

TSS: Tool for Sketching Statecharts

Shahla Almasri

School of Computer Science, McGill University
salmas1@cs.mcgill.ca

Abstract

Sketching is an important natural activity that helps people to quickly write their ideas before they forget them. Most current modeling tools constrain users to having to use traditional GUIs to draw diagrams (i.e. buttons, drop-down lists, etc.).

TSS is a sketch-based tool for drawing Statecharts diagrams. Its on-line recognition engine interprets diagrams as they are drawn. This paper presents the first iteration in the development of this tool. The motivation behind developing TSS is to explore the main challenges that are faced when developing a sketch-based modeling tool. Our research's long-term goal is to expand AToM³ to have a sketch-based meta-modeling tool.

1. Introduction

People tend to sketch their ideas before implementing them or even before using any computer-based tool. Sketching help us to brainstorm. Studies have shown that when people sketch freely, after a new idea is born most likely other ideas will follow. Vinod Goel [3] ran an experiment asking a group of designers to solve design problems using sketching with pen and paper, and asked another group of designers to solve problems using a computer-based drawing tool. He found that designers who used drawing tools spent most of their effort on working on the same idea. On the other hand, the group who sketched their ideas on paper were able to think more creatively.

Sketching is an important aspect of the modeling process. However, paper-based sketching lacks interactivity which makes it hard to modify diagrams as the model evolves [7]. Thus, modelers usually sketch their design on paper first and then they use a

computer-based tool to refine their original model and possibly generate a code from it¹.

My project aims at combining these two steps together in one tool. The idea is that as the modeler sketches their diagram using the tool, the diagram is automatically recognized. This approach will save modelers the time of entering their sketches into a computer-based tool.

The Modelling, Simulation, and Design Lab of the School of Computer Science at McGill University is developing a Tool for Multi-formalism and Meta-Modelling called AToM³ [4]. The goal of my research is to extend AToM³ to give modelers the flexibility of either sketching their diagrams, or using the traditional GUIs and their traditional interaction behavior.

As a proof of concept, I developed TSS (Tool for Sketching Statecharts). TSS is simply a prototype to demonstrate how diagrams can be recognized and processed. The Statecharts formalism [5] was chosen because it is simple enough to reveal the challenges of developing a sketch-based modeling tool without adding too much complexity to the problem. This paper describes TSS and outlines the limitations that can be faced when developing a sketch-based modeling tool.

2. Related Work

There are some sketch-based applications in areas like User Interface Design, Website Design, and Architecture. SILK [7] is a sketch-based tool for the early stages of user interface design. The idea is that the user interface designers would sketch their design using a stylus and it will be immediately ready for testing. A similar concept applies to DENIM [8], which is a sketch-based tool for website design.

¹ More studies need to be done to confirm this work model.

In the area of sketch-based modeling tools, there are some tools that support UML diagrams. Tahuti [10] is a tool that allows sketching UML Class Diagrams. It recognizes multi-stroke objects by examining their geometric properties rather than monitoring the way they were drawn. There is another tool developed at Queen's University (Kingston, ON) for sketching UML's Class Diagrams, Use Case Diagrams, and Sequence Diagrams [11]. There are also few commercial sketch-based UML tools like Ideogramic UML [12] and Tablet UML [13]. In addition to UML-related tools, there is also Sim-U-Sketch [9], which is an experimental sketch-based interface for MatLab's SimuLink.

Our approach is different in that it supports sketch-based meta-modelling and not just built-in formalisms.

3. Introduction to Statecharts

In the late 1980s, David Harel introduced Statecharts [5], which is an extension of State Transition Diagram. In 1997, Harel's Statecharts was adopted by UML for describing reactive behavior.

Harel introduced two new types of states in his new formalism: Composite and Orthogonal states. Composite states allow us to define hierarchy and thus allow viewing the system at different level of abstractions. On the other hand, orthogonal states add concurrency to the system, which can dramatically simplify a diagram. Harel also introduced the concept of broadcasting a message in an orthogonal state.

