SHADY: A Shape Description Debugger for Use in Sketch Recognition

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

Abstract

Sketch recognition systems are currently being developed for many domains, but can be time consuming to build if they are to handle the intricacies of each domain. LADDER is a language for describing how domain shapes are drawn, displayed, and edited in a sketch recognition system for that domain. LADDER shape descriptions can be automatically translated into JAVA code to be compiled with a multi-domain sketch recognition system to create a domain specific sketch interface. In this paper we present SHADY, a graphical tool to aid in the creation and debugging of LADDER shape descriptions. SHADY allows sketch interface developers to enter new shape descriptions or debug previously created descriptions, finding both syntactic and conceptual bugs. SHADY checks to see whether a shape descriptions is over-constrained by allowing the developer to draw sample shapes and then indicating which constraints are not met. This paper also describes work in progress on debugging under-constrained descriptions by automatically generating near-miss shapes.

Introduction

Pen-based sketch recognition interfaces are increasingly common and are being built for a variety of domains, including UML class diagrams, flowcharts, finite state machines, and course of action diagrams. These interfaces provide a more natural interaction than the traditional mouse and palette tool, but can be quite time consuming to build if they are to handle the intricacies of each domain. Our philosophy proposes that rather than build a separate recognition system for each domain, we instead build a single, domain independent recognition system that can be customized for each domain. To build a sketch recognition system for a new domain, the developer would need only write a domain description, describing how shapes are drawn, displayed and edited. This description would then be transformed for use in the domain independent system. The inspiration for such a framework stems from work in speech recognition and compiler compilers, which have been using this approach with some success (Zue & Glass 2000; Costagliola *et al.* 1995).

In previous work, we have built LADDER, a symbolic language for describing how shapes are drawn, displayed, and edited in a domain (Hammond & Davis 2003). We have also built a translator, which takes LADDER shape descriptions and transforms them into shape recognizers, editing recognizers, and shape exhibitors for use in recognition, display, and editing of the domain shapes (Hammond & Davis 2004). The implementation of this translator and domain independent sketch recognition system serves to show both that such a framework is feasible and that LADDER is an acceptable language for describing domain information.

Although efforts are being made to make LADDER as intuitive as possible, LADDER shape descriptions can be difficult to describe textually. It is much more natural to draw a shape than type out a verbal description. Veselova and Davis have developed a program to generate descriptions automatically from a single drawn example (Veselova & Davis 2004). However, these automatically generated descriptions may be imperfect, since it may be difficult for the computer to perceive the intention of the developer. For instance, if the developer draws a square, the developer may intend something as specific as a square or as general as a rectangle or even a quadrilateral. Creating a hand-typed description may be more time-consuming, but the developer can be specific about the shape intended. The developer may also prefer a hand-typed description because she can describe the shape in a way that is most intuitive, (as many of the constraints are overlapping, shapes can be described in several different ways), and she may want to use intuitive variable names. However, hand-typed descriptions also may contain errors. We performed a user study where we asked 30 people to describe shapes using both their natural language and in a more structured language such as LADDER, and we found both versions to contain a number of errors. Developers may forget a constraint, allowing unintended shapes to be recognized, or they may add too many incorrect or conflicting constraints, and the intended shape may not be recognized.

The contributions of the paper is SHADY, a graphical debugging tool (shown in Figure 1) to help the developer debug an LADDER shape description. SHADY debugs improper syntax, over-constrained descriptions and under-constrained descriptions. This paper describes completed work on debugging over-constrained descriptions; we also describe work in progress on debugging under-constrained descriptions.

