RLine line11 RPoint p1 (277, 323, 1170046781706)
               RPoint p2 (177, 408, 1170046781488)
neg: hor: ver: angle:
  40.0:
    RLine line8 RPoint p1 (177, 408, 1170046781488)
                RPoint p2 (277, 323, 1170046781706)
    RLine line11 RPoint p1 (277, 323, 1170046781706)
                 RPoint p2 (177, 408, 1170046781488)
```

E.1.2 Type Index

This lists all of the accessible components for each type.

```
Main Types: Total = 4
  LAC:
    RLine line8 RPoint p1 (177, 408, 1170046781488)
                RPoint p2 (277, 323, 1170046781706)
    RLine line11 RPoint p1 (277, 323, 1170046781706)
                 RPoint p2 (177, 408, 1170046781488)
  Shape:
    RLine line8 RPoint p1 (177, 408, 1170046781488)
                RPoint p2 (277, 323, 1170046781706)
    RLine line11 RPoint p1 (277, 323, 1170046781706)
                 RPoint p2 (177, 408, 1170046781488)
  DrawnShape:
    RLine line8 RPoint p1 (177, 408, 1170046781488)
                RPoint p2 (277, 323, 1170046781706)
    RLine line11 RPoint p1 (277, 323, 1170046781706)
                 RPoint p2 (177, 408, 1170046781488)
  Line:
    RLine line8 RPoint p1 (177, 408, 1170046781488)
                RPoint p2 (277, 323, 1170046781706)
    RLine line11 RPoint p1 (277, 323, 1170046781706)
                 RPoint p2 (177, 408, 1170046781488)
Subshape Types: Total = 5
  LAC:
```

```
  RLine line17 RPoint p1 (177, 323, 1170046782815)
               RPoint p2 (277, 323, 1170046782815)
  RLine line22 RPoint p1 (177, 408, 1170046782831)
               RPoint p2 (277, 408, 1170046782831)
  RLine line27 RPoint p1 (177, 323, 1170046782831)
               RPoint p2 (177, 408, 1170046782831)
  RLine line32 RPoint p1 (277, 323, 1170046782831)
               RPoint p2 (277, 408, 1170046782831)
  RLine line66 RPoint p1 (177, 323, 1170046782831)
               RPoint p2 (277, 323, 1170046782831)
  RLine line71 RPoint p1 (177, 408, 1170046782831)
               RPoint p2 (277, 408, 1170046782831)
  RLine line76 RPoint p1 (177, 323, 1170046782831)
               RPoint p2 (177, 408, 1170046782831)
  RLine line81 RPoint p1 (277, 323, 1170046782831)
               RPoint p2 (277, 408, 1170046782831)
Shape:
  RPoint p1 (177, 408, 1170046781488)
  RPoint p2 (277, 323, 1170046781706)
  RPoint point14 (227, 365, 1170046782815)
  RLine line17 RPoint p1 (177, 323, 1170046782815)
               RPoint p2 (277, 323, 1170046782815)
  RLine line22 RPoint p1 (177, 408, 1170046782831)
               RPoint p2 (277, 408, 1170046782831)
  RLine line27 RPoint p1 (177, 323, 1170046782831)
               RPoint p2 (177, 408, 1170046782831)
  RLine line32 RPoint p1 (277, 323, 1170046782831)
               RPoint p2 (277, 408, 1170046782831)
  RPoint point35 (177, 323, 1170046782831)
  RPoint point36 (277, 323, 1170046782831)
  RPoint point37 (177, 408, 1170046782831)
  RPoint point38 (277, 408, 1170046782831)
  RPoint point44 (227, 323, 1170046782831)
  RPoint point50 (227, 408, 1170046782831)
  RPoint point56 (177, 365, 1170046782831)
  RPoint point62 (277, 365, 1170046782831)
  RPoint p1 (277, 323, 1170046781706)
  RPoint p2 (177, 408, 1170046781488)
  RPoint point63 (227, 365, 1170046782815)
  RLine line66 RPoint p1 (177, 323, 1170046782831)
               RPoint p2 (277, 323, 1170046782831)
  RLine line71 RPoint p1 (177, 408, 1170046782831)
               RPoint p2 (277, 408, 1170046782831)
  RLine line76 RPoint p1 (177, 323, 1170046782831)
               RPoint p2 (177, 408, 1170046782831)
  RLine line81 RPoint p1 (277, 323, 1170046782831)
               RPoint p2 (277, 408, 1170046782831)
  RPoint point84 (177, 323, 1170046782846)
  RPoint point85 (277, 323, 1170046782846)
  RPoint point86 (177, 408, 1170046782846)
  RPoint point87 (277, 408, 1170046782846)
  RPoint point93 (227, 323, 1170046782846)
  RPoint point99 (227, 408, 1170046782846)
  RPoint point105 (177, 365, 1170046782846)
  RPoint point111 (277, 365, 1170046782846)
DrawnShape:
  RPoint p1 (177, 408, 1170046781488)
  RPoint p2 (277, 323, 1170046781706)
  RPoint point14 (227, 365, 1170046782815)
  RLine line17 RPoint p1 (177, 323, 1170046782815)
```

```
             RPoint p2 (277, 323, 1170046782815)
RLine line22 RPoint p1 (177, 408, 1170046782831)
             RPoint p2 (277, 408, 1170046782831)
RLine line27 RPoint p1 (177, 323, 1170046782831)
             RPoint p2 (177, 408, 1170046782831)
RLine line32 RPoint p1 (277, 323, 1170046782831)
             RPoint p2 (277, 408, 1170046782831)
RPoint point35 (177, 323, 1170046782831)
RPoint point36 (277, 323, 1170046782831)
RPoint point37 (177, 408, 1170046782831)
RPoint point38 (277, 408, 1170046782831)
RPoint point44 (227, 323, 1170046782831)
RPoint point50 (227, 408, 1170046782831)
RPoint point56 (177, 365, 1170046782831)
RPoint point62 (277, 365, 1170046782831)
RPoint p1 (277, 323, 1170046781706)
RPoint p2 (177, 408, 1170046781488)
RPoint point63 (227, 365, 1170046782815)
RLine line66 RPoint p1 (177, 323, 1170046782831)
             RPoint p2 (277, 323, 1170046782831)
RLine line71 RPoint p1 (177, 408, 1170046782831)
             RPoint p2 (277, 408, 1170046782831)
RLine line76 RPoint p1 (177, 323, 1170046782831)
             RPoint p2 (177, 408, 1170046782831)
RLine line81 RPoint p1 (277, 323, 1170046782831)
             RPoint p2 (277, 408, 1170046782831)
RPoint point84 (177, 323, 1170046782846)
RPoint point85 (277, 323, 1170046782846)
RPoint point86 (177, 408, 1170046782846)
RPoint point87 (277, 408, 1170046782846)
```

```
  RPoint point93 (227, 323, 1170046782846)
  RPoint point99 (227, 408, 1170046782846)
  RPoint point105 (177, 365, 1170046782846)
  RPoint point111 (277, 365, 1170046782846)
Line:
  RLine line17 RPoint p1 (177, 323, 1170046782815)
               RPoint p2 (277, 323, 1170046782815)
  RLine line22 RPoint p1 (177, 408, 1170046782831)
               RPoint p2 (277, 408, 1170046782831)
  RLine line27 RPoint p1 (177, 323, 1170046782831)
               RPoint p2 (177, 408, 1170046782831)
  RLine line32 RPoint p1 (277, 323, 1170046782831)
               RPoint p2 (277, 408, 1170046782831)
  RLine line66 RPoint p1 (177, 323, 1170046782831)
               RPoint p2 (277, 323, 1170046782831)
  RLine line71 RPoint p1 (177, 408, 1170046782831)
               RPoint p2 (277, 408, 1170046782831)
  RLine line76 RPoint p1 (177, 323, 1170046782831)
               RPoint p2 (177, 408, 1170046782831)
  RLine line81 RPoint p1 (277, 323, 1170046782831)
               RPoint p2 (277, 408, 1170046782831)
Point:
  RPoint p1 (177, 408, 1170046781488)
  RPoint p2 (277, 323, 1170046781706)
  RPoint point14 (227, 365, 1170046782815)
  RPoint point35 (177, 323, 1170046782831)
  RPoint point36 (277, 323, 1170046782831)
  RPoint point37 (177, 408, 1170046782831)
  RPoint point38 (277, 408, 1170046782831)
  RPoint point44 (227, 323, 1170046782831)
```

```
    RPoint point50 (227, 408, 1170046782831)
    RPoint point56 (177, 365, 1170046782831)
    RPoint point62 (277, 365, 1170046782831)
    RPoint p1 (277, 323, 1170046781706)
    RPoint p2 (177, 408, 1170046781488)
    RPoint point63 (227, 365, 1170046782815)
    RPoint point84 (177, 323, 1170046782846)
    RPoint point85 (277, 323, 1170046782846)
    RPoint point86 (177, 408, 1170046782846)
    RPoint point87 (277, 408, 1170046782846)
    RPoint point93 (227, 323, 1170046782846)
    RPoint point99 (227, 408, 1170046782846)
    RPoint point105 (177, 365, 1170046782846)
    RPoint point111 (277, 365, 1170046782846)
```

E.1.3 X Index

This section lists accessible values at particular x values.

```
x Shapes:
  227.0:
    RLine line8 RPoint p1 (177, 408, 1170046781488)
                RPoint p2 (277, 323, 1170046781706)
    RLine line11 RPoint p1 (277, 323, 1170046781706)
                 RPoint p2 (177, 408, 1170046781488)
x Subshapes:
  177.0:
    RPoint p1 (177, 408, 1170046781488)
    RLine line27 RPoint p1 (177, 323, 1170046782831)
                 RPoint p2 (177, 408, 1170046782831)
    RPoint point35 (177, 323, 1170046782831)
    RPoint point37 (177, 408, 1170046782831)
    RPoint point56 (177, 365, 1170046782831)
    RPoint p2 (177, 408, 1170046781488)
    RLine line76 RPoint p1 (177, 323, 1170046782831)
                 RPoint p2 (177, 408, 1170046782831)
    RPoint point84 (177, 323, 1170046782846)
    RPoint point86 (177, 408, 1170046782846)
    RPoint point105 (177, 365, 1170046782846)
  227.0:
    RPoint point14 (227, 365, 1170046782815)
    RLine line17 RPoint p1 (177, 323, 1170046782815)
                 RPoint p2 (277, 323, 1170046782815)
    RLine line22 RPoint p1 (177, 408, 1170046782831)
                 RPoint p2 (277, 408, 1170046782831)
    RPoint point44 (227, 323, 1170046782831)
    RPoint point50 (227, 408, 1170046782831)
    RPoint point63 (227, 365, 1170046782815)
    RLine line66 RPoint p1 (177, 323, 1170046782831)
                 RPoint p2 (277, 323, 1170046782831)
    RLine line71 RPoint p1 (177, 408, 1170046782831)
                 RPoint p2 (277, 408, 1170046782831)
    RPoint point93 (227, 323, 1170046782846)
    RPoint point99 (227, 408, 1170046782846)
  277.0:
    RPoint p2 (277, 323, 1170046781706)
    RLine line32 RPoint p1 (277, 323, 1170046782831)
                 RPoint p2 (277, 408, 1170046782831)
    RPoint point36 (277, 323, 1170046782831)
    RPoint point38 (277, 408, 1170046782831)
```

```
    RPoint point62 (277, 365, 1170046782831)
    RPoint p1 (277, 323, 1170046781706)
    RLine line81 RPoint p1 (277, 323, 1170046782831)
                 RPoint p2 (277, 408, 1170046782831)
    RPoint point85 (277, 323, 1170046782846)
    RPoint point87 (277, 408, 1170046782846)
    RPoint point111 (277, 365, 1170046782846)
```

E.1.4 Y Index