Unfortunately, in this first prototype of TSS, I am not supporting composite and orthogonal states. This is because I wanted to focus on the main challenges in developing a sketch-based application. Nevertheless, the second iteration of TSS will definitely support hierarchy and concurrency.

4. Implementation Details

TSS was written in Java (JDK 1.5) using SATIN's [1] framework. Although JDK 1.5 was used, TSS can be run on any machine that has JDK 1.4 or higher (this is a SATIN's requirement.) This is due to the fact that I did not use any of the new features in JDK 1.5.

4.1. SATIN

SATIN [1] is an open-source toolkit for building informal 2D pen-based applications. It was built using

Java JDK 1.3, and later JDK 1.4. It is a layer on top of Java Swing.

The architecture of SATIN handles pen input by providing recognizers, interpreters, and multi-interpreters. SATIN comes with Quill [2], which is a tool for designing gestures. SATIN's recognition engine uses Rubine's algorithm [14], however other recognition algorithms can be plugged in.

SATIN has support for manipulating and rendering objects. It also offers many of the common commands in pen-based applications, such as select, cut, copy, paste, delete, undo and redo.

The fact that SATIN is an open-source toolkit allows it to be easily extended and customized to meet any application's need.

4.2. Quill

Quill [2] is a training tool for designing gestures for pen-based applications. It facilitates the gesture design process by giving gesture designers suggestions on improving gestures so that they can be recognized more easily.

The Group of User Interface Research at University of California at Berkley (the group that developed Quill) wants to expand Quill to allow designers to use the gesture recognizer of their choice. However, for the time being, Quill only supports Rubine's recognizer [14]. Consequently, each gesture has to be trained by an average of 10-15 examples.

Quill saves gestures in a textual format that can be parsed later by SATIN or any other application.

4.3. Rubine's Recognizer

Rubine's recognizer [14] is a single-stroke feature-based recognizer. It categorizes gestures by measuring different features of the gesture. Some of these features are²:

- Length: The total length of the gesture.
- Distance between first and last points: The distance between the first and last points of the gesture.
- Total angle: The total amount of counter-clockwise turning. It is negative for clockwise turning.
- Total absolute angle: The total amount of turning that the gesture does in either direction.

² The explanation of the features is taken from Quill's Reference Manual.

- Cosine of the initial angle: This feature is how rightward the gesture goes at the beginning. This feature is highest for a gesture that begins directly to the right, and lowest for one that begins directly to the left. Only the first part of the gesture (the first 3 points) is significant.
- Sine of the initial angle: This feature is how upward the gesture goes at the beginning. This feature is highest for a gesture that begins directly up, and lowest for one that begins directly down. As in the previous feature, only the first part of the gesture (the first 3 points) is significant.
- Size of the bounding box: The length of the bounding box diagonal. The bounding box for a gesture is the smallest upright rectangle that encloses the gesture.
- Angle of the bounding box: The angle that the bounding box diagonal makes with the bottom of the bounding box.
- Cosine of angle between first and last points: This feature is the horizontal distance that the end of the gesture is from the start, divided by the distance between the ends. If the end is to the left of the start, this feature is negative.
- Sine of angle between first and last points: This feature is the vertical distance that the end of the gesture is from the start, divided by the distance between the ends. If the end is below of the start, this feature is negative.
- Sharpness: This feature is intuitively how sharp, or pointy, the gesture is. A gesture with many sharp corners will have a high sharpness. A gesture with smooth, gentle curves will have a low sharpness. A gesture with no turns or corners will have the lowest sharpness.

In order to be able to calculate a gesture's features, Rubine's algorithm needs to compare an average of 15 examples of that gesture with variations in size and directions.

5. Handling Strokes in TSS

As in all sketch-based applications, there are three types of strokes in TSS (see Figure 1), which are ink-objects' strokes, gestures' strokes, and regular ink

strokes. The next sub-sections below give detailed explanation of each type of these strokes.