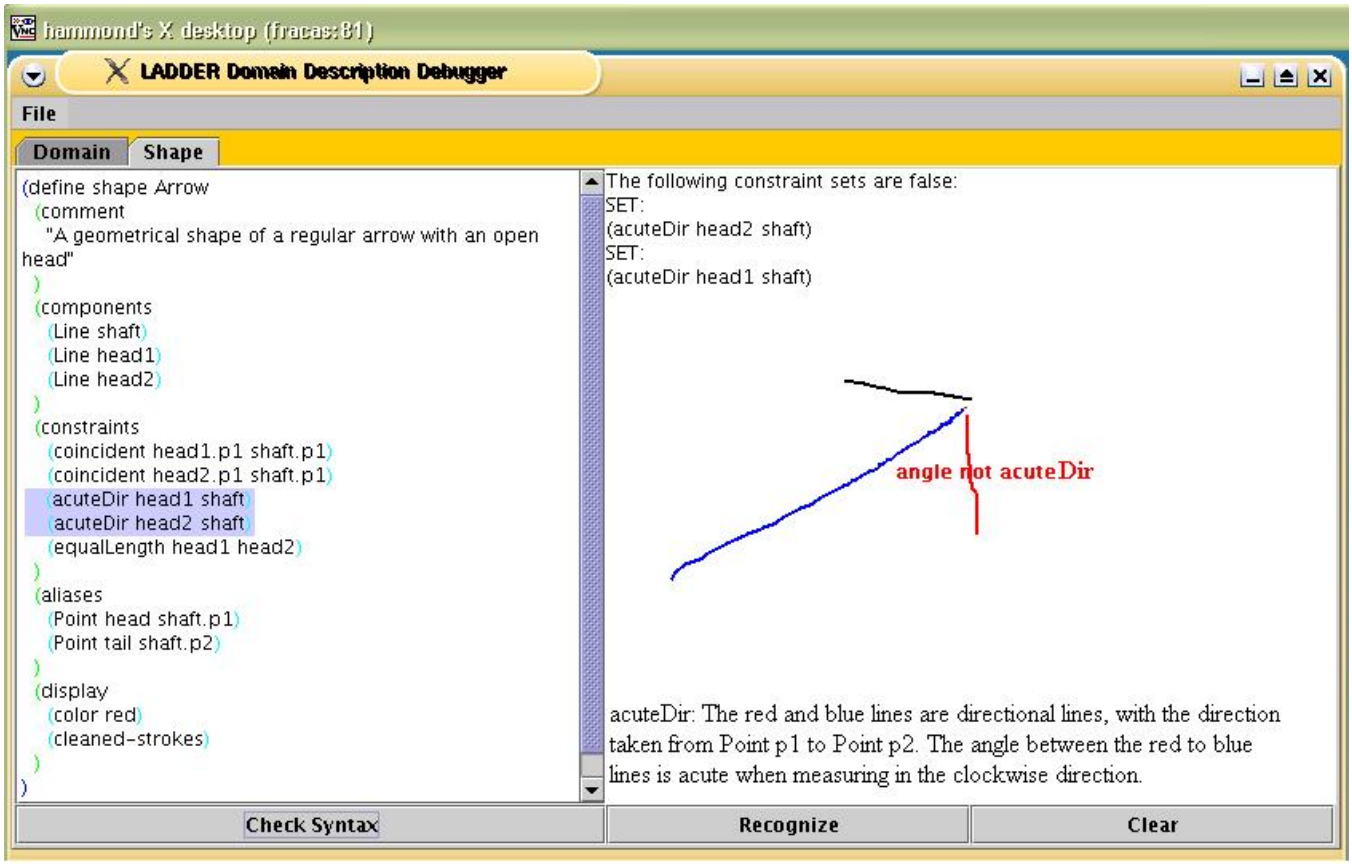

Figure 1: An incorrect arrow LADDER shape description being debugged by SHADY. The arguments in the second acuteDir constraint are incorrectly flipped. The constraint should read (acuteDir shaft head2), indicating that the counter-clockwise angle formed from the directional lines is acute.

```
(define shape Arrow
  (components
    (Line shaft)
    (Line head1)
    (Line head2))
  (constraints
    (coincident head1.p1 shaft.p1)
    (coincident head2.p1 shaft.p1)
    (acuteDir head1 shaft)
    (acuteDir shaft head2)
    (equalLength head1 head2)))
```

Figure 2: A correct arrow LADDER shape description.

Over-constrained Descriptions

We have created SHADY, a graphical tool (shown in Figure 1) to debug LADDER shape descriptions by determining if and how the description is over-constrained. If a shape description is over-constrained it will produce a false negative, i.e., a drawn shape that should have been recognized but was not. SHADY provides a draw panel for the developer to draw a positive example of the shape described. If the shape is not recognized based on the description given, SHADY highlights the failed constraint or constraints. The developer can then decide to remove or adjust the specified constraint(s).