```
y Shapes:
  365.5:
    RLine line8 RPoint p1 (177, 408, 1170046781488)
                RPoint p2 (277, 323, 1170046781706)
    RLine line11 RPoint p1 (277, 323, 1170046781706)
                 RPoint p2 (177, 408, 1170046781488)
y Subshapes:
  323.0:
    RPoint p2 (277, 323, 1170046781706)
    RLine line17 RPoint p1 (177, 323, 1170046782815)
                 RPoint p2 (277, 323, 1170046782815)
    RPoint point35 (177, 323, 1170046782831)
    RPoint point36 (277, 323, 1170046782831)
    RPoint point44 (227, 323, 1170046782831)
    RPoint p1 (277, 323, 1170046781706)
    RLine line66 RPoint p1 (177, 323, 1170046782831)
                 RPoint p2 (277, 323, 1170046782831)
    RPoint point84 (177, 323, 1170046782846)
    RPoint point85 (277, 323, 1170046782846)
    RPoint point93 (227, 323, 1170046782846)
```

```
365.5:
  RPoint point14 (227, 365, 1170046782815)
  RLine line27 RPoint p1 (177, 323, 1170046782831)
               RPoint p2 (177, 408, 1170046782831)
  RLine line32 RPoint p1 (277, 323, 1170046782831)
               RPoint p2 (277, 408, 1170046782831)
  RPoint point56 (177, 365, 1170046782831)
  RPoint point62 (277, 365, 1170046782831)
  RPoint point63 (227, 365, 1170046782815)
  RLine line76 RPoint p1 (177, 323, 1170046782831)
               RPoint p2 (177, 408, 1170046782831)
  RLine line81 RPoint p1 (277, 323, 1170046782831)
               RPoint p2 (277, 408, 1170046782831)
  RPoint point105 (177, 365, 1170046782846)
  RPoint point111 (277, 365, 1170046782846)
408.0:
  RPoint p1 (177, 408, 1170046781488)
  RLine line22 RPoint p1 (177, 408, 1170046782831)
               RPoint p2 (277, 408, 1170046782831)
  RPoint point37 (177, 408, 1170046782831)
  RPoint point38 (277, 408, 1170046782831)
  RPoint point50 (227, 408, 1170046782831)
  RPoint p2 (177, 408, 1170046781488)
  RLine line71 RPoint p1 (177, 408, 1170046782831)
               RPoint p2 (277, 408, 1170046782831)
  RPoint point86 (177, 408, 1170046782846)
  RPoint point87 (277, 408, 1170046782846)
  RPoint point99 (227, 408, 1170046782846)
```

E.1.5 MinX Index

```
minX Shapes:
  177.0:
    RLine line8 RPoint p1 (177, 408, 1170046781488)
                RPoint p2 (277, 323, 1170046781706)
    RLine line11 RPoint p1 (277, 323, 1170046781706)
                 RPoint p2 (177, 408, 1170046781488)
minX Subshapes:
  177.0:
    RPoint p1 (177, 408, 1170046781488)
    RLine line17 RPoint p1 (177, 323, 1170046782815)
                 RPoint p2 (277, 323, 1170046782815)
    RLine line22 RPoint p1 (177, 408, 1170046782831)
                 RPoint p2 (277, 408, 1170046782831)
    RLine line27 RPoint p1 (177, 323, 1170046782831)
                 RPoint p2 (177, 408, 1170046782831)
    RPoint point35 (177, 323, 1170046782831)
    RPoint point37 (177, 408, 1170046782831)
    RPoint point56 (177, 365, 1170046782831)
    RPoint p2 (177, 408, 1170046781488)
    RLine line66 RPoint p1 (177, 323, 1170046782831)
                 RPoint p2 (277, 323, 1170046782831)
    RLine line71 RPoint p1 (177, 408, 1170046782831)
                 RPoint p2 (277, 408, 1170046782831)
    RLine line76 RPoint p1 (177, 323, 1170046782831)
                 RPoint p2 (177, 408, 1170046782831)
    RPoint point84 (177, 323, 1170046782846)
    RPoint point86 (177, 408, 1170046782846)
    RPoint point105 (177, 365, 1170046782846)
  227.0:
    RPoint point14 (227, 365, 1170046782815)
    RPoint point44 (227, 323, 1170046782831)
    RPoint point50 (227, 408, 1170046782831)
    RPoint point63 (227, 365, 1170046782815)
    RPoint point93 (227, 323, 1170046782846)
    RPoint point99 (227, 408, 1170046782846)
  277.0:
    RPoint p2 (277, 323, 1170046781706)
    RLine line32 RPoint p1 (277, 323, 1170046782831)
                 RPoint p2 (277, 408, 1170046782831)
    RPoint point36 (277, 323, 1170046782831)
    RPoint point38 (277, 408, 1170046782831)
    RPoint point62 (277, 365, 1170046782831)
    RPoint p1 (277, 323, 1170046781706)
    RLine line81 RPoint p1 (277, 323, 1170046782831)
                 RPoint p2 (277, 408, 1170046782831)
    RPoint point85 (277, 323, 1170046782846)
    RPoint point87 (277, 408, 1170046782846)
    RPoint point111 (277, 365, 1170046782846)
```

E.1.6 MinY Index

```
minY Shapes:
  323.0:
    RLine line8 RPoint p1 (177, 408, 1170046781488)
                RPoint p2 (277, 323, 1170046781706)
    RLine line11 RPoint p1 (277, 323, 1170046781706)
                 RPoint p2 (177, 408, 1170046781488)
minY Subshapes:
```

```
323.0:
  RPoint p2 (277, 323, 1170046781706)
  RLine line17 RPoint p1 (177, 323, 1170046782815)
               RPoint p2 (277, 323, 1170046782815)
  RLine line27 RPoint p1 (177, 323, 1170046782831)
               RPoint p2 (177, 408, 1170046782831)
  RLine line32 RPoint p1 (277, 323, 1170046782831)
               RPoint p2 (277, 408, 1170046782831)
  RPoint point35 (177, 323, 1170046782831)
  RPoint point36 (277, 323, 1170046782831)
  RPoint point44 (227, 323, 1170046782831)
  RPoint p1 (277, 323, 1170046781706)
  RLine line66 RPoint p1 (177, 323, 1170046782831)
               RPoint p2 (277, 323, 1170046782831)
  RLine line76 RPoint p1 (177, 323, 1170046782831)
               RPoint p2 (177, 408, 1170046782831)
  RLine line81 RPoint p1 (277, 323, 1170046782831)
               RPoint p2 (277, 408, 1170046782831)
  RPoint point84 (177, 323, 1170046782846)
  RPoint point85 (277, 323, 1170046782846)
  RPoint point93 (227, 323, 1170046782846)
365.5:
  RPoint point14 (227, 365, 1170046782815)
  RPoint point56 (177, 365, 1170046782831)
  RPoint point62 (277, 365, 1170046782831)
  RPoint point63 (227, 365, 1170046782815)
  RPoint point105 (177, 365, 1170046782846)
  RPoint point111 (277, 365, 1170046782846)
408.0:
  RPoint p1 (177, 408, 1170046781488)
```

```
    RLine line22 RPoint p1 (177, 408, 1170046782831)
                 RPoint p2 (277, 408, 1170046782831)
    RPoint point37 (177, 408, 1170046782831)
    RPoint point38 (277, 408, 1170046782831)
    RPoint point50 (227, 408, 1170046782831)
    RPoint p2 (177, 408, 1170046781488)
    RLine line71 RPoint p1 (177, 408, 1170046782831)
                 RPoint p2 (277, 408, 1170046782831)
    RPoint point86 (177, 408, 1170046782846)
    RPoint point87 (277, 408, 1170046782846)
    RPoint point99 (227, 408, 1170046782846)
maxX Shapes:
  277.0:
    RLine line8 RPoint p1 (177, 408, 1170046781488)
                RPoint p2 (277, 323, 1170046781706)
    RLine line11 RPoint p1 (277, 323, 1170046781706)
                 RPoint p2 (177, 408, 1170046781488)
```

E.1.7 MaxX Index

```
maxX Subshapes:
  177.0:
    RPoint p1 (177, 408, 1170046781488)
    RLine line27 RPoint p1 (177, 323, 1170046782831)
                 RPoint p2 (177, 408, 1170046782831)
    RPoint point35 (177, 323, 1170046782831)
    RPoint point37 (177, 408, 1170046782831)
    RPoint point56 (177, 365, 1170046782831)
    RPoint p2 (177, 408, 1170046781488)
    RLine line76 RPoint p1 (177, 323, 1170046782831)
```

```
             RPoint p2 (177, 408, 1170046782831)
  RPoint point84 (177, 323, 1170046782846)
  RPoint point86 (177, 408, 1170046782846)
  RPoint point105 (177, 365, 1170046782846)
227.0:
  RPoint point14 (227, 365, 1170046782815)
  RPoint point44 (227, 323, 1170046782831)
  RPoint point50 (227, 408, 1170046782831)
  RPoint point63 (227, 365, 1170046782815)
  RPoint point93 (227, 323, 1170046782846)
  RPoint point99 (227, 408, 1170046782846)
277.0:
  RPoint p2 (277, 323, 1170046781706)
  RLine line17 RPoint p1 (177, 323, 1170046782815)
               RPoint p2 (277, 323, 1170046782815)
  RLine line22 RPoint p1 (177, 408, 1170046782831)
               RPoint p2 (277, 408, 1170046782831)
  RLine line32 RPoint p1 (277, 323, 1170046782831)
               RPoint p2 (277, 408, 1170046782831)
  RPoint point36 (277, 323, 1170046782831)
  RPoint point38 (277, 408, 1170046782831)
  RPoint point62 (277, 365, 1170046782831)
  RPoint p1 (277, 323, 1170046781706)
  RLine line66 RPoint p1 (177, 323, 1170046782831)
               RPoint p2 (277, 323, 1170046782831)
  RLine line71 RPoint p1 (177, 408, 1170046782831)
               RPoint p2 (277, 408, 1170046782831)
  RLine line81 RPoint p1 (277, 323, 1170046782831)
               RPoint p2 (277, 408, 1170046782831)
  RPoint point85 (277, 323, 1170046782846)
```

```
    RPoint point87 (277, 408, 1170046782846)
    RPoint point111 (277, 365, 1170046782846)
```

E.1.8 MaxY Index

```
maxY Shapes:
  408.0:
    RLine line8 RPoint p1 (177, 408, 1170046781488)
                RPoint p2 (277, 323, 1170046781706)
    RLine line11 RPoint p1 (277, 323, 1170046781706)
                 RPoint p2 (177, 408, 1170046781488)
maxY Subshapes:
  323.0:
    RPoint p2 (277, 323, 1170046781706)
    RLine line17 RPoint p1 (177, 323, 1170046782815)
                 RPoint p2 (277, 323, 1170046782815)
    RPoint point35 (177, 323, 1170046782831)
    RPoint point36 (277, 323, 1170046782831)
    RPoint point44 (227, 323, 1170046782831)
    RPoint p1 (277, 323, 1170046781706)
    RLine line66 RPoint p1 (177, 323, 1170046782831)
                 RPoint p2 (277, 323, 1170046782831)
    RPoint point84 (177, 323, 1170046782846)
    RPoint point85 (277, 323, 1170046782846)
    RPoint point93 (227, 323, 1170046782846)
  365.5:
    RPoint point14 (227, 365, 1170046782815)
    RPoint point56 (177, 365, 1170046782831)
    RPoint point62 (277, 365, 1170046782831)
    RPoint point63 (227, 365, 1170046782815)
    RPoint point105 (177, 365, 1170046782846)
    RPoint point111 (277, 365, 1170046782846)
  408.0:
    RPoint p1 (177, 408, 1170046781488)
    RLine line22 RPoint p1 (177, 408, 1170046782831)
                 RPoint p2 (277, 408, 1170046782831)
    RLine line27 RPoint p1 (177, 323, 1170046782831)
                 RPoint p2 (177, 408, 1170046782831)
    RLine line32 RPoint p1 (277, 323, 1170046782831)
                 RPoint p2 (277, 408, 1170046782831)
    RPoint point37 (177, 408, 1170046782831)
    RPoint point38 (277, 408, 1170046782831)
    RPoint point50 (227, 408, 1170046782831)
    RPoint p2 (177, 408, 1170046781488)
    RLine line71 RPoint p1 (177, 408, 1170046782831)
                 RPoint p2 (277, 408, 1170046782831)
    RLine line76 RPoint p1 (177, 323, 1170046782831)
                 RPoint p2 (177, 408, 1170046782831)
    RLine line81 RPoint p1 (277, 323, 1170046782831)
                 RPoint p2 (277, 408, 1170046782831)
    RPoint point86 (177, 408, 1170046782846)
    RPoint point87 (277, 408, 1170046782846)
    RPoint point99 (227, 408, 1170046782846)
```

E.1.9 Area Index