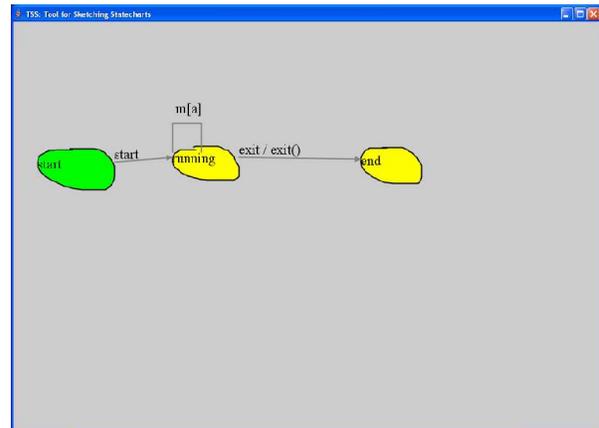


Figure 1: Snapshot of TSS

5.1. Ink-objects' Strokes

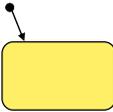
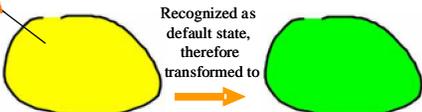
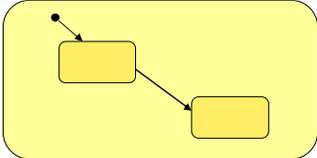
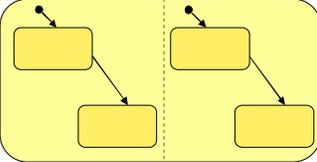
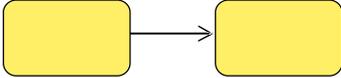
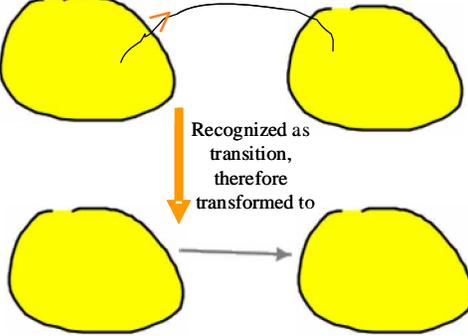
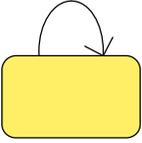
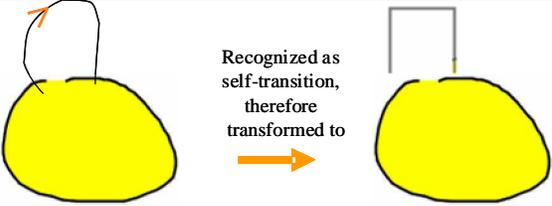
Ink-objects' strokes are the pen strokes that are recognized as objects and then processed and rendered on the canvas. There are three types of ink-objects in TSS: states, transitions, and characters. Table 1 describes the different strokes for drawing Statecharts' diagrams in TSS and Table 2 describes the strokes for drawing characters supported in TSS.

As shown in Table 1, states in Harel's notation are represented by rounded rectangles. In TSS, any thing that looks like a shape is considered to be a state. In other words, a rectangle, a circle, a triangle, or even a flower-like shape will all be recognized as a state in TSS. It is important to support this kind of ambiguity in sketch-based applications. We do not want to distract modelers by imposing strict rules on them. The goal is to give them the same freedom they would get when they use a pen and a paper. Internally, the shape of a state is stored as a polygon that is built from the points which constructed the stroke. When a shape is recognized as a state, it will be filled with yellow to indicate that.

Default states in Harel's notations are represented by a rounded rectangle with a short arrow pointing to them. In TSS, when a stroke starts from a point outside a state and it ends on a state, then that state will become a default state. The default state is filled with green, to distinguish it from other states. The user can change the default state at any time by drawing a line pointing to the new default state.

Transitions in both Harel's and TSS notations are represented by an arrow that starts from the source state and ends in the target state. If the transition is a

Table 1: Statecharts notations in TSS

Notation	Harel's	TSS'
States		
Default States		
Composite States		Not supported
Orthogonal States		Not Supported
Transitions		
Self-transition		
Labels	Any combination of characters	a - z, ., /, [, and] (Refer to Table 2 for more details)

self-transition then it is represented by a loop-arrow starting from and ending in the same state. In TSS, any stroke that starts in a state and ends in another will be recognized as a transition. Likewise, any stroke that starts and ends in the same state, and

and it has not been recognized as a character, will be recognized as a self-transition. We draw recognized transitions in gray to distinguish them from regular ink strokes.