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 2 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 3 presents 3 of the 48 possible assignments. Figure 3a and b give only one false constraint, where Figure 3c gives several false 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 their two endpoints assigned in 2 ways (2^3), giving the total possible number of assignment 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.)

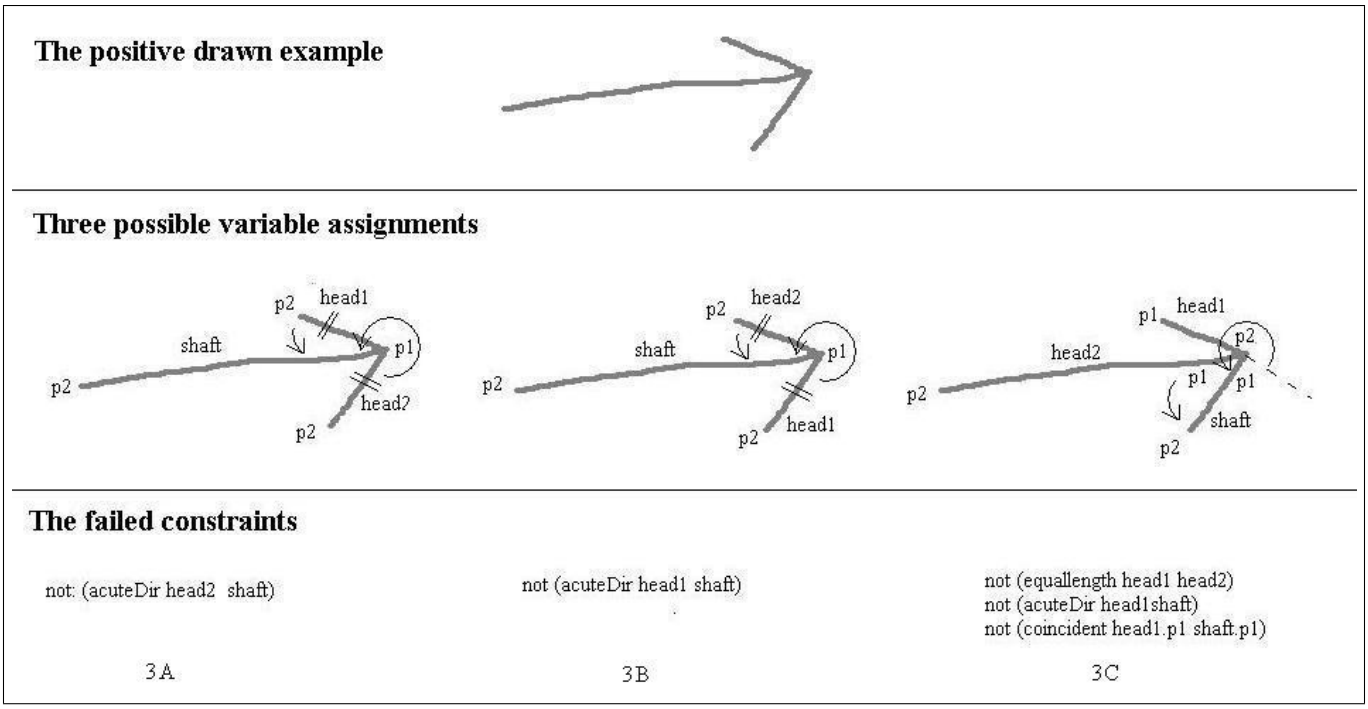

Figure 3: Three different variable assignments for an arrow with the over-constrained descriptions from Figure 1 and their failed constraints.

Selecting Variable Assignments

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

In Figure 1 a developer actually intended to describe an arrow as in Figure 2, but he mistyped one of the constraints. The arguments in the second acuteDir constraint are incorrect. The constraint should read (acuteDir shaft head2), indicating that the angle formed is acute if you 1) shift the lines to collocate their p1 endpoints, and then 2) travel in a counter clockwise direction from head2.p2 to shaft.p2.

We assume that the description given by the developer is mostly correct, and use Occham's razor [2] to select the variable assignments with the fewest failed constraints. If there are several variable assignments containing the minimum number of failed constraints, SHADY chooses all of them. In the case of Figure 3, SHADY chooses the variable assignments represented by a and b.