```
area Shapes:
  131.24404748406687:
    RLine line8 RPoint p1 (177, 408, 1170046781488)
                RPoint p2 (277, 323, 1170046781706)
    RLine line11 RPoint p1 (277, 323, 1170046781706)
                 RPoint p2 (177, 408, 1170046781488)
area Subshapes:
  1.0:
    RPoint p1 (177, 408, 1170046781488)
    RPoint p2 (277, 323, 1170046781706)
    RPoint point14 (227, 365, 1170046782815)
    RPoint point35 (177, 323, 1170046782831)
    RPoint point36 (277, 323, 1170046782831)
    RPoint point37 (177, 408, 1170046782831)
    RPoint point38 (277, 408, 1170046782831)
    RPoint point44 (227, 323, 1170046782831)
    RPoint point50 (227, 408, 1170046782831)
    RPoint point56 (177, 365, 1170046782831)
    RPoint point62 (277, 365, 1170046782831)
    RPoint p1 (277, 323, 1170046781706)
    RPoint p2 (177, 408, 1170046781488)
    RPoint point63 (227, 365, 1170046782815)
    RPoint point84 (177, 323, 1170046782846)
    RPoint point85 (277, 323, 1170046782846)
    RPoint point86 (177, 408, 1170046782846)
    RPoint point87 (277, 408, 1170046782846)
    RPoint point93 (227, 323, 1170046782846)
    RPoint point99 (227, 408, 1170046782846)
    RPoint point105 (177, 365, 1170046782846)
    RPoint point111 (277, 365, 1170046782846)
  85.0:
    RLine line27 RPoint p1 (177, 323, 1170046782831)
                 RPoint p2 (177, 408, 1170046782831)
    RLine line32 RPoint p1 (277, 323, 1170046782831)
```

```
                RPoint p2 (277, 408, 1170046782831)
    RLine line76 RPoint p1 (177, 323, 1170046782831)
                 RPoint p2 (177, 408, 1170046782831)
    RLine line81 RPoint p1 (277, 323, 1170046782831)
                 RPoint p2 (277, 408, 1170046782831)
  100.0:
    RLine line17 RPoint p1 (177, 323, 1170046782815)
                 RPoint p2 (277, 323, 1170046782815)
    RLine line22 RPoint p1 (177, 408, 1170046782831)
                 RPoint p2 (277, 408, 1170046782831)
    RLine line66 RPoint p1 (177, 323, 1170046782831)
                 RPoint p2 (277, 323, 1170046782831)
    RLine line71 RPoint p1 (177, 408, 1170046782831)
                 RPoint p2 (277, 408, 1170046782831)
```

E.1.10 Width Index

```
width Shapes:
  100.0:
    RLine line8 RPoint p1 (177, 408, 1170046781488)
                RPoint p2 (277, 323, 1170046781706)
    RLine line11 RPoint p1 (277, 323, 1170046781706)
                 RPoint p2 (177, 408, 1170046781488)
width Subshapes:
  0.0:
    RLine line27 RPoint p1 (177, 323, 1170046782831)
                 RPoint p2 (177, 408, 1170046782831)
    RLine line32 RPoint p1 (277, 323, 1170046782831)
                 RPoint p2 (277, 408, 1170046782831)
    RLine line76 RPoint p1 (177, 323, 1170046782831)
```

```
             RPoint p2 (177, 408, 1170046782831)
  RLine line81 RPoint p1 (277, 323, 1170046782831)
             RPoint p2 (277, 408, 1170046782831)
1.0:
  RPoint p1 (177, 408, 1170046781488)
  RPoint p2 (277, 323, 1170046781706)
  RPoint point14 (227, 365, 1170046782815)
  RPoint point35 (177, 323, 1170046782831)
  RPoint point36 (277, 323, 1170046782831)
  RPoint point37 (177, 408, 1170046782831)
  RPoint point38 (277, 408, 1170046782831)
  RPoint point44 (227, 323, 1170046782831)
  RPoint point50 (227, 408, 1170046782831)
  RPoint point56 (177, 365, 1170046782831)
  RPoint point62 (277, 365, 1170046782831)
  RPoint p1 (277, 323, 1170046781706)
  RPoint p2 (177, 408, 1170046781488)
  RPoint point63 (227, 365, 1170046782815)
  RPoint point84 (177, 323, 1170046782846)
  RPoint point85 (277, 323, 1170046782846)
  RPoint point86 (177, 408, 1170046782846)
  RPoint point87 (277, 408, 1170046782846)
  RPoint point93 (227, 323, 1170046782846)
  RPoint point99 (227, 408, 1170046782846)
  RPoint point105 (177, 365, 1170046782846)
  RPoint point111 (277, 365, 1170046782846)
100.0:
  RLine line17 RPoint p1 (177, 323, 1170046782815)
             RPoint p2 (277, 323, 1170046782815)
  RLine line22 RPoint p1 (177, 408, 1170046782831)
```

```
             RPoint p2 (277, 408, 1170046782831)
    RLine line66 RPoint p1 (177, 323, 1170046782831)
             RPoint p2 (277, 323, 1170046782831)
    RLine line71 RPoint p1 (177, 408, 1170046782831)
             RPoint p2 (277, 408, 1170046782831)
```

E.1.11 Height Index

```
height Shapes:
  85.0:
    RLine line8 RPoint p1 (177, 408, 1170046781488)
            RPoint p2 (277, 323, 1170046781706)
    RLine line11 RPoint p1 (277, 323, 1170046781706)
             RPoint p2 (177, 408, 1170046781488)
height Subshapes:
  0.0:
    RLine line17 RPoint p1 (177, 323, 1170046782815)
             RPoint p2 (277, 323, 1170046782815)
    RLine line22 RPoint p1 (177, 408, 1170046782831)
             RPoint p2 (277, 408, 1170046782831)
    RLine line66 RPoint p1 (177, 323, 1170046782831)
             RPoint p2 (277, 323, 1170046782831)
    RLine line71 RPoint p1 (177, 408, 1170046782831)
             RPoint p2 (277, 408, 1170046782831)
  1.0:
    RPoint p1 (177, 408, 1170046781488)
    RPoint p2 (277, 323, 1170046781706)
    RPoint point14 (227, 365, 1170046782815)
    RPoint point35 (177, 323, 1170046782831)
    RPoint point36 (277, 323, 1170046782831)
    RPoint point37 (177, 408, 1170046782831)
    RPoint point38 (277, 408, 1170046782831)
    RPoint point44 (227, 323, 1170046782831)
    RPoint point50 (227, 408, 1170046782831)
    RPoint point56 (177, 365, 1170046782831)
    RPoint point62 (277, 365, 1170046782831)
    RPoint p1 (277, 323, 1170046781706)
    RPoint p2 (177, 408, 1170046781488)
    RPoint point63 (227, 365, 1170046782815)
    RPoint point84 (177, 323, 1170046782846)
    RPoint point85 (277, 323, 1170046782846)
    RPoint point86 (177, 408, 1170046782846)
    RPoint point87 (277, 408, 1170046782846)
    RPoint point93 (227, 323, 1170046782846)
    RPoint point99 (227, 408, 1170046782846)
    RPoint point105 (177, 365, 1170046782846)
    RPoint point111 (277, 365, 1170046782846)
  85.0:
    RLine line27 RPoint p1 (177, 323, 1170046782831)
                 RPoint p2 (177, 408, 1170046782831)
    RLine line32 RPoint p1 (277, 323, 1170046782831)
                 RPoint p2 (277, 408, 1170046782831)
    RLine line76 RPoint p1 (177, 323, 1170046782831)
                 RPoint p2 (177, 408, 1170046782831)
    RLine line81 RPoint p1 (277, 323, 1170046782831)
                 RPoint p2 (277, 408, 1170046782831)
```

E.1.12 Length Index

```
length Shapes:
  131.24404748406687:
    RLine line8 RPoint p1 (177, 408, 1170046781488)
                RPoint p2 (277, 323, 1170046781706)
    RLine line11 RPoint p1 (277, 323, 1170046781706)
                 RPoint p2 (177, 408, 1170046781488)
length Subshapes:
  85.0:
    RLine line27 RPoint p1 (177, 323, 1170046782831)
                 RPoint p2 (177, 408, 1170046782831)
    RLine line32 RPoint p1 (277, 323, 1170046782831)
                 RPoint p2 (277, 408, 1170046782831)
    RLine line76 RPoint p1 (177, 323, 1170046782831)
                 RPoint p2 (177, 408, 1170046782831)
    RLine line81 RPoint p1 (277, 323, 1170046782831)
                 RPoint p2 (277, 408, 1170046782831)
  100.0:
    RLine line17 RPoint p1 (177, 323, 1170046782815)
                 RPoint p2 (277, 323, 1170046782815)
    RLine line22 RPoint p1 (177, 408, 1170046782831)
                 RPoint p2 (277, 408, 1170046782831)
    RLine line66 RPoint p1 (177, 323, 1170046782831)
                 RPoint p2 (277, 323, 1170046782831)
    RLine line71 RPoint p1 (177, 408, 1170046782831)
                 RPoint p2 (277, 408, 1170046782831)
```

E.2 Indexing of an Arrow

This section shows the indexing values for after an arrow and all of its accessible components have been indexed.

E.2.1 Name Index

```
Shape Names:
  arrow_1:
    arrow553 Arrow
  line:
    RLine line8 RPoint p1 (177, 408, 1170046781488)
                RPoint p2 (277, 323, 1170046781706)
    RLine shaft RPoint head (277, 323, 1170046781706)
                RPoint tail (177, 408, 1170046781488)
    RLine head2 RPoint p1 (276, 321, 1170046787675)
                RPoint p2 (224, 319, 1170046787925)
    RLine line186 RPoint p1 (224, 319, 1170046787925)
                  RPoint p2 (276, 321, 1170046787675)
    RLine head1 RPoint p1 (278, 321, 1170046788519)
                RPoint p2 (273, 354, 1170046788753)
    RLine line358 RPoint p1 (273, 354, 1170046788753)
                  RPoint p2 (278, 321, 1170046788519)
Subshape Names:
  boundRight:
    RLine line604 RPoint p1 (278, 319, 1170046789081)
                  RPoint p2 (278, 408, 1170046789081)
  boundTopRight:
    RPoint point608 (278, 319, 1170046789081)
  boundBottomRight:
    RPoint point610 (278, 408, 1170046789081)
  boundBottomMiddle:
    RPoint point622 (227, 408, 1170046789081)
  boundTop:
    RLine line589 RPoint p1 (177, 319, 1170046789081)
```

```
                RPoint p2 (278, 319, 1170046789081)
boundTopMiddle:
  RPoint point616 (227, 319, 1170046789081)
boundLeft:
  RLine line599 RPoint p1 (177, 319, 1170046789081)
                RPoint p2 (177, 408, 1170046789081)
head1:
  RLine head1 RPoint p1 (278, 321, 1170046788519)
              RPoint p2 (273, 354, 1170046788753)
boundRightMiddle:
  RPoint point634 (278, 363, 1170046789081)
boundBottomLeft:
  RPoint point609 (177, 408, 1170046789081)
head2:
  RLine head2 RPoint p1 (276, 321, 1170046787675)
              RPoint p2 (224, 319, 1170046787925)
tail:
  RPoint tail (177, 408, 1170046781488)
shaft:
  RLine shaft RPoint head (277, 323, 1170046781706)
              RPoint tail (177, 408, 1170046781488)
center:
  RPoint point586 (227, 363, 1170046789065)
boundLeftMiddle:
  RPoint point628 (177, 363, 1170046789081)
boundBottom:
  RLine line594 RPoint p1 (177, 408, 1170046789081)
                RPoint p2 (278, 408, 1170046789081)
boundTopLeft:
  RPoint point607 (177, 319, 1170046789081)
  head:
    RPoint head (277, 323, 1170046781706)
pos: neg: hor: ver: angle:
  40.0:
    RLine line8 RPoint p1 (177, 408, 1170046781488)
                RPoint p2 (277, 323, 1170046781706)
    RLine shaft RPoint head (277, 323, 1170046781706)
                RPoint tail (177, 408, 1170046781488)
  81.0:
    RLine head1 RPoint p1 (278, 321, 1170046788519)
                RPoint p2 (273, 354, 1170046788753)
    RLine line358 RPoint p1 (273, 354, 1170046788753)
                  RPoint p2 (278, 321, 1170046788519)
  177.0:
    RLine head2 RPoint p1 (276, 321, 1170046787675)
                RPoint p2 (224, 319, 1170046787925)
    RLine line186 RPoint p1 (224, 319, 1170046787925)
                  RPoint p2 (276, 321, 1170046787675)
```

E.2.2 Type Index