Table 2: Characters supported in TSS

Character	Stroke
a	
b	
c	
d	
e	
f	
g	
h	
i	
j	
k	
l	
m	
n	
o	

Character	Stroke
p	
q	
r	
s	
t	
u	
v	
w	
x	
y	
z	
[
]	
/	
.	

5.2. Gestures' Strokes

Gestures' strokes are strokes that are recognized as gestures for commands. Example of these would be copy and paste commands. TSS makes use of some of the commands that are provided with SATIN. I had to modify some of those built-in commands in order for them to work in TSS; however I was able to use some others without any modifications. Also, some TSS-specific commands were added.

The following is a list of all TSS gestures in the order they are processed:

1. Hold-Select gesture:
 - a. Gesture: Clicking on an object and holding the mouse's left-button for few seconds.
 - b. Interpretation: The object that the user clicked on becomes selected.
2. Circle-Select gesture:
 - a. Gesture: Drawing a circle-like shape while holding the mouse's right-button. The interpreter does not try to recognize a circle, but instead it checks if portions near the beginning and end of the stroke intersect or are sufficiently near each other.
 - b. Interpretation: All objects inside that circle-like shape become selected.
3. Resize gesture:
 - a. Gesture: If an object is selected and the user clicked on one of the small squares around it then moved the mouse while the left-button is clicked.
 - b. Interpretation: Resize the selected object. At the present this command does not always work. There are few bugs that need to be fixed.
4. Move gesture:
 - a. Gesture: If an object is selected and the user clicked on it and moved the mouse while the left-button is down.
 - b. Interpretation: Move the selected object. Note that this command has been disabled for transitions. This is because a transition cannot be moved far away from its source and target states. Therefore I had the

option of either moving the whole diagram when a transition is being moved, or disallowing this command for transitions. I chose the latter because the former might be confusing to the user. When moving states, however, the associated transitions are updated accordingly. This is done by redrawing the transition's arrow when a state has been moved.

5. View-Port gesture:

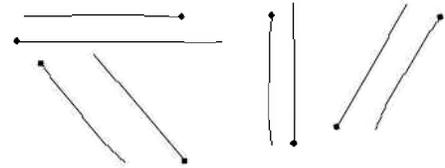


Figure 2: Scrolling Gestures

- a. Gesture: Drawing a line that looks like one of the lines in Figure 2, while holding the mouse's right-button.
- b. Interpretation: The window will be scrolled in the direction which the line was drawn in. The amount of scrolling corresponds to the length of the line.

6. Copy gesture:



Figure 3: Copy Gesture

- a. Gesture: Drawing a symbol similar to the one in Figure 3 while pressing the mouse's right-button.
- b. Interpretation: The object will be copied to the clipboard.

7. Cut gesture:



Figure 4: Cut Gesture

- a. Gesture: Drawing a symbol similar to the one in Figure 4 while pressing the mouse's right-button.
- b. Interpretation: The object will be copied to the clipboard and removed from the canvas.

8. Paste gesture:



Figure 5: Paste Gesture

- a. Gesture: Drawing a symbol similar to the one in Figure 5 while pressing the mouse's right-button.
- b. Interpretation: The object that is in the clipboard will be added to the canvas.

9. Delete gesture:



Figure 6: Delete Gesture

- a. Gesture: Drawing a symbol similar to the one in Figure 6 while pressing the mouse's right-button. This command is accessible through the keyboard's delete key as well.
- b. Interpretation: The object will be deleted.

10. Undo gesture:



Figure 7: Undo Gesture

- a. Gesture: Drawing a symbol similar to the one in Figure 7 while pressing the mouse's right-button.
- b. Interpretation: The last command will be undone and will be kept in memory in case the user wants to redo it later.