Displaying the Failed Constraints

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

SHADY lists both the collection of failed constraints for each chosen variable assignment, and it displays the failed constraints visually on the drawn shape. In Figure 3a and 3b, the failed constraints both represent the same angle between the same two lines. Thus SHADY displays that failed constraint only once. SHADY also explains the failed constraint in case the developer has misused it at the bottom of the screen. Figure 1 shows a screen shot of the system telling the developer which constraints have failed. Figure 4 provides another example of a more complicated hierarchically defined shape being debugged.

Under-constrained Descriptions

Above we discussed implemented work debugging over-constrained shape descriptions. The remainder of this paper describes work not yet complete on debugging under-constrained shape descriptions.

Once the shape description is determined not to be over-constrained, we want to test whether it is under-constrained, If so, drawn shapes other than the one intended will be recognized (giving false positives). To make sure the description is sufficiently constrained, SHADY will randomly generate several near-miss shapes and ask the developer whether the shape shown is an acceptable instance of the shape. Based on the answers given by the developer, SHADY will suggest the addition of constraints when appropriate.

[2] Occham's razor: "one should not increase, beyond what is necessary, the number of entities required to explain anything"

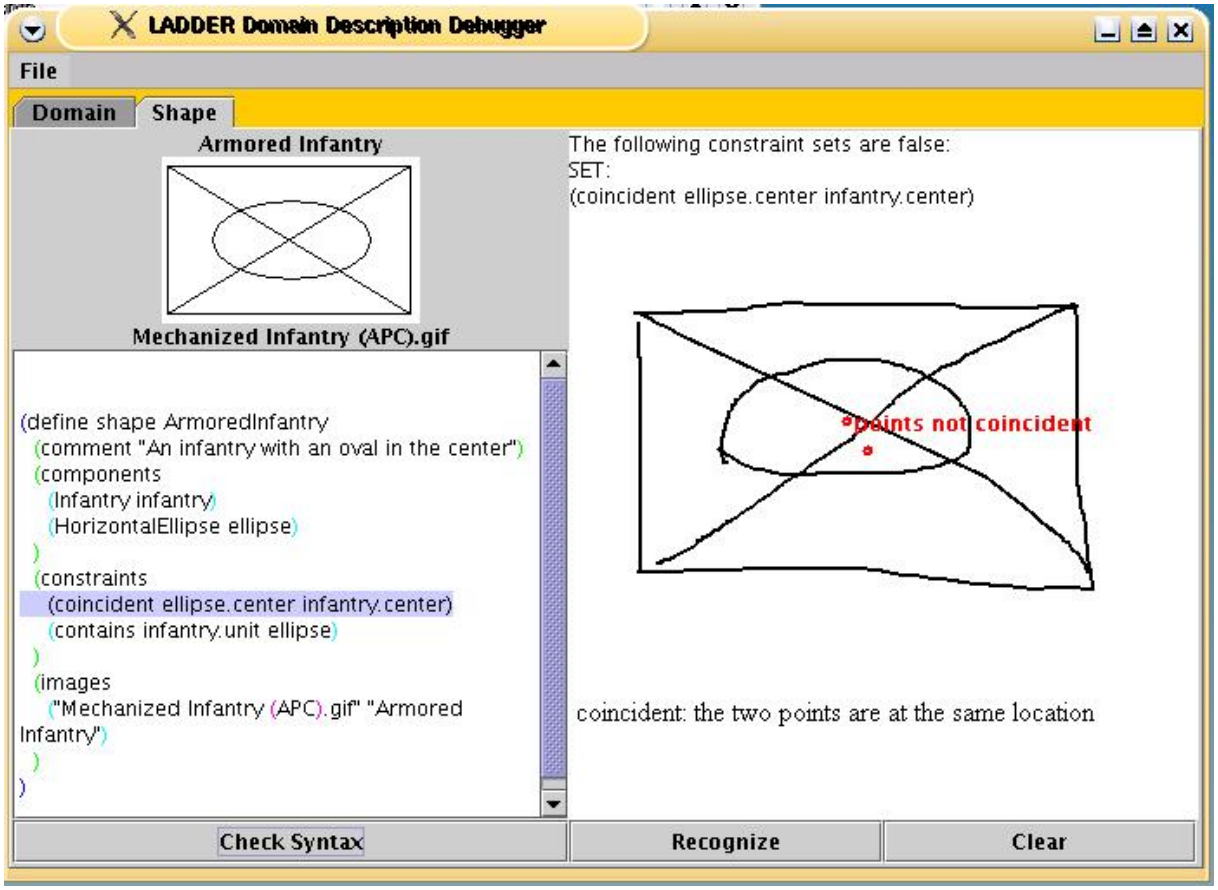

Figure 4: An armored infantry shape description from the domain of course of action diagrams being debugged with the debugger.