```
  Shape:
    arrow553 Arrow
  DrawnShape:
    arrow553 Arrow
  Arrow:
    arrow553 Arrow
Subshape Types: Total = 5
  LAC:
    RLine head1 RPoint p1 (278, 321, 1170046788519)
```

```
            RPoint p2 (273, 354, 1170046788753)
  RLine head2 RPoint p1 (276, 321, 1170046787675)
            RPoint p2 (224, 319, 1170046787925)
  RLine shaft RPoint head (277, 323, 1170046781706)
            RPoint tail (177, 408, 1170046781488)
  RLine line589 RPoint p1 (177, 319, 1170046789081)
              RPoint p2 (278, 319, 1170046789081)
  RLine line594 RPoint p1 (177, 408, 1170046789081)
              RPoint p2 (278, 408, 1170046789081)
  RLine line599 RPoint p1 (177, 319, 1170046789081)
              RPoint p2 (177, 408, 1170046789081)
  RLine line604 RPoint p1 (278, 319, 1170046789081)
              RPoint p2 (278, 408, 1170046789081)
Shape:
  RLine head1 RPoint p1 (278, 321, 1170046788519)
            RPoint p2 (273, 354, 1170046788753)
  RLine head2 RPoint p1 (276, 321, 1170046787675)
            RPoint p2 (224, 319, 1170046787925)
  RLine shaft RPoint head (277, 323, 1170046781706)
            RPoint tail (177, 408, 1170046781488)
  RPoint head (277, 323, 1170046781706)
  RPoint tail (177, 408, 1170046781488)
  RPoint point586 (227, 363, 1170046789065)
  RLine line589 RPoint p1 (177, 319, 1170046789081)
              RPoint p2 (278, 319, 1170046789081)
  RLine line594 RPoint p1 (177, 408, 1170046789081)
              RPoint p2 (278, 408, 1170046789081)
  RLine line599 RPoint p1 (177, 319, 1170046789081)
              RPoint p2 (177, 408, 1170046789081)
  RLine line604 RPoint p1 (278, 319, 1170046789081)
```

```
              RPoint p2 (278, 408, 1170046789081)
  RPoint point607 (177, 319, 1170046789081)
  RPoint point608 (278, 319, 1170046789081)
  RPoint point609 (177, 408, 1170046789081)
  RPoint point610 (278, 408, 1170046789081)
  RPoint point616 (227, 319, 1170046789081)
  RPoint point622 (227, 408, 1170046789081)
  RPoint point628 (177, 363, 1170046789081)
  RPoint point634 (278, 363, 1170046789081)
DrawnShape:
  RLine head1 RPoint p1 (278, 321, 1170046788519)
              RPoint p2 (273, 354, 1170046788753)
  RLine head2 RPoint p1 (276, 321, 1170046787675)
              RPoint p2 (224, 319, 1170046787925)
  RLine shaft RPoint head (277, 323, 1170046781706)
              RPoint tail (177, 408, 1170046781488)
  RPoint head (277, 323, 1170046781706)
  RPoint tail (177, 408, 1170046781488)
  RPoint point586 (227, 363, 1170046789065)
  RLine line589 RPoint p1 (177, 319, 1170046789081)
                RPoint p2 (278, 319, 1170046789081)
  RLine line594 RPoint p1 (177, 408, 1170046789081)
                RPoint p2 (278, 408, 1170046789081)
  RLine line599 RPoint p1 (177, 319, 1170046789081)
                RPoint p2 (177, 408, 1170046789081)
  RLine line604 RPoint p1 (278, 319, 1170046789081)
                RPoint p2 (278, 408, 1170046789081)
  RPoint point607 (177, 319, 1170046789081)
  RPoint point608 (278, 319, 1170046789081)
  RPoint point609 (177, 408, 1170046789081)
  RPoint point610 (278, 408, 1170046789081)
  RPoint point616 (227, 319, 1170046789081)
  RPoint point622 (227, 408, 1170046789081)
  RPoint point628 (177, 363, 1170046789081)
  RPoint point634 (278, 363, 1170046789081)
Line:
  RLine head1 RPoint p1 (278, 321, 1170046788519)
              RPoint p2 (273, 354, 1170046788753)
  RLine head2 RPoint p1 (276, 321, 1170046787675)
              RPoint p2 (224, 319, 1170046787925)
  RLine shaft RPoint head (277, 323, 1170046781706)
              RPoint tail (177, 408, 1170046781488)
  RLine line589 RPoint p1 (177, 319, 1170046789081)
                RPoint p2 (278, 319, 1170046789081)
  RLine line594 RPoint p1 (177, 408, 1170046789081)
                RPoint p2 (278, 408, 1170046789081)
  RLine line599 RPoint p1 (177, 319, 1170046789081)
                RPoint p2 (177, 408, 1170046789081)
  RLine line604 RPoint p1 (278, 319, 1170046789081)
                RPoint p2 (278, 408, 1170046789081)
Point:
  RPoint head (277, 323, 1170046781706)
  RPoint tail (177, 408, 1170046781488)
  RPoint point586 (227, 363, 1170046789065)
  RPoint point607 (177, 319, 1170046789081)
  RPoint point608 (278, 319, 1170046789081)
  RPoint point609 (177, 408, 1170046789081)
  RPoint point610 (278, 408, 1170046789081)
  RPoint point616 (227, 319, 1170046789081)
  RPoint point622 (227, 408, 1170046789081)
```

```
    RPoint point628 (177, 363, 1170046789081)
    RPoint point634 (278, 363, 1170046789081)
```

E.2.3 X Index

```
x Shapes:
  227.5:
    arrow553 Arrow
x Subshapes:
  177.0:
    RPoint tail (177, 408, 1170046781488)
    RLine line599 RPoint p1 (177, 319, 1170046789081)
                  RPoint p2 (177, 408, 1170046789081)
    RPoint point607 (177, 319, 1170046789081)
    RPoint point609 (177, 408, 1170046789081)
    RPoint point628 (177, 363, 1170046789081)
  227.0:
    RLine shaft RPoint head (277, 323, 1170046781706)
                RPoint tail (177, 408, 1170046781488)
  227.5:
    RPoint point586 (227, 363, 1170046789065)
    RLine line589 RPoint p1 (177, 319, 1170046789081)
                  RPoint p2 (278, 319, 1170046789081)
    RLine line594 RPoint p1 (177, 408, 1170046789081)
                  RPoint p2 (278, 408, 1170046789081)
    RPoint point616 (227, 319, 1170046789081)
    RPoint point622 (227, 408, 1170046789081)
  250.0:
    RLine head2 RPoint p1 (276, 321, 1170046787675)
                RPoint p2 (224, 319, 1170046787925)
  275.5:
    RLine head1 RPoint p1 (278, 321, 1170046788519)
                RPoint p2 (273, 354, 1170046788753)
  277.0:
    RPoint head (277, 323, 1170046781706)
  278.0:
    RLine line604 RPoint p1 (278, 319, 1170046789081)
                  RPoint p2 (278, 408, 1170046789081)
    RPoint point608 (278, 319, 1170046789081)
    RPoint point610 (278, 408, 1170046789081)
    RPoint point634 (278, 363, 1170046789081)
```

E.2.4 Y Index

```
y Shapes:
  363.5:
    arrow553 Arrow
y Subshapes:
  319.0:
    RLine line589 RPoint p1 (177, 319, 1170046789081)
                  RPoint p2 (278, 319, 1170046789081)
    RPoint point607 (177, 319, 1170046789081)
    RPoint point608 (278, 319, 1170046789081)
    RPoint point616 (227, 319, 1170046789081)
  320.0:
    RLine head2 RPoint p1 (276, 321, 1170046787675)
                RPoint p2 (224, 319, 1170046787925)
  323.0:
    RPoint head (277, 323, 1170046781706)
  337.5:
    RLine head1 RPoint p1 (278, 321, 1170046788519)
                RPoint p2 (273, 354, 1170046788753)
  363.5:
    RPoint point586 (227, 363, 1170046789065)
    RLine line599 RPoint p1 (177, 319, 1170046789081)
                  RPoint p2 (177, 408, 1170046789081)
    RLine line604 RPoint p1 (278, 319, 1170046789081)
                  RPoint p2 (278, 408, 1170046789081)
    RPoint point628 (177, 363, 1170046789081)
    RPoint point634 (278, 363, 1170046789081)
  365.5:
    RLine shaft RPoint head (277, 323, 1170046781706)
                RPoint tail (177, 408, 1170046781488)
  408.0:
    RPoint tail (177, 408, 1170046781488)
    RLine line594 RPoint p1 (177, 408, 1170046789081)
                  RPoint p2 (278, 408, 1170046789081)
    RPoint point609 (177, 408, 1170046789081)
    RPoint point610 (278, 408, 1170046789081)
    RPoint point622 (227, 408, 1170046789081)
```

E.2.5 minX Index

```
minX Shapes:
  177.0:
    arrow553 Arrow
minX Subshapes:
  177.0:
    RLine shaft RPoint head (277, 323, 1170046781706)
                RPoint tail (177, 408, 1170046781488)
    RPoint tail (177, 408, 1170046781488)
    RLine line589 RPoint p1 (177, 319, 1170046789081)
                  RPoint p2 (278, 319, 1170046789081)
    RLine line594 RPoint p1 (177, 408, 1170046789081)
                  RPoint p2 (278, 408, 1170046789081)
    RLine line599 RPoint p1 (177, 319, 1170046789081)
                  RPoint p2 (177, 408, 1170046789081)
    RPoint point607 (177, 319, 1170046789081)
    RPoint point609 (177, 408, 1170046789081)
    RPoint point628 (177, 363, 1170046789081)
  224.0:
    RLine head2 RPoint p1 (276, 321, 1170046787675)
                RPoint p2 (224, 319, 1170046787925)
  227.5:
    RPoint point586 (227, 363, 1170046789065)
    RPoint point616 (227, 319, 1170046789081)
    RPoint point622 (227, 408, 1170046789081)
  273.0:
    RLine head1 RPoint p1 (278, 321, 1170046788519)
                RPoint p2 (273, 354, 1170046788753)
  277.0:
    RPoint head (277, 323, 1170046781706)
  278.0:
    RLine line604 RPoint p1 (278, 319, 1170046789081)
                  RPoint p2 (278, 408, 1170046789081)
    RPoint point608 (278, 319, 1170046789081)
    RPoint point610 (278, 408, 1170046789081)
    RPoint point634 (278, 363, 1170046789081)
```

E.2.6 MinY Index

```
minY Shapes:
  319.0:
    arrow553 Arrow
minY Subshapes:
  319.0:
    RLine head2 RPoint p1 (276, 321, 1170046787675)
                RPoint p2 (224, 319, 1170046787925)
    RLine line589 RPoint p1 (177, 319, 1170046789081)
                  RPoint p2 (278, 319, 1170046789081)
    RLine line599 RPoint p1 (177, 319, 1170046789081)
                  RPoint p2 (177, 408, 1170046789081)
    RLine line604 RPoint p1 (278, 319, 1170046789081)
                RPoint p2 (278, 408, 1170046789081)
    RPoint point607 (177, 319, 1170046789081)
    RPoint point608 (278, 319, 1170046789081)
    RPoint point616 (227, 319, 1170046789081)
  321.0:
    RLine head1 RPoint p1 (278, 321, 1170046788519)
                RPoint p2 (273, 354, 1170046788753)
  323.0:
    RLine shaft RPoint head (277, 323, 1170046781706)
                RPoint tail (177, 408, 1170046781488)
    RPoint head (277, 323, 1170046781706)
  363.5:
    RPoint point586 (227, 363, 1170046789065)
    RPoint point628 (177, 363, 1170046789081)
    RPoint point634 (278, 363, 1170046789081)
  408.0:
    RPoint tail (177, 408, 1170046781488)
    RLine line594 RPoint p1 (177, 408, 1170046789081)
                  RPoint p2 (278, 408, 1170046789081)
    RPoint point609 (177, 408, 1170046789081)
    RPoint point610 (278, 408, 1170046789081)
    RPoint point622 (227, 408, 1170046789081)
```

E.2.7 MaxX Index