11. Redo gesture:



Figure 8: Redo Gesture

- a. Gesture: Drawing a symbol similar to the one in Figure 8 while pressing the mouse's right-button.
- b. Interpretation: Redo the last undone command.

12. Edit Label gesture:



Figure 9: Edit Label Gesture

- a. Gesture: Drawing a symbol similar to the one in Figure 9 while pressing the mouse's right-button.
- b. Interpretation: If this gesture is drawn over a state or a transition, a dialog will come up allowing the user to edit the label of that state or transition.

Commands 6 through 11 are also available from the pie menu. In addition to the commands mentioned previously, there are some commands that are only available from the pie menu, like Save, and Open. Choosing Save will save the diagram as an XML file. Similarly, choosing Open will parse the XML file and reconstruct the diagram.

5.3. Regular Ink Strokes

These are strokes which were neither recognized as ink-object strokes, nor as gesture strokes. In TSS, these types of strokes are drawn as ink without any kind of processing.

5.4. The Process of Interpreting Strokes

When a stroke is drawn, SATIN passes the stroke to the gesture interpreter first, to check if this is a command gesture or not. If it is, then it will be processed and marked as consumed so other interpreters will not see it. If it has not been

recognized as a gesture, then SATIN passes the stroke to the ink-object interpreter to check if this is a gesture for an object that we should recognize (i.e. state, transition, or letter.) If the stroke has not been consumed by neither the gesture interpreter nor the ink-object interpreter, then the stroke will be drawn as a regular ink stroke. Figure 10 below demonstrates this process:

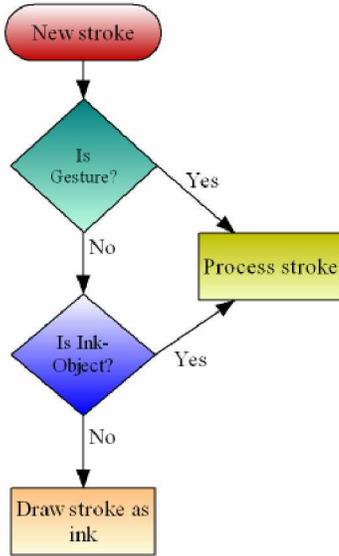


Figure 10: Strokes recognition process

In order to determine whether a stroke is a gesture or not, SATIN passes the gesture to all interpreters that were added to the *GestureInterpreter* in the order in which they were added. Figure 11 below illustrates what happens inside the “Is Gesture?” check in Figure 10 above.

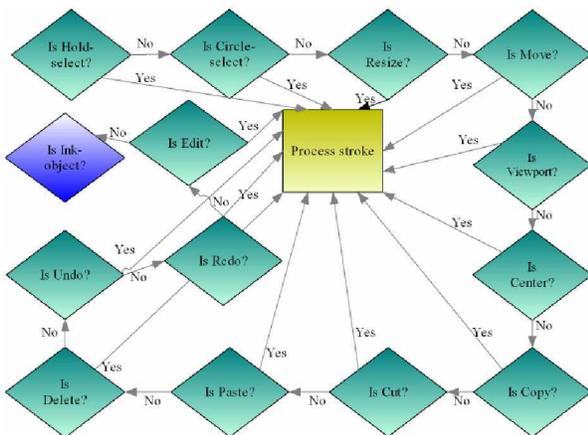


Figure 11: The process of gestures recognition in TSS

If the stroke was not recognized by any gesture’s interpreter, it is then passed to interpreters that were added to the *InkInterpreter*, by the order in which they were added. Figure 12 illustrates the details of the “Is Ink-object?” check in Figure 10.

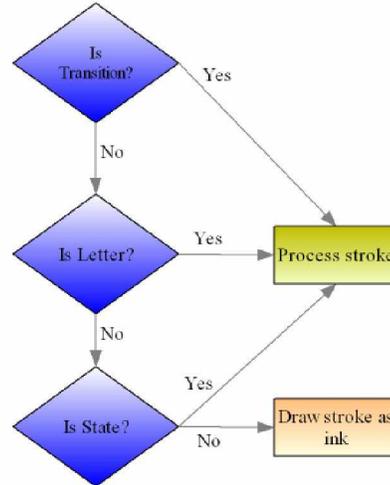


Figure 12: The process of ink-objects recognition in TSS

6. Diagram Validation

A Statecharts diagram in TSS is valid if it passed the following two checks.