Selecting Additional Constraints

We want SHADY to generate near-miss shapes, and want to show as few shapes as possible to the developer to speed the debugging process, and as a result choose the extra constraints for testing wisely. Selecting extra constraints at random often simply collapses several points of the shape to a single location.

SHADY should first determine constraints that may be missing from the description. We propose to do this by having SHADY first examine a positive example provided by the developer. This could be a shape newly drawn for this purpose, or preferably, the system should reuse a previously drawn positive example such as one provided when testing whether the description is over-constrained.

The positive example should be used to automatically generate a description of the shape consisting of all of the possible true constraints for the shape drawn. The right side of Figure 5 shows several of the constraints that would be automatically generated for the description of the drawn arrow, which contains all true constraints from the drawn example.

SHADY should match the user-provided constraints of the arrow description (shown on the left side of Figure 5) to the automatically generated constraints. The first step is to find a variable-name matching between the two versions. We know that such a variable matching exists because both sets of constraints are based on the same drawn shape. There may be more than one possible variable matching, but it is enough to take the first one that works since the developer will not see the matching chosen. Once SHADY finds the variable-name matching, it should match the constraints. In Figure 5 the matched constraints are labelled with a circle.

After we match the constraints, there are still many constraints that don't actually provide any additional information beyond those of the originally matched constraints. For instance, acute is a less-constrained version of acuteDir. The redundant constraints have been labelled in Figure 5 with a rectangle.

Once SHADY eliminates all of the constraints already present in or redundant to the user-provided description, it should select which constraints were most likely omitted. We plan to group and order the remaining constraints according to their perceptual importance using techniques from (Veselova & Davis 2004). Connections (coincident, meet, etc.) have the highest perceptual importance, followed by orientations (horizontal, posSlope, acute, etc.). Relative lengths (equalLength, longer, etc.) have a low perceptual importance. Based on the ordering, several constraints