```
maxX Shapes:
  278.0:
    arrow553 Arrow
maxX Subshapes:
  177.0:
    RPoint tail (177, 408, 1170046781488)
    RLine line599 RPoint p1 (177, 319, 1170046789081)
                  RPoint p2 (177, 408, 1170046789081)
    RPoint point607 (177, 319, 1170046789081)
    RPoint point609 (177, 408, 1170046789081)
    RPoint point628 (177, 363, 1170046789081)
  227.5:
    RPoint point586 (227, 363, 1170046789065)
    RPoint point616 (227, 319, 1170046789081)
    RPoint point622 (227, 408, 1170046789081)
  276.0:
    RLine head2 RPoint p1 (276, 321, 1170046787675)
                RPoint p2 (224, 319, 1170046787925)
  277.0:
    RLine shaft RPoint head (277, 323, 1170046781706)
                RPoint tail (177, 408, 1170046781488)
    RPoint head (277, 323, 1170046781706)
  278.0:
    RLine head1 RPoint p1 (278, 321, 1170046788519)
                RPoint p2 (273, 354, 1170046788753)
    RLine line589 RPoint p1 (177, 319, 1170046789081)
                  RPoint p2 (278, 319, 1170046789081)
    RLine line594 RPoint p1 (177, 408, 1170046789081)
                  RPoint p2 (278, 408, 1170046789081)
    RLine line604 RPoint p1 (278, 319, 1170046789081)
                  RPoint p2 (278, 408, 1170046789081)
    RPoint point608 (278, 319, 1170046789081)
    RPoint point610 (278, 408, 1170046789081)
    RPoint point634 (278, 363, 1170046789081)
```

E.2.8 MaxY Index

```
maxY Shapes:
  408.0:
    arrow553 Arrow
maxY Subshapes:
  319.0:
    RLine line589 RPoint p1 (177, 319, 1170046789081)
                  RPoint p2 (278, 319, 1170046789081)
    RPoint point607 (177, 319, 1170046789081)
    RPoint point608 (278, 319, 1170046789081)
    RPoint point616 (227, 319, 1170046789081)
  321.0:
    RLine head2 RPoint p1 (276, 321, 1170046787675)
                RPoint p2 (224, 319, 1170046787925)
  323.0:
    RPoint head (277, 323, 1170046781706)
  354.0:
    RLine head1 RPoint p1 (278, 321, 1170046788519)
                RPoint p2 (273, 354, 1170046788753)
  363.5:
    RPoint point586 (227, 363, 1170046789065)
    RPoint point628 (177, 363, 1170046789081)
    RPoint point634 (278, 363, 1170046789081)
  408.0:
    RLine shaft RPoint head (277, 323, 1170046781706)
                RPoint tail (177, 408, 1170046781488)
    RPoint tail (177, 408, 1170046781488)
    RLine line594 RPoint p1 (177, 408, 1170046789081)
                  RPoint p2 (278, 408, 1170046789081)
    RLine line599 RPoint p1 (177, 319, 1170046789081)
                  RPoint p2 (177, 408, 1170046789081)
    RLine line604 RPoint p1 (278, 319, 1170046789081)
                  RPoint p2 (278, 408, 1170046789081)
    RPoint point609 (177, 408, 1170046789081)
    RPoint point610 (278, 408, 1170046789081)
    RPoint point622 (227, 408, 1170046789081)
```

E.2.9 Area Index

```
area Shapes:
  8989.0:
    arrow553 Arrow
area Subshapes:
  1.0:
    RPoint head (277, 323, 1170046781706)
    RPoint tail (177, 408, 1170046781488)
    RPoint point586 (227, 363, 1170046789065)
    RPoint point607 (177, 319, 1170046789081)
    RPoint point608 (278, 319, 1170046789081)
    RPoint point609 (177, 408, 1170046789081)
    RPoint point610 (278, 408, 1170046789081)
    RPoint point616 (227, 319, 1170046789081)
    RPoint point622 (227, 408, 1170046789081)
    RPoint point628 (177, 363, 1170046789081)
    RPoint point634 (278, 363, 1170046789081)
  33.37663853655727:
    RLine head1 RPoint p1 (278, 321, 1170046788519)
                RPoint p2 (273, 354, 1170046788753)
  52.03844732503075:
    RLine head2 RPoint p1 (276, 321, 1170046787675)
                RPoint p2 (224, 319, 1170046787925)
  89.0:
    RLine line599 RPoint p1 (177, 319, 1170046789081)
                  RPoint p2 (177, 408, 1170046789081)
    RLine line604 RPoint p1 (278, 319, 1170046789081)
                  RPoint p2 (278, 408, 1170046789081)
  101.0:
    RLine line589 RPoint p1 (177, 319, 1170046789081)
                  RPoint p2 (278, 319, 1170046789081)
    RLine line594 RPoint p1 (177, 408, 1170046789081)
                  RPoint p2 (278, 408, 1170046789081)
  131.24404748406687:
    RLine shaft RPoint head (277, 323, 1170046781706)
                RPoint tail (177, 408, 1170046781488)
```

E.2.10 Width Index

```
width Shapes:
  101.0:
    arrow553 Arrow
width Subshapes:
  0.0:
    RLine line599 RPoint p1 (177, 319, 1170046789081)
                  RPoint p2 (177, 408, 1170046789081)
    RLine line604 RPoint p1 (278, 319, 1170046789081)
                  RPoint p2 (278, 408, 1170046789081)
  1.0:
    RPoint head (277, 323, 1170046781706)
    RPoint tail (177, 408, 1170046781488)
    RPoint point586 (227, 363, 1170046789065)
    RPoint point607 (177, 319, 1170046789081)
    RPoint point608 (278, 319, 1170046789081)
    RPoint point609 (177, 408, 1170046789081)
    RPoint point610 (278, 408, 1170046789081)
    RPoint point616 (227, 319, 1170046789081)
    RPoint point622 (227, 408, 1170046789081)
    RPoint point628 (177, 363, 1170046789081)
    RPoint point634 (278, 363, 1170046789081)
  5.0:
    RLine head1 RPoint p1 (278, 321, 1170046788519)
                RPoint p2 (273, 354, 1170046788753)
  52.0:
    RLine head2 RPoint p1 (276, 321, 1170046787675)
                RPoint p2 (224, 319, 1170046787925)
  100.0:
    RLine shaft RPoint head (277, 323, 1170046781706)
                RPoint tail (177, 408, 1170046781488)
  101.0:
    RLine line589 RPoint p1 (177, 319, 1170046789081)
                  RPoint p2 (278, 319, 1170046789081)
    RLine line594 RPoint p1 (177, 408, 1170046789081)
                  RPoint p2 (278, 408, 1170046789081)
```

E.2.11 Height Index

```
height Shapes:
  89.0:
    arrow553 Arrow
height Subshapes:
  0.0:
    RLine line589 RPoint p1 (177, 319, 1170046789081)
                  RPoint p2 (278, 319, 1170046789081)
    RLine line594 RPoint p1 (177, 408, 1170046789081)
                  RPoint p2 (278, 408, 1170046789081)
  1.0:
    RPoint head (277, 323, 1170046781706)
    RPoint tail (177, 408, 1170046781488)
    RPoint point586 (227, 363, 1170046789065)
    RPoint point607 (177, 319, 1170046789081)
    RPoint point608 (278, 319, 1170046789081)
    RPoint point609 (177, 408, 1170046789081)
    RPoint point610 (278, 408, 1170046789081)
    RPoint point616 (227, 319, 1170046789081)
    RPoint point622 (227, 408, 1170046789081)
    RPoint point628 (177, 363, 1170046789081)
    RPoint point634 (278, 363, 1170046789081)
  2.0:
    RLine head2 RPoint p1 (276, 321, 1170046787675)
                RPoint p2 (224, 319, 1170046787925)
  33.0:
    RLine head1 RPoint p1 (278, 321, 1170046788519)
                RPoint p2 (273, 354, 1170046788753)
  85.0:
    RLine shaft RPoint head (277, 323, 1170046781706)
                RPoint tail (177, 408, 1170046781488)
  89.0:
    RLine line599 RPoint p1 (177, 319, 1170046789081)
                  RPoint p2 (177, 408, 1170046789081)
    RLine line604 RPoint p1 (278, 319, 1170046789081)
                  RPoint p2 (278, 408, 1170046789081)
```

E.2.12 Length Index

```
length Shapes: length Subshapes:
  33.37663853655727:
    RLine head1 RPoint p1 (278, 321, 1170046788519)
                RPoint p2 (273, 354, 1170046788753)
  52.03844732503075:
    RLine head2 RPoint p1 (276, 321, 1170046787675)
                RPoint p2 (224, 319, 1170046787925)
  89.0:
    RLine line599 RPoint p1 (177, 319, 1170046789081)
                  RPoint p2 (177, 408, 1170046789081)
    RLine line604 RPoint p1 (278, 319, 1170046789081)
                  RPoint p2 (278, 408, 1170046789081)
```