6.1. Default State Checking

In Harel’s Statecharts [5], there has to be a default state at each level of abstraction and in each sub-diagram in an orthogonal state. However, the fact that TSS does not support hierarchy and concurrency as yet means that there must be only one default state in the diagram in order for it to be valid. The validation process can be illustrated by the model in Figure 13.

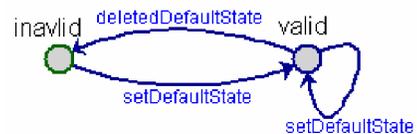


Figure 13: Validation of Statecharts Diagram in TSS

TSS will warn the user if they tried to save a diagram that does not have a default state (i.e. an invalid diagram.)

6.2. States' Labels Checking

According to Harel's [5] specifications, each state has to have a unique label in a level of abstraction. Furthermore, states must have a unique label in a sub-diagram of an orthogonal state. Once again, since we do not have composite and orthogonal states in TSS, then we only need to worry about labels' uniqueness at one level.

As in default state checking, TSS warns the user of duplicate labels of states when they try to save an invalid diagram. It also warns them if they left some states unlabeled.

7. Limitations

The major issue in TSS is its pattern recognition algorithm. As explained in Section 4, TSS uses SATIN's framework which uses Rubine's recognizer. Evaluation of other applications that use Rubine's recognizer, like SILK [7], DENIM [8], and Quill [2] has exposed similar recognition problems. On a second note, when evaluating Quill, studies have shown that users who are accustomed to using PDAs are better at drawing "good" gestures [2].

Consequently, further study of other recognition algorithms is needed. I need to explore the effectiveness of approaches like JavaSketchIt [16], SKETCHIT [17], SmartSketchpad [19], and the Novel Constraint-Based Approach [15]. I especially need to explore LADDER [18], which is a language for describing how sketched diagrams in a domain are drawn, displayed, and edited.

8. Future Work

As mentioned previously in Section 7 above, we need to improve the recognition engine in TSS by either finding an algorithm that fits our needs better than Rubine's recognizer, or by developing an algorithm on our own. Since we are not experts in the pattern recognition field, then we must go with the first option.

Moreover, as explained in Section 3, TSS lacks support for composite and orthogonal states. This is something that I need to address in our next iteration of TSS since it might reveal some problems that I am unaware of at this time.

I would also like to start integrating TSS with AToM³ [4], as this is the final goal of this project.

9. Conclusion

In this paper, I discussed the importance of sketching and how users who use traditional drawing tools that do not support sketching usually end up spending most of the time working on refining the same idea [3]. Thus what modelers often do is that they sketch their diagrams on papers first and then they redraw them using a modeling tool.

Our research proposes having a sketch-based meta-modelling tool and TSS is our first step in that direction. In this first iteration of TSS, I have come across some limitations in the recognition engine that SATIN uses, which is Rubine's algorithm. In the Limitations section (Section 7) I highlighted some alternative approaches to Rubine's recognizer and suggested further exploration of these algorithms.

My future plan is to integrate my work into AToM³ [4] to enable meta-modelers to benefit from a sketch-based interface.

10. Acknowledgements

Thanks to Professor Hans Vangheluwe who came up with the idea of developing a sketch-based AToM³ and offered guidance on approaching the problem. Also thanks to Rana Ali Adeeb who offered to proofread this paper and gave valuable comments on it.