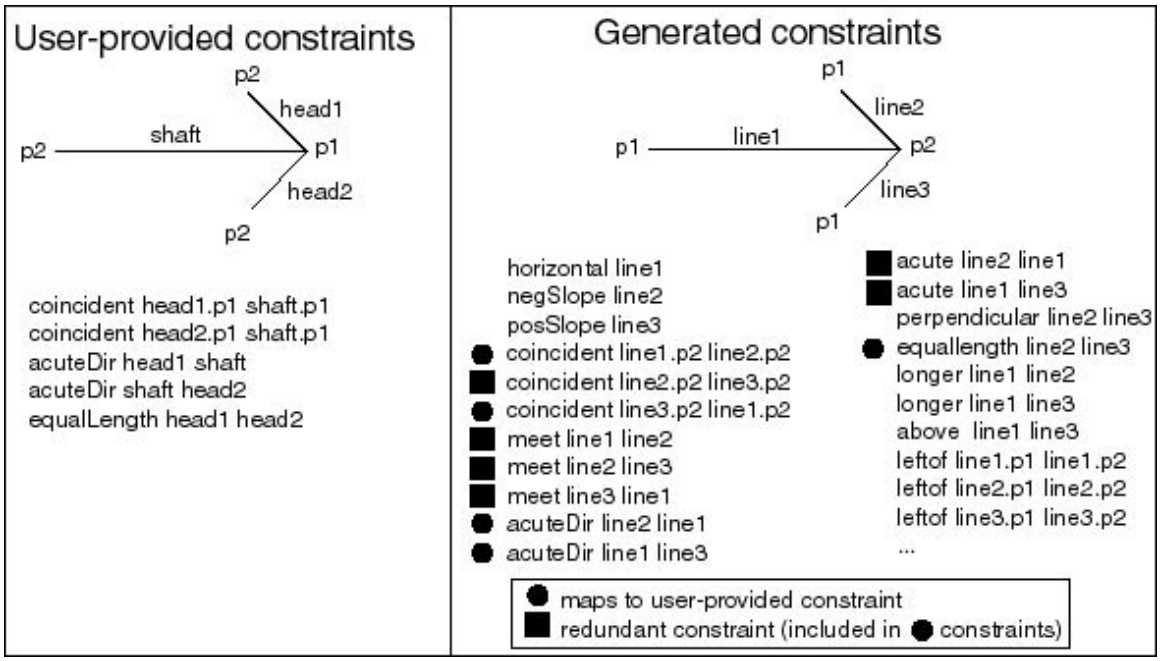

Figure 5: User-provided and generated constraints for an arrow.

should be chosen to determine whether they have been mistakenly left out of the description. We also want to make sure SHADY checks for constraints that are more likely to be forgotten, such as those of low perceptual importance.

When selecting the constraints to check for, it is important to note that we do not need to check for all of the remaining true constraints. Many of the constraints are related to each other because of the shape, and thus only one of the related constraints needs to be checked. For instance, (horizontal line1), (negSlope line2), and (posSlope line3) are all related constraints. When shifting line1 to not be horizontal as we did in Figure 6b, we caused the other two constraints to also be false. Thus, if the developer says that Figure 6b is an example of an arrow, we do not need to test the other two constraints.

Based on the generated constraints displayed in Figure 5, SHADY may chose to check if the following three constraints are missing: (perpendicular line2 line3), (horizontal line1), (longer line1 line2). To determine if these constraints are missing from the description, the system should generate three near-miss shapes (shown in Figure 6, one to test each of the suggested constraints. Perceptually the first two generated shapes, testing the constraints (perpendicular line2 line3) and (horizontal line1), are still arrows, implying that line2 and line3 don't have to be perpendicular and that line3 does not have to be horizontal. However, the last arrow with the short shaft is not an arrow, implying that we should add the constraint (longer line1 line2) to our shape description.

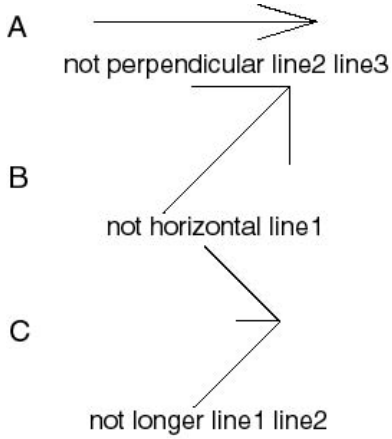

Figure 6: Possible near-miss shapes for an arrow.

Generating Shapes

For each extra constraint selected above, SHADY should automatically generate a shape that satisfies all of the user-provided constraints and does not satisfy the additional selected constraint. SHADY should then ask the developer whether the automatically generated shape is a positive example. We know that the added constraint was true in the drawn positive example. If the generated shape is a positive example, we do not want to add the constraint because it is false in a positive example (the generated shape) and true in another positive example (the original drawn shape). However, if the generated shape is a negative example, we do want to add the constraint.

SHADY will use a constraint solver to generate shapes. Each shape description specifies the components that make up a shape (for instance, an arrow is made up of three

lines). In principle, each component has an algebraic equation representing it. SHADY will use this same equation in its representation. For instance, the equation of a line is $Ax + By + C$. [3] Each constraint also has an algebraic representation, such as those represented in Figure 7. For example, if two points are coincident, then their x and y values are equivalent. All of the constraints are listed in Figure 7 with their integer x y values as inputs rather than the higher order points or shapes that are used in the shape description.

We are building a constraint solver specifically for use in sketch recognition that takes the algebraic equations represented by a shape description made up of solely of lines and reduces the expression to find the number of free variables. Creating non-linear constraints solvers (which would be used to solve higher order equations such as those for curves and arcs) is still a hard problem, thus we are limiting our constraint solver to solving linear constraints. There are also some constraints which may prove more difficult to solve for. For instance, constraints such as contains, which tests if one shape is inside of another, is difficult to specify by a simple equation. The final step for generating a shape is to specify random values for each of the free variables and display in for the user.