```
101.0:
  RLine line589 RPoint p1 (177, 319, 1170046789081)
                RPoint p2 (278, 319, 1170046789081)
  RLine line594 RPoint p1 (177, 408, 1170046789081)
                RPoint p2 (278, 408, 1170046789081)
131.24404748406687:
  RLine shaft RPoint head (277, 323, 1170046781706)
              RPoint tail (177, 408, 1170046781488)
```

Bibliography

[1] Sinan Si Abhir. *UML in a Nutshell: A Desktop Quick Reference*. O'Reilly, Cambridge, MA, 1998.

[2] Aaron Adler. Creating a multimodal design environment using speech and sketching. In *Third Annual MIT CSAIL Student Oxygen Workshop*, 2003.

[3] Aaron Adler. Segmentation and alignment of speech and sketching in a design environment. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, February 2003.

[4] Aaron Adler and Randall Davis. Speech and sketching for multimodal design. In *Proceedings of the 9th International Conference on Intelligent User Interfaces*, pages 214–216. ACM Press, 2004.

[5] James F. Allen, Donna K. Byron, Myroslava Dzikovska, George Ferguson, Lucian Galescu, and Amanda Stent. Towards conversational human-computer interaction. *AI Magazine*, 22(4):27–38, 2001.

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

[7] Christine Alvarado. Dynamically constructed bayesian networks for sketch understanding. In *Proceedings of the 3nd Annual MIT Student Oxygen Workshop*, 2003.

[8] Christine Alvarado. *Multi-Domain Sketch Understanding.* PhD thesis, Massachusetts Institute of Technology, August 2004.

[9] Christine Alvarado. Sketch recognition and usability: Guidelines for design and development. In *Proceedings of AAAI Fall Symposium on Pen-Based Interfaces*, 2004.

[10] Christine Alvarado and Randall Davis. Preserving the freedom of sketching to create a natural computer-based sketch tool. In *Proceedings of Human Computer Interaction International*, 2001.

[11] Christine Alvarado and Randall Davis. Resolving ambiguities to create a natural sketch based interface. In *Proceedings of IJCAI-2001*, August 2001.

[12] Christine Alvarado and Randall Davis. Sketchread: A multi-domain sketch recognition engine. In *Proceedings of UIST '04*, pages 23–32, 2004.

[13] Richard Anderson, Ruth Anderson, Beth Simon, Steven Wolfman, T. VanDeGrift, and Ken Yasuhara. Experiences with a tablet pc based lecture presentation system in computer science courses. In *Proc. SIGCSE '04*, 2004.

[14] Marco Anelli, Alessandro Micarelli, and Enver Sangineto. Content based image retrieval for unsegmented images. In *AIIA (Italian Association of Artificial Intelligence)*, 2003.

[15] Dana Angluin. Queries revisited. In *Proceedings of the 12th International Conference on Algorithmic Learning Theory*, volume LNAI 2225, page 1231, 2001.

[16] Sameer Antani, Rangachar Kasturi, and Ramesh Jain. A survey on the use of pattern recognition methods for abstraction, indexing, and retrieval of images and video. *Pattern Recognition*, 35:945–965, 2002.

[17] Ajay Apte, Van Vo, and Takayuki Dan Kimura. Recognizing multistroke geometric shapes: An experimental evaluation. In *UIST*, pages 121–128, 1993.

[18] Farshid Arman and J. K. Aggarwal. Cad-based vision: Object recognition strategies in cluttered range images using recognition strategies. *CVGIP: Image Understanding*, pages 33–48, 1993.

[19] James Arvo and Kevin Novins. Fluid sketches: Continous recognition and morphing of simple hand-drawn shapes. In *UIST*, 2000.

[20] James Arvo and Kevin Novins. Smart text: A synthesis of recognition and morphing. In *AAAI Spring Symposium on Smart Graphics*, pages 140–147, Stanford, California, 2000.

[21] Greg J. Badros, Alan Borning, and Peter J. Stuckey. The cassowary linear arithmetic constraint solving algorithm. In *ACM Transactions on Computer Human Interaction*, volume 8(4), pages 267–306, December 2001.

[22] M.-F. Balcan, A. Beygelzimer, and J. Langford. Agnostic active learning. In *International Conference on Machine Learning*, 2006.

[23] Srinivas Bangalore and Michael Johnston. Balancing data-driven and rule-based approaches in the context of a multimodal conversational system. In *Proceedings of HLT-NAACL'04*, pages 33–40. ACL Press, 2004.

[24] M. Banks and E. Cohen. Realtime spline curves from interactively sketched data. In *SIGGRAPH, Symposium on 3D Graphics*, pages 99–107, 1990.

[25] Thomas Baudel. A mark-based interaction paradigm for free-hand drawing. In *UIST '94: Proceedings of the 7th Annual ACM Symposium on User Interface Software and Technology*, pages 185–192. ACM Press, 1994.

[26] Serge Belongie, Jitendra Malik, and Jan Puzicha. Shape matching and object recognition using shape contexts. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 24(4):509–522, April 2002.

[27] A. Bentsson and J. Eklundh. Shape representation by multiscale contour approximation. *IEEE PAMI 13*, pages 85–93, 1992.

[28] F. Bergenti and A. Poggi. Agent-oriented software construction with UML. *Handbook of Software Engineering and Knowledge Engineering*, 2, 2001.

[29] Elie Bienenstock, Stuart Geman, and Daniel Potter. Compositionality, mdl priors, and object recognition. In T. Petsche M. C. Mozer, M. I. Jordan, editor, *Advances in Neural Information Processing Systems 9*, pages 838–844. MIT Press, 1997.

[30] Oliver Bimber, L. Miguel Encarnacao, and Andre Stork. 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*, 24(6):851–867, 2000.

[31] Alberto Del Bimbo and Pietro Pala. Visual image retrieval by elastic matching of user sketches. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 19(2):121–132, 1997.

[32] D. Blostein and L. Haken. Using diagram generation software to improve diagram recognition: A case study of music notation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 21(11), 1999.

[33] A. Blum and T. Mitchell. Combining labeled and unlabeled data with co-training. In *COLT: Proceedings of the Workshop on Computational Learning Theory*, pages 92–100. Morgan Kaufmann, 1998.

[34] Péter Pál Boda and Edward Filisko. Virtual modality: a framework for testing and building multimodal applications. In *HLT-NAACL 2004 Workshop on Spoken Language Understanding for Conversational Systems*, May 2004.

[35] Richard A. Bolt. Put-that-there: Voice and gesture at the graphics interface. In *Proceedings of the 7th Annual Conference on Computer Graphics and Interactive Techniques*, pages 262–270, 1980.

[36] C. Bonwell and J. Eison. Active learning: Creating excitement in the classroom. AEHE-ERIC Higher Education Report 1, AEHE-ERIC, Washington, D.C., 1991. ISBN 1-87838-00-87.

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

[38] A.K. Brown and M. Parker. *Dance Notation for Beginners*. Dance Books, London, 1984.

[39] R. Brunelli, O. Mich, and C.M. Modena. A survey on video indexing. *IRST Technical Report*, 1996.

[40] J. S. Bruner. The act of discovery. *Harvard Educational Review*, 31(1):21–32, 1961.

[41] Anabela Caetano, Neri Goulart, Manuel Fonseca, and Joaquim Jorge. JavaSketchIt: Issues in sketching the look of user interfaces. *Sketch Understanding, Papers from the 2002 AAAI Spring Symposium*, 2002.

[42] Anabela Caetano, Neri Goulart, Manuel Fonseca, and Joaquim Jorge. Sketching user interfaces with visual patterns. In *Proceedings of the 1st Ibero-American Symposium in Computer Graphics (SIACG02)*, pages 271–279, Guimares, Portugal, 2002.

[43] Chris Calhoun, Thomas F. Stahovich, Tolga Kurtoglu, and Levent Burak Kara. Recognizing multi-stroke symbols. *Sketch Understanding, Papers from the 2002 AAAI Spring Symposium*, pages 15–23, March 25-27 2002.

[44] Andrea Califano and Rakesh Mohan. Multidimensional indexing for recognizing visual shapes. *IEEE Transaction on Pattern Analysis and Machine Intelligence*, pages 373–391, 1994.

[45] Sonya Cates and Randall Davis. New approach to early sketch processing. In *Making Pen-Based Interaction Intelligent and Natural*, pages 29–34, Menlo Park, California, October 21-24 2004. AAAI Press.

[46] O. Chapelle, B. Schölkopf, and A. Zien. *Semi-Supervised Learning.* MIT Press, Cambridge, MA, 2006.

[47] S.J. Cho and S.I. Yoo. Image retrieval using topological structure of user sketch. In *Proceedings of IEEE SMC98*, 1998.

[48] Michael H. Coen, Brenton Phillips, Nimrod Warshawsky, Luke Weisman, Stephen Peters, and Peter Finin. Meeting the computational needs of intelligent environments: The metaglue system. In *Proceedings of MANSE'99*, 1999.

[49] P. R. Cohen, M. Johnston, D. R. McGee, S. L. Oviatt, J. Pittman, I. Smith, L. Chen, and J. Clowi. Quickset: Multimodal interaction for distributed applications. In *Proceedings of Mutimedia '97*, pages 31–40. ACM Press, 1997.

[50] Philip R. Cohen, M. Johnston, D. McGee, Sharon L. Oviatt, J. Clow, and I. Smith. The efficiency of multimodal interaction: A case study. In *Proceedings of the International Conference on Spoken Language*, 1998.

[51] David Cohn, Les Atlas, and Richard Ladner. Improving generalization with active learning. In *Machine Learning*, pages 201–221, 1994.

[52] Dukane Corporation. Dukane a/v products division mimio white paper. Technical report, Dukane Corporation, June 28 2001.

[53] Mauro Costa and Linda G. Shapiro. Object recognition and pose with relational indexing. *Computer Vision and Image Understanding*, pages 364–477, 2000.

[54] Gennaro Costagliola, Genoveffa Tortora, Sergio Orefice, and De Lucia. Automatic generation of visual programming environments. In *IEEE Computer*, pages 56–65, 1995.

[55] Nils Dahlback, Arne Jonsson, and Lars Ahrenberg. Wizard of Oz studies - why and how. *Intelligent User Interfaces (IUI93)*, pages 193–200, 1993.

[56] Christian Heide Damm, Klaus Marius Hansen, and Michael Thomsen. Tool support for cooperative object-oriented design: Gesture based modeling on an electronic whiteboard. In *CHI 2000*, pages 518–525. CHI, April 2000.

[57] L. David. *Perceptual Organization and Visual Recognition.* Kluwer Academic Publishers, Boston, MA, 1985.

[58] Ellen Yi-Luen Do. 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)*, pages 161–172, July 2001.

[59] Ellen Yi-Luen Do. Functional and formal reasoning in architectural sketches. *Sketch Understanding, Papers from the 2002 AAAI Spring Symposium*, pages 37–44, March 25-27 2002.

[60] Ellen Yi-Luen Do and Mark D. Gross. Drawing as a means to design reasoning. *AI and Design*, 1996.

[61] Max J. Egenhofer. Query processing in spatial-query-by-sketch. In *Journal of Visual Languages and Computing*, volume 8-4, pages 403–424, 1997.

[62] L. Eggli. Sketching with constraints. Master's thesis, University of Utah, 1994.

[63] Jacob Eisenstein and C. Mario Christoudias. A salience-based approach to gesture-speech alignment. In *HLT-NAACL'04*, pages 25–32. ACL Press, 2004.

[64] Jacob Eisenstein and Chris Mario Christoudias. Mulitmodal alignment by optimization. In *Third Annual MIT CSAIL Student Oxygen Workshop*, 2003.

[65] Jacob Eisenstein and Randall Davis. Natural gesture in descriptive monologues. In *UIST'03 Supplemental Proceedings*, pages 69–70. ACM Press, 2003.

[66] Jacob Eisenstein and Randall Davis. Visual and linguistic information in gesture classification. In *Proceedings of International Conference on Multimodal Interfaces(ICMI'04)*. ACM Press, 2004.

[67] Jacob Eisenstein and Randall Davis. Gestural cues for sentence segmentation. Technical Report AIM-2005-014, MIT AI Memo, 2005.

[68] Jacob Eisenstein and Randall Davis. Gestural features for sentence segmentation. In *Proceedings of 6th European Gesture Workshop (GW2005)*. Springer-Verlag, 2005.

[69] Palm Europe. Assessing enterprise requirements for handheld computing. *A Palm White Paper*, 2002.

[70] Ronald W. Ferguson and Kenneth D. Forbus. A cognitive approach to sketch understanding. *Sketch Understanding, Papers from the 2002 AAAI Spring Symposium*, pages 45–50, March 25-27 2002.

[71] Ronald W. Ferguson, Robert A. Rasch, William Turmel, and Kenneth D. Forbus. Qualitative spacial interpretation of course-of-action diagrams. In *Proceedings of the 14th International Workshop on Qualitative Reasoning*, Morelia, Mexico, 2000.

[72] Fabian Di Fiore and Frank Van Reeth. A multi-level sketching tool for pencil and paper animation. *Sketch Understanding, Papers from the 2002 AAAI Spring Symposium*, pages 32–36, March 25-27 2002.

[73] F. Flippo, A. Krebs, and I. Marsic. A framework for rapid development of multimodal interfaces. In *Proc. of International Conference on Multimodal Interfaces (ICMI)*, 2003.