11. References

- [1] J.I. Hong and J.A. Landay, "SATIN: A Toolkit for Informal Ink-based Applications", *Proceedings of the 13th annual ACM symposium on User interface software and technology*, ACM Press, November 2000, pp. 63-72.
- [2] A.C. Long, Jr., J.A. Landay, and L.A. Rowe, "Implications for a Gesture Design Tool", *Proceedings of the SIGCHI conference on Human factors in computing systems*, ACM Press, May 1999, pp. 40-47.
- [3] V. Goel, *Sketches of thought*, MIT Press, Cambridge, MA, 1995.
- [4] J. de Lara and H. Vangheluwe, "AToM³: A Tool for Multi-formalism and Meta-modelling", *Lecture Notes in Computer Science vol:2306*, Springer-Verlag GmbH, April 2002, pp. 174-188.
- [5] D. Harel, "Statecharts: A visual formalism for complex systems", *Science of Computer Programming vol:8 issue:3*, Elsevier North-Holland Inc, June 1987, pp. 231-274.

- [6] J.A. Landay and B.A. Myers, "Sketching Interfaces: Toward More Human Interface Design", *IEEE Computer vol:34 num:3*, March 2001, pp. 56-64.
- [7] J.A. Landay, *Interactive Sketching for the Early Stages of User Interface Design*, Ph.D. dissertation, Report #CMU-CS-96-201, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA., December 1996.
- [8] M.W. Newman, J. Lin, J.I. Hong, and J.A. Landay, "DENIM: An Informal Web Site Design Tool Inspired by Observations of Practice", *In Human-Computer Interaction vol:18 num:3*, Lawrence Erlbaum Associates, 2003, pp. 259-324
- [9] L.B. Kara, and T.F. Stahovich, "Sim-U-Sketch: a sketch-based interface for SimuLink", *Proceedings of the working conference on Advanced visual interfaces, Session: Designing better visual interfaces*, ACM Press, 2004, pp. 354-357.
- [10] T. Hammond, and R. Davis, "Tahuti: A Geometrical Sketch Recognition System for UML Class Diagrams", *The 2002 AAAI Spring Symposium on Sketch Understanding*, The AAAI Press, 2002, pp. 59-66.
- [11] E. Lank, J.S. Thorley, and S. Jy-Shyang Chen, "An interactive system for recognizing hand drawn UML diagrams", *Proceedings of the 2000 conference of the Centre for Advanced Studies on Collaborative research*, IBM Press, November 2000, pp. 7.
- [12] K.M. Hansen, and A.V. Ratzer, "Tool support for collaborative teaching and learning of object-oriented modeling", *SIGCSE Bulletin, vol:34, num:3*, ACM Press, September 2002, pp. 146-150.
- [13] M.L. Shoemaker, "Introducing Tablet UML", <http://www.tabletuml.com/IntroducingTabletUML.aspx>.
- [14] D. Rubine, "Specifying gestures by example", *Proceedings of the 18th annual conference on Computer graphics and interactive techniques*, ACM Press, July 1991, pp. 329-337.
- [15] L. Yan, G. Hunag, L. Yin, and L. Wenyin, "A Novel Constraint-Based Approach to Online Graphics Recognition", *Lecture Notes in Computer Science. vol:3138*, Springer-Verlag, August 2004, pp. 104-113.
- [16] A. Caetano, N. Goulart, M. Fonseca and J. Jorge, "JavaSketchIt: Issues in Sketching the Look of User Interfaces", *Proceedings of the 2002 AAAI Spring Symposium - Sketch Understanding*, AAAI Press, March 2002, pp. 9-14.
- [17] C. Calhoun, T.F. Stahovich, T. Kurtoglu, and L.B. Kara, "Recognizing multi-stroke symbols", *Proceedings of the 2002 AAAI Spring Symposium - Sketch Understanding*, AAAI Press, March 2002, pp. 15-23.
- [18] T. Hammond, R. Davis, "LADDER: A Language to Describe Drawing, Display, and Editing in Sketch Recognition", *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence*, Morgan Kaufmann, August 2003, pp. 461-467.
- [19] L. Wenyin, X. Jin, Z. Sun, "Sketch-Based User Interface for Inputting Graphic Objects on Small Screen Devices", *Lecture Notes in Computer Science vol:2390*, Springer-Verlag, June 2003, pp. 67-80.