Re-evaluating Over-constrained Shapes

Once we complete our method for generating near-miss shapes to help determine whether a description is under-constrained, we hope to use the same method to determine whether a description is over-constrained. We think having SHADY generate several shapes will be an advantage over asking the user to draw several examples. First, we think it will be less work for the developer to simply acknowledge whether or not a shape generated is what they intended. Also, we can show several shapes simultaneously speeding the process. Second, it is quite possible that the developer may draw several shapes, but that the description is still over-constrained because the developer never draws a shape that conflicted with the over-constraint. By generating intuitive near misses, we hope that we can better refine the exact shape that the developer intends, requiring fewer clarifications by focusing the questions to the developer.

We plan to perform a user study to determine what constraints are commonly forgotten. SHADY will be able to use this knowledge to generate careful near miss shapes.

Related Work

Previous Work in our Group

This work is part of a larger effort to create a multi-domain recognition system. We created 1) a simple multi-domain recognition system (Hammond & Davis 2004), LADDER, a language for describing how shapes are to be recognized, displayed, and edited in a domain (Hammond & Davis 2003), and a code generator that translates a LADDER shape description into JAVA code and, when compiled along with the multi-domain recognition system, creates a domain-specific recognition system (Hammond & Davis 2004). Alvarado and Davis built a more sophisticated multi-domain recognition system that performs bottom-up and top-down

[3] We use an equation with 3 unknowns rather than the familiar $y = mx + b$ because we need to be able to handle vertical lines. Once we are able to determine whether or not the line is vertical, we are able to simplify the equation again to one with fewer unknowns using one of the additional rules, such as those listed at the bottom of Figure 7.

..

Line $(x_1,\ y_1)\ (x_2,\ y_2)$
$Ax_1 + By_1 + C = 0$
$Ax_2 + By_2 + C = 0$

coincident $(x_1,\ y_1)\ (x_2,\ y_2)$
$x_1 = x_2$
$y_1 = y_2$

equallength $(x_1,\ y_1\ x_2,\ y_2)\ (x_3,\ y_3,\ x_4,\ y_4)$
$(y_2 - y_1)^2 + (x_2 - x_1)^2 = (y_4 - y_3)^2 + (x_4 - x_3)^2$

angle $(x_1,\ y_1)\ (x_2,\ y_2)\ angle$
$(y_2 - y_1)/(x_2 - x_1) = tan(angle)$

dirangle $(x_1,\ y_1)\ (x_2,\ y_2)\ angle$
$atan2(y_2 - y_1, x_2 - x_1) = angle$
$(y_2 - y_1)/(x_2 - x_1) = tan(angle)$
$if (y_2 - y_1) then\ (0 < a < \pi)$

horizontal $(x_1,\ y_1)\ (x_2,\ y_2)$
$y_1 = y_2$

vertical $(x_1,\ y_1)\ (x_2,\ y_2)$
$x_1 = x_2$

acute $(x_1,\ y_1)\ (x_2,\ y_2)$
$(y_2 - y_1)/(x_2 - x_1) > 0$
$(y_2 > y_1\ \&\&\ x_2 > x_1) or (y_2 < y_1\ \&\&\ x_2 < x_1)$

obtuse $(x_1,\ y_1)\ (x_2,\ y_2)$
$(y_2 - y_1)/(x_2 - x_1) < 0$
$(y_2 < y_1\ \&\&\ x_2 > x_1) or (y_2 > y_1\ \&\&\ x_2 < x_1)$

acutedir $(x_1,\ y_1)\ (x_2,\ y2)$
$y_2 > y_1$
$x_2 > x_1$

obtusedir $(x_1,\ y_1)\ (x_2,\ y_2)$
$y_2 > y_1\ \&\&\ x_2 < x_1$

Additional Rules:
if $x_1 = x_2$, then $A = 1, B = 0, C = -x_1$
if $y_1 = y_2$, then $A = 0, B = 1, C = -y_1$
else $B = -1, A = (y_2 - y_1)/(x_2 - x_1)$,
$C = -y_1 - A * x_1$

Figure 7: Algebraic Equations for the linear constraints.

recognition with the help of Bayesian networks (Alvarado & Davis 2004). Sezgin develops techniques for improving recognition efficiency during the translation process (Sezgin 2003). Cates is improving the accuracy of low-level recognizers (Cates & Davis 2004). Other people in the project have explored adding speech and gesture recognition, respectively (Adler & Davis 2004; Eisenstein & Davis 2003).