[74] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Introduction to Computer Graphics*. Addison Wesley, Reading, Massachusetts, 1999.

[75] Mark Foltz. Ligature, gesture-based configuration of the e21 intelligent environment. In *Proceedings of the MIT Student Oxygen Workshop*, 2001.

[76] Manuel J. Fonseca and Joaquim A. Jorge. Visual languages for sketching documents. In *IEEE Symposium on Visual Languages*, Seattle, Washington, September 10-14 2000.

[77] Manuel J. Fonseca, César Pimentel, and Joaquim Jorge. Cali: An online scribble recognizer for calligraphic interfaces. *Sketch Understanding, Papers from the 2002 AAAI Spring Symposium*, pages 51–58, March 25-27 2002.

[78] Kenneth Forbus, R. Ferguson, and J. Usher. Towards a computational model of sketching. In *Intelligent User Interfaces '01*, pages 77–83, 2001.

[79] Kenneth D. Forbus, Kate Lockwood, Matthew Klenk, Emmett Tomai, and Jeffrey Usher. Open-domain sketch understanding: The nusketch approach. In *Making Pen-Based Interaction Intelligent and Natural*, pages 58–63, Menlo Park, California, October 21-24 2004. AAAI Fall Symposium.

[80] Kenneth D. Forbus, Jeffry Usher, and Vernell Chapman. Sketching for military course of action diagrams. In *Proceedings of IUI 2003*, 2003.

[81] A. S. Forsberg, M. K. Dieterich, and R. C. Zeleznik. The music notepad. In *Proceedings of UIST '98*. ACM SIGGRAPH, 1998.

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

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

[84] Krzysztof Gajos. Rascal - a resource manager for multi agent systems in smart spaces. In *Proceedings of CEEMAS'01*, Cracow, Poland, 1999. Also available in LNAI 2296.

[85] Andrew Gelsey. Automated reasoning about machine geometry and kinematics. In *IEEE*, 1987.

[86] Leslie Genari, Levent Burak Kara, and Thomas F. Stahovich. Combining geometry and domain knowledge to interpret hand-drawn diagrams. In *Making Pen-Based Interaction Intelligent and Natural*, Menlo Park, California, October 21-24 2004. AAAI Fall Symposium.

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

[88] Susan Goldin-Meadow, San Kim, and Melissa Singer. What the teachers hands tell the students mind about math. *Journal of Educational Psychology*, 91:720–730, 1999.

[89] Erich Goldmeier. Similarity in visually perceived forms. In *Psychological Issues*, volume 8:1, 1972.

[90] Christian Griesbeck. Labanotation handwriting recognition. WEB, 1996.

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

[92] Mark Gross and Ellen Yi-Luen Do. Ambiguous intentions: a paper-like interface for creative design. In *Proceedings of UIST 96*, pages 183–192, 1996.

[93] Mark D. Gross. Recognizing and interpreting diagrams in design. In *2nd Annual International Conference on Image Processing*, pages 308–311, 1995.

[94] Mark D. Gross. The electronic cocktail napkin - a computational environment for working with design diagrams. *Design Studies*, 17:53–69, 1996.

[95] Mark D. Gross and Ellen Yi-Luen Do. Demonstrating the electronic cocktail napkin: a paper-like interface for early design. *'Common Ground' appeared in ACM Conference on Human Factors in Computing (CHI)*, pages 5–6, 1996.

[96] Tracy Hammond. Natural sketch recognition in UML class diagrams. In *Proceedings of the MIT Student Oxygen Workshop*, 2001.

[97] Tracy Hammond. A domain description language for sketch recognition. In *Proceedings of the 2nd Annual MIT Student Oxygen Workshop*, 2002.

[98] Tracy Hammond. Automatically generating sketch interfaces from shape descriptions. In *Proceedings of the 4th Annual MIT Student Oxygen Workshop*, 2004.

[99] Tracy Hammond and Randall Davis. Tahuti: A geometrical sketch recognition system for UML class diagrams. *AAAI Spring Symposium on Sketch Understanding*, pages 59–68, March 25-27 2002.

[100] Tracy Hammond and Randall Davis. LADDER: A language to describe drawing, display, and editing in sketch recognition. In *Proceedings of the 2003 Internaltional Joint Conference on Artificial Intelligence (IJCAI-03)*, Acapulco, Mexico, 2003.

[101] Tracy Hammond and Randall Davis. Automatically transforming symbolic shape descriptions for use in sketch recognition. In *Proceedings of the Nineteenth National Conference on Artificial Intelligence (AAAI-04)*, pages 450–456, San Jose, CA, 2004.

[102] Tracy Hammond and Randall Davis. ShaDy: A shape description debugger for use in sketch recognition. *AAAI Fall Symposium on Making Pen-Based Interaction Intelligent and Natural*, 2004.

[103] Tracy Hammond and Randall Davis. LADDER, a sketching language for user interface developers. *Elsevier, Computers and Graphics*, 28:518–532, 2005.

[104] Tracy Hammond and Randall Davis. Interactive learning of structural shape descriptions from automatically-generated near-miss examples. *Intelligent User Interfaces (IUI)*, 2006.

[105] Tracy Hammond, Krzysztof Gajos, Randall Davis, and Howard Shrobe. An agent-based system for capturing and indexing software design meetings. In *Proceedings of International Workshop on Agents In Design, WAID'02*, 2002.

[106] Tracy Hammond and Jan Hammond. Gender-based underrepresentation in computer science and related disciplines. In *Proceedings of the 32nd ASEE/IEEE Frontiers in Education Conference*, Boston, MA, 2002.

[107] Tracy Hammond, Metin Sezgin, Olya Veselova, Aaron Adler, Michael Oltmans, Christine Alvarado, and Rebecca Hitchcock. Multi-domain sketch recognition. In *Proceedings of the 2nd Annual MIT Student Oxygen Workshop*, 2002.

[108] Song Han and Grard Medioni. 3dsketch: modeling by digitizing with a smart 3d pen. In *Proceedings of the Fifth ACM International Conference on Multimedia*, pages 41–49, 1997.

[109] Nicholas Hanssens, Ajay Kulkarni, Rattapoom Tuchinda, and Tyler Horton. Building agent-based intelligent workspaces. In *Proceedings of The International Workshop on Agents for Business Automation*, 2000.

[110] Marti Hearst. Sketching intelligent systems. *IEEE Intelligent Systems*, pages 10–18, May/June 1998.

[111] Richard Helm, Kim Marriott, and Martin Odersky. Building visual language parsers. In *Proceedings of CHI 1991*, pages 105–112, 1991.

[112] Christopher Herot. Graphical input through machine recognition of sketches. In *Proceedings of the 3rd Annual Conference on Computer Graphics and Interactive Techniques*, pages 97 – 102, 1976.

[113] Jason Hong, James Landay, A. Chris Long, and Jennifer Mankoff. Sketch recognizers from the end-user's, the designer's, and the programmer's perspective. *Sketch Understanding, Papers from the 2002 AAAI Spring Symposium*, pages 73–77, March 25-27 2002.

[114] Robert Horn. *Visual Language: Global Communication for the 21st Century*. MacroVU, Inc, December 1998.

[115] Heloise Hse, Michael Shilman, and A. Richard Newton. Robust sketched symbol fragmentation using templates. In *Proceedings of the 9th International Conference on Intelligent User Interface*, pages 156–160. ACM Press, 2004.

[116] Heloise Hse, Michael Shilman, A. Richard Newton, and James Landay. Sketch-based user interfaces for collaborative object-oriented modeling. Berkley CS260 Class Project, December 1999.

[117] Heloise Hwawen Hse and A. Richard Newton. Recognition and beautification of multi-stroke symbols in digital ink. In *Making Pen-Based Interaction Intelligent and Natural*, pages 78–84, Menlo Park, California, October 21-24 2004. AAAI Fall Symposium.

[118] Ideogramic. *Ideogramic UML^{TM}*. Ideogramic ApS, Denmark, http://www.ideogramic.com/products/, 2001.

[119] T. Igarashi, Satoshi Matsuoka, Sachiko Kawachiya, and Hidehiko Tanaka. Interactive beautification: A technique for rapid geometric design. In *UIST '97*, pages 105–114, 1997.

[120] Takeo Igarashi, Satoshi Matsuoka, and Hidehiko Tanaka. Teddy: A sketching interface for 3d freeform design. *SIGGRAPH 99*, pages 409–416, August 1999.

[121] Katsushi Ikeuchi and Takeo Kanade. Automatic generation of object recognition programs. In *Proceedings of the IEEE*, 1988.

[122] Robert J. K. Jacob, Leonidas Deligiannidis, and Stephen Morrison. A software model and specification language for non-WIMP= user interfaces. *ACM Transactions on Computer-Human Interaction*, 6(1):1–46, 1999.

[123] C. Jacobs, A. Finkelstein, and D. Salesin. Fast multiresolution image querying. In *Proceedings of Siggraph 1995, Computer Graphics, Annual Conference Series*, pages 277–286, 1995.

[124] Robin Jeffries, James R. Miller, Cathleen Wharton, and Kathy M. Uyeda. User interface evaluation in the real world: A comparison of four techniques. In *Proceedings of CHI '91*, pages 119–124, 1991.

[125] D. L. Jenkins and R. R. Martin. Applying constraints to enforce users' intentions in free-hand 2-D sketches. *Intelligent Systems Engineering*, 1992.

[126] Alan Jepson and Richard Mann. Qualitative probabilities for image interpretation. In *Proceedings of IEEE ICCV*, 1999.

[127] Michael Johnston, Philip R. Cohen, David McGee, Sharon L. Oviatt, James A. Pittman, and Ira Smith. Unification-based multimodal integration. In Philip R. Cohen and Wolfgang Wahlster, editors, *ACL'97*, pages 281–288, Somerset, New Jersey, 1997. Association for Computational Linguistics.

[128] Michael I. Jordan and David E. Rumelhart. Forward models: Supervised learning with a distal teacher. *Cognitive Science: A Multidisciplinary Journal*, 16(3):307–354, 1992.

[129] Joaquim A. P. Jorge and Ephraim P. Glinert. Online parsing of visual languages using adjacency grammars. In *11th International IEEE Symposium on Visual Languages*, 1995.

[130] M. Kääriäinen. On active learning in the non-realizable case. In *Foundations of Active Learning Workshop at Neural Information Processing Systems Conference*, 2005.

[131] Levent Burak Kara and Thomas F. Stahovich. An image-based trainable symbol recognizer for sketch-based interfaces. In *Making Pen-Based Interaction Intelligent and Natural*, pages 99–105, Menlo Park, California, October 21-24 2004. AAAI Fall Symposium.

[132] Rangachar Kasturi, Sing T. Bow, Wassim El-Masri, Jayesh Shah, James R. Gattiker, and Umesh B. Mokate. A system for interpretation of line drawings.

IEEE Transactions on Pattern Analysis and Machine Intelligence, 12(10):978–992, October 1990.

[133] T. Kato, T. Kurita, N. Otsu, and K. Hirata. A sketch retrieval method for full color image databases - query by visual example. *11th IAPA International Conference on Pattern Recognition*, pages 530–533, 1992.

[134] Manolya Kavakli and John S. Gero. The structure of concurrent cognitive actions: A case study on novice and expert designers. *Design Studies*, 23(1):25–40, January 2002.

[135] Manolya Kavakli, Stephen A. R. Scrivener, and Linden J. Ball. Structure in idea sketching behaviour. *Design Studies*, 19(4):485–517, October 1998.

[136] Jerome P. Keating and Robert L. Mason. Some practical aspects of covariance estimation. In *Pattern Recognition Letters*, volume 3(5), pages 295–350. International Association for Pattern Recognition, 1985.

[137] Robert Krauss. Why do we gesture when we speak? *Current Directions in Pschological Science*, 7(54-59), 2001.

[138] Robin L. Kullberg. Mark your calendar! Learning personalized annotation from integrated sketch and speech. In *Proceedings of CHI Short Papers*, 1995.

[139] M. Lades, J.C. Vorbruggen, J. Buhmann, J. Lange, C. von der Malsburg, R.P. Wurtz, and W. Konen. Distortion invariant object recognition in the dynamic linkarchitecture. *IEEE Transactions on Computers*, 42(3):300–311, March 1993.

[140] James A. Landay, Jason Hong, Scott Klemmer, James Lin, and Mark Newman. Informal puis: No recognition required. *Sketch Understanding, Papers from the 2002 AAAI Spring Symposium*, pages 86–90, March 25-27 2002. description of things at berkeley, no recognition, all over the place.

[141] James A. Landay and Brad A. Myers. Interactive sketching for the early stages of user interface design. In *Proceedings of CHI '95: Human Factors in Computing Systems*, pages 43–50, May 1995.

[142] James A. Landay and Brad A. Myers. Sketching interfaces: Toward more human interface design. *IEEE Computer*, 34(3):56–64, March 2001.

[143] Edward Lank, Jeb S. Thorley, and Sean Jy-Shyang Chen. An interactive system for recognizing hand drawn UML diagrams. In *Proceedings for CASCON 2000*, page 7, 2000.

[144] Edward H. Lank. A retargetable framework for interactive diagram recognitiont. In *Proceedings of the Seventh International Conference on Document Analysis and Recognition (ICDAR'03)*, 2003.