Multi-Domain Recognition System Debugging

(Long 2001) has created a multi-domain recognition system where the developer can specify the shapes to be recognized in a domain by drawing them. The system helps the developer debug shapes by letting her know which shapes are similar and may be confused with other shapes causing recognition problems. (Long 2001) is solving a different problem than what is discussed in this paper in that they allow developers to debug the graphical vocabulary where ambiguity is a bug. (Gross 1996) have created a multi-domain recognition system, but they have no methods for debugging the shapes specified within them.

Geometric Constraint Solvers

A lot of work has been done on constraint solvers in general. The University of Washington created the Cassowary geometric constraint solver (Badros, Borning, & Stuckey 2001). Stahovich used an off the shelf constraint solver to solve mechanical engineering constraints to generate geometries from constraints (Stahovich 1996).

Other Future Work

A domain description consists of several shape descriptions, so it is also helpful to determine whether a description is similar to a previous description, to help debug shapes and prevent ambiguity. We would like SHADY to tell the developer if a shape description is equivalent or possibly can be confused with another description as in (Long 2001). Also, shape descriptions can be described hierarchically, making them simpler to understand. SHADY will tell the developer when a shape description is a sub- or super- shape description of another previously defined shape description. By comparing shape descriptions to other shape descriptions we can relax some of the shape constraints when the shape is easily distinguishable without it, which we hope will help to improve recognition in cases of very messy sketching.

Conclusion

We are building SHADY, a graphical tool to aid in the creation and debugging of LADDER shape descriptions, which are written by developers for use in a domain independent sketch recognition system. SHADY allows developers to type new descriptions and debug previously created descriptions, finding both syntactical bugs and conceptual bugs. SHADY checks to see if a description is over-constrained by allowing the developer to draw sample shapes and informing the developer which constraints are incorrect. This paper also describes work in process for debugging under-constrained shape descriptions.

Acknowledgements

The authors would like to thank Michal Karczmarek and Vineet Sinha for their help in reviewing this paper. The authors would also like to thank the members of the Design Rationale Group at MIT, including Michael Oltmans, Christine Alvarado, Jacob Eisenstein, Metin Sezgin, Aaron Adler, and Sonya Cates, for their help in discussing the ideas presented in this paper.

Research Support

This work is supported in part by the MIT/Microsoft iCampus initiative and in part by MIT's Project Oxygen.

References

Adler, A., and Davis, R. 2004. Speech and sketching for multimodal design. In *Proceedings of the 9th International Conference on Intelligent User Interfaces*, 214–216. ACM Press.

Alvarado, C., and Davis, R. 2004. Sketchread: A multi-domain sketch recognition engine. In *Proceedings of UIST '04*.

Badros, G. J.; Borning, A.; and Stuckey, P. J. 2001. The cassowary linear arithmetic constraint solving algorithm. In *ACM Transactions on Computer Human Interaction*, volume 8(4), 267–306.

Cates, S., and Davis, R. 2004. New approach to early sketch processing. In *AAAI Symposium : Making Pen-Based Interaction Intelligent and Natural*.

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

Eisenstein, J., and Davis, R. 2003. Natural gesture in descriptive monologues. In *UIST'03 Supplemental Proceedings*, 69–70. ACM Press.

Gross, M. D. 1996. The electronic cocktail napkin - a computational environment for working with design diagrams. *Design Studies* 17:53–69.

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

Hammond, T., and Davis, R. 2004. Automatically transforming symbolic shape descriptions for use in sketch recognition. *Proceedings of the Nineteenth National Conference on Artificial Intelligence (AAAI-04)*.

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

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

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

Stahovich, T. 1996. Sketchit: a sketch interpretation tool for conceptual mechanism design. Technical report, MIT AI Laboratory.

Veselova, O., and Davis, R. 2004. Perceptually based learning of shape descriptions. *Proceedings of the Nineteenth National Conference on Artificial Intelligence (AAAI-04)*.

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