[145] Eric Lecolinet. Designing GUIs by sketch drawing and visual programming. In *Proceedings of the International Conference on Advanced Visual Interfaces (AVI 1998)*, pages 274–276. AVI, ACM Press, 1998.

[146] Seong-Whan Lee. Recognizing circuit symbols with attributed graph matching. In H.S. Baird, H. Bunke, and K. Yamamoto, editors, *Structured Document Image Analysis*, pages 340–358, 1992.

[147] James Lin, Mark W. Newman, Jason I. Hong, and James Landay. DENIM: Finding a tighter fit between tools and practice for web site design. In *CHI Letters: Human Factors in Computing Systems, CHI 2000*, pages 510–517, 2000.

[148] James Lin, Mark W. Newman, Jason I. Hong, and James A. Landay. DENIM: An informal tool for early stage web site design. In *Video Poster in Extended Abstracts of Human Factors in Computing Systems: CHI 2001*, pages 205–206., Seattle, WA, March 31 - April 5 2001.

[149] Hod Lispon and Moshe Shpitalni. Correlation-based reconstruction of a 3d object from a single freehand sketch. *Sketch Understanding, Papers from the 2002 AAAI Spring Symposium*, pages 99–104, March 25-27 2002.

[150] Allan Christian Long. *Quill: a Gesture Design Tool for Pen-based User Interfaces.* EECS department, computer science division, U. C. Berkeley, Berkeley, California, December 2001.

[151] Allan Christian Long, James A. Landay, and Lawrence A. Rowe. “Those look similar!” issues in automating gesture design advice. *PUI*, November 15-16, 2001.

[152] Allan Christian Long, James A. Landay, Lawrence A. Rowe, and Joseph Michiels. Visual similarities of pen gestures. In *Proceedings of the CHI 2000 Conference on Human Factors in Computing Systems*, 2000.

[153] P. Louridas and P. Loucopoulos. A generic model for reflective design. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2000.

[154] Wei Lu, Wei Wu, and Masao Sakauchi. A drawing recognition system with rule acquisition ability. In *Proceedings of the Third International Conference on Document Analysis and Recognition*, volume 1, pages 512–515, 1995.

[155] James V. Mahoney and Markus P. J. Fromherz. Interpreting sloppy stick figures by graph rectification and constraint-based matching. *Fourth IAPR International Workshop on Graphics Recognition*, 2001.

[156] James V. Mahoney and Markus P. J. Fromherz. Handling ambiguity in constraint-based recognition of stick figure sketches. *SPIE Document Recognition and Retrieval IX Conference*, January 2002.

[157] James V. Mahoney and Markus P. J. Fromherz. Three main concerns in sketch recognition and an approach to addressing them. In *Sketch Understanding, Papers from the 2002 AAAI Spring Symposium*, pages 105–112, Stanford, California, March 25-27 2002. AAAI Press.

[158] Jennifer Mankoff, Scott E Hudson, and Gregory D. Abowd. Providing integrated toolkit-level support for ambiguity in recognition-based interfaces. In *Proceedings of the CHI 2000 Conference on Human Factors in Computing Systems*, pages 368–375, 2000.

[159] Nicolas Matsakis. Recognition of handwritten mathematical expressions. Master’s thesis, Massachusetts Institute of Technology, 1999.

[160] Microsoft. Tablet PC and the enterprise. *Microsoft White Paper*, 2002.

[161] Microsoft. With launch of tablet PCs, pen-based computing is a reality. *Press Pass: Information for Journalists*, 2002.

[162] T. Mitchell. *Machine Learning.* McGraw Hill, 1997.

[163] T. M. Mitchell. *Version Spaces: An Approach to Concept Learning.* PhD thesis, Stanford University, 1978.

[164] T. P. Moran and J. M. Carroll, editors. *Design Rationale Capture: Concepts, Techniques and Use.* Lawrence Erlbaum Associates, 1996.

[165] T.P. Moran and J.M. Carroll. Overview of design rationale, design rationale: Concepts, techniques, and use. *LEA Computers, Cognition, and Work-Series*, pages 1–19, 1996.

[166] Manoj Muzumdar. ICEMENDR: Intelligent capture environment for mechanical engineering drawing. Master's thesis, Massachusetts Institute of Technology, 1999.

[167] Brad A Myers, Richard G. McDaniel, Robert C. Miller, Alan S. Ferrency, Andrew Faulring, Bruce D. Kyle, Andrew Mickish, Alex Klimovitski, and Patrick Doane. The amulet environment: New models for effective user interface software development. In *IEEE Transactions on Software Engineering*, volume 23-6, pages 347–365, 1997.

[168] Mark W. Newman, James Lin, Jason I. Hong, and James A. Landay. DENIM: An informal web site design tool inspired by observations of practice. *Human-Computer Interaction*, 18(3):259–324, 2003.

[169] Matt Notowidigdo and Robert C. Miller. Off-line sketch interpretation. In *Making Pen-Based Interaction Intelligent and Natural*, pages 120–126, Menlo Park, California, October 21-24 2004. AAAI Fall Symposium.

[170] J. Odell, H.V.D. Parunak, and B. Bauer. Extending UML for agents. *AOIS Workshop at AAAI*, 2000.

[171] Department of the Army. *Staff Organizations and Operations*, volume Field Manual 101-5. United States Army, Washington, DC, 1997.

[172] Alice Oh, Rattapoom Tuchinda, and Lin Wu. Meetingmanager: A collaborative tool in the intelligent room. In *Student Oxygen Workshop*, 2001.

[173] Clark F. Olson. Probabilistic indexing for object recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 518–522, 17(5),518-522 (1995).

[174] Michael Oltmans. Understanding naturally conveyed explanations of device behavior. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, 2000.

[175] Sharon Oviatt, Phil Cohen, Lizhong Wu, John Vergo, Lisbeth Duncan, Berhard Suhm, Josh Bers, Thomas Holzman, Terry Winograd, James Landay, Jim Larson, and David Ferro. Designing the user interface for multimodal speech and pen-based gesture applications: State-of-the-art systems and future research directions. In *Human Computer Interaction*, pages 263–322, August 2000.

[176] S. E. Palmer. The effects of contextual scenes of the identification of objects. *Memory and Cognition*, 3:519–526, 1975.

[177] Inc PalmOne. Graffiti, 2004. http://www.palmone.com/us/products/input/.

[178] Stephen Peters. Using semantic networks for knowledge representation in an intelligent environment. *IEEE Pervasive Computing and Communications (PerCom 2003)*, 2003.

[179] J. Pittman, I. Smith, Phil Cohen, Sharon Oviatt, and T. Yang. Quickset: A multimodal interface for military simulations. In *Proceedings of the 6th Conference on Computer-Generated Forces and Behavioral Representation*, pages 217–224, University of Central Florida, 1996.

[180] Arthur Pope. Model-based object recognition: A survey of recent literature. *TR 94-04*, January 1994.

[181] D. Pugh. Designing solid objects using interactive sketch interpretation. *Computer Graphics*, 1992.

[182] Inc. Quest Software. JProbe. website, http://www.quest.com/jprobe, 2006.

[183] Anand Raghavan and Thomas F. Stahovich. Computing sketch reasonings by interpreting simulations. In *Tenth International ASME Conference on Design Theory and Methodology*, 1998.

[184] Henry L. Roediger and Jeffrey D. Karpicke. The power of testing memory: Basic rearch and implications for educational practice. *Association for Psychological Science*, 1:3:181–210, 2006.

[185] Dean Rubine. Specifying gestures by example. In *Computer Graphics*, volume 25(4), pages 329–337, 1991.

[186] Sudeep Sarkar and Kim L. Boyer. Quantitative measures of change based on feature organization: Eigenvalues and eigenvectors. *Computer Vision and Image Understanding*, 71(1):110–136, July 1998.

[187] Eric Saund. Finding perceptually closed paths in sketches and drawings. *IEEE Trans on PAMI*, 25(4):476–491, April 2003.

[188] Eric Saund, David Fleet, Daniel Larner, and James Mahoney. Perceptually supported image editing of text and graphics. In *Proceedings of UIST '03*, 2003.

[189] Eric Saund, James Mahoney, David Fleet, Dan Larner, and Edward Lank. Perceptual organization as a foundation for intelligent sketch editing. *Sketch Understanding, Papers from the 2002 AAAI Spring Symposium*, pages 118–125, March 25-27 2002. editing drawn strokes, grouping of lines, scanscribe.

[190] Eric Saund and Thomas P. Moran. Perceptual organization in an interactive sketch editing application. In *ICCV 1995*, 1995.

[191] Steve Sedaker and Burton Holmes. Tablet pc makers select wacom penabled technology for unique cordless, batteryless, pressure-pen input. *News Wacom*, November 7, 2002.

[192] Tachyon Semiconductor. More than a touch of improvement for touch-screen control. *A Tachyon Semiconductor White Paper*, 2001.

[193] Tevfik Metin Sezgin. *Sketch Interpretation Using Multiscale Stochastic Models of Temporal Patterns*. PhD thesis, Massachusetts Institute of Technology, May 2006.

[194] Tevfik Metin Sezgin, Thomas Stahovich, and Randall Davis. Sketch based interfaces: Early processing for sketch understanding. In *The Proceedings of 2001 Perceptive User Interfaces Workshop (PUI'01)*, Orlando, FL, November 2001.

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

[196] Michael Shilman, Hanna Pasula, Stuart Russell, and Richard Newton. Statistical visual language models for ink parsing. In *Sketch Understanding, Papers from the 2002 AAAI Spring Symposium*, pages 126–132, Stanford, California, March 25-27 2002. AAAI Press.

[197] FM Shipman and RJ McCall. Integrating different perspectives on design rationale: Supporting the emergence of design rationale from design communication. *Artificial Intelligence in Engineering Design, Analysis, and Manufacturing (AIEDAM)*, 11(2):141–154, 1997.

[198] S.J.B. Shum, A. MacLean, V.M.E. Bellotti, and N.V. Hammond. Graphical argumentation and design cognition. *Human-Computer Interaction*, 12(3):267–300, 1996.

[199] Hewlett Packard (iPAQ Mobile Solutions). Cross-platform communications with iPAQ pocket PCs. *A Hewlett Packard Company White Paper*, 2002.

[200] Thomas F. Stahovich. Sketchit: a sketch interpretation tool for conceptual mechanism design. Technical report, MIT AI Laboratory, 1996.

[201] Thomas F. Stahovich, Randall Davis, and Howard E. Shrobe. Generating multiple new designs from a sketch. In *AAAI/IAAI, Vol. 2*, pages 1022–1029, 1996.

[202] F. Stein and G. Medioni. Structural indexing: Efficient 3-d object recognition. *IEEE Transaction on Pattern Analysis And Machine Intelligence*, pages 125–125, 1992.

[203] Mark R. Stevens, Charles W. Anderson, and J. Ross Beveridge. Efficient indexing for object recognition using large networks. In *Proceedings of IEEE International Conference on Neural Networks*, 1997.

[204] George Stiny and James Gips. Shape grammars and the generative specification of painting and sculpture. In C V Freiman, editor, *Information Processing 71*, pages 1460–1465. North-Holland, 1972.

[205] David G. Stork. The open mind initiative. *IEEE Expert Systems and Their Applications*, pages 16–20, May/June 1999.

[206] Ivan B. Sutherland. Sketchpad, a man-machine graphical communication system. In *Proceedings of the Spring Joint Computer Conference*, pages 329–346, 1963.

[207] Barbara Tversky. What do sketches say about thinking? *Sketch Understanding, Papers from the 2002 AAAI Spring Symposium*, pages 148–151, March 25-27 2002.

[208] Olya Veselova. Perceptually based learning of shape descriptions from one example. In *Proceedings of the 2nd Annual MIT Student Oxygen Workshop*, 2002.

[209] Olya Veselova. Perceptually based learning of shape descriptions. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, 2003.

[210] Olya Veselova and Randall Davis. Perceptually based learning of shape descriptions. In *Proceedings of the Nineteenth National Conference on Artificial Intelligence (AAAI-04)*, pages 482–487, San Jose, CA, 2004.

[211] VoiceXML Forum, http://www.voicexml.org/specs/VoiceXML-100.pdf. *Voice eXtensible Markup Language*, 1.00 edition, March 07 2000.

[212] Christian von Ehrenfels. Über gestaltqualitäten. In *Vierteljaheresschrift für wissenschaftliche Philosophie*, 1890.

[213] Max Werthemeimer. *Productive Thinking*. Harper, New York, 1959.

[214] Patrick H. Winston. Learning structural description from examples. *Psychology of Computer Vision*, 1975.

[215] Bo Yu and Shijie Cai. A domain-independent system for sketch recognition. In *Computer graphics and interactive techniques in Australasia and South East Asia*, 2003.

[216] R. Zeleznik. Sketch: An interface for sketching 3d scenes. In *Proceedings of SIGGRAPH'96*, pages 163–170, 1996.

[217] Victor Zue and Jim Glass. Conversational interfaces: Advances and challenges. In *Proceedings of IEEE*, pages 1166–1180, 2000.