Visual Syntax Analysis for Calligraphic Interfaces

Joaquim A. Jorge, Manuel J. Fonseca
INESC-ID/IST/UTL
R. Alves Redol, 9, 1000-029 Lisboa
jaj@inesc-id.pt, mjf@inesc-id.pt

Filipe M Garcia Pereira
IST/UTL
Lisboa
fmgp@mega.ist.utl.pt

Abstract

This paper presents an approach to analyse sketches of user interfaces by means of Visual Languages. To this end we use a shape recognizer, CALI [Fonseca00], and build on the ideas presented in JavaSketchIt[Caetano02] where the parsing is done by recognizing pairs of sketched symbols using a restricted visual syntax. The present paper presents our ongoing work on Visual Language Analysis to expand the possibilities afforded by JavaSkecthIt in a more flexible manner. Our approach provides more flexibility with the ability to describe an increasing number of visual symbols, and making use of more complex productions which wouldn't be so simple to handle with previous pattern recognition techniques employed in JavaSketchIt, such as recursive productions and ambiguous arrangements involving more than two symbols.

Keywords

Calligraphic Interfaces, Visual Parser, Visual Syntax Parsing, Visual Grammar.

1. INTRODUCTION

A desire common to developers, programmers and especially interface designers, is to pick up those hand drawn sketches and drafts and magically create the interface they've been so carefully designing. However, the majority of commercially available tools do not support this. To overcome these problems we are working on calligraphic interfaces. These focus on sketch and gesture recognition and thus allow users to sketch the outline of user interfaces on a tablet PC, instead of entering a sequence of commands and menu selections.

In this paper we present an approach based on parsing sketches using visual grammars. Our technique draws on earlier research work, which it attempts to generalize. After a brief overview of related research we describe our approach from a few selected examples.

2. RELATED WORK

Until recently, there have been different approaches to specify visual syntax and parse visual sentences into machine symbols. One approach is DENIM [Lin01] (an evolution from SILK [Landay95]) to simulate interfaces from sketches in a storyboard environment. Designers are able to sketch low-fidelity prototypes and dynamically simulate their behaviour. However, there are no beautifications or elaborate translation steps in the system.

JavaSketchIt [Caetano02] provides a framework to bridge the gap between conceptual mock-ups and functional prototypes. Shapes are recognized using a lexicon provided by CALI [Fonseca00]. User interface widgets can be recognized by combining primitive shapes using spatial relations and geometric attributes. These constitute a visual language that cannot be expanded since it is embedded in the application code.

SketchiXML [Coyette04] is another system based on JavaSketchIt and aimed at user interface prototyping. SketchiXML uses a multi-agent platform to perform the different tasks associated with parsing. Individual tasks are performed by distinct agents. One agent recognizes input scribbles, while others support removing redundancy and ambiguity. It is also possible to include other agents to perform extra tasks such as beautifying and aligning recognized widgets, as well as suggesting improvements to layout and design.

3. OUR APPROACH

To extend JavaSketchIt, we aim at developing a method for visual syntax analysis which should be both simple and general enough to allow users to define their own visual languages. Such visual languages are to be specified by means of a grammar using a set basic symbols and composition rules, using a textual language such as XML which should be easy to modify. While JavaSketchIt was a first attempt at this, the limitations inherent to the programming language used (Scheme) which resulted in syntax rules becoming embedded in the program code and thus difficult to modify and extend. Moreover, *widgets* (production rules) in JavaSketchIt could only rewrite at most groups of two symbols. Our aim is to provide a mechanism both strong and general enough to allow users to define productions with any number of symbols. Also it should be possible to include recursive productions by repeating on the right-hand-side of a production the same symbol specified in the left hand-side.

To specify widgets using sketches we need to recognize sets of strokes as geometrical entities and then to associate these entities using a visual grammar. The terminal alphabet allowed for the possible visual languages correspond to the scribbles recognized by CALI as shown in Figure 1.

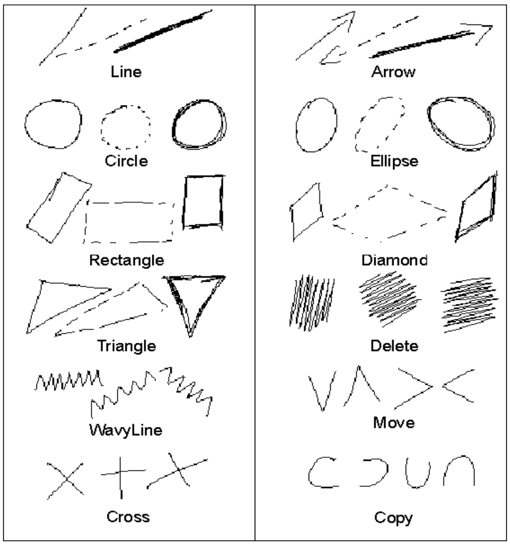

Figure 1: Scribbles recognized by CALI[Fonseca00]

Using this vocabulary, we define a set of rules based on spatial relations applied to constituent symbols. Remember from Visual Languages that it is spatial and topological relations that specify constraints rather than sequence [Jorge94]. Spatial constraints are defined either as unary or binary. Unary rules are fuzzy predicates that assert conditions on individual items such as `isHorizontal(Symbol)`. Binary rules are used to denote spatial relationships involving two visual elements. One such example is the binary fuzzy spatial relation predicate `isInside(Symbol1,Symbol2)` which allows the parser to check whether `Symbol1` is Inside `Symbol2`. Fuzzy predicates [Jorge94] differ from regular Boolean predicates in that they return a *degree of likelihood* between 0 and 1, rather than a crisp value of either true or false depending on whether the arguments satisfy (or not) the predicate.

By working out a simple example, these ideas can be better understood. Let's begin by providing a sample grammar definition for a simple widget (`TextField`), which is depicted by an horizontal line inside a rectangle. We use XML to specify grammars, as this provides a very flexible and easy to understand textual notation.

```
<widget type="TextField">
  <symbol type="Line" id="1"/>
  <symbol type="Rectangle" id="2"/>
  <rule name="isHorizontal" arg1="1"/>
  <rule name="isInside" arg1="1" arg2="2"/>
</widget>
```

In the example above, **`widget`** specifies a production in our grammar. Visual tokens correspond to `symbol` markers. Each **`symbol`** recognized by CALI will have a **`type`** assigned to it. We use positional identifiers to associate each symbol with arguments to constraints or **`rules.`** In principle any number of symbols or rules in a production.

Figure 2 shows the architecture of our system. The input is provided by strokes entered with a tablet/stylus combination. We use CALI to recognize sets of temporally contiguous strokes (scribbles) which are classified as one of the shapes shown in Figure 1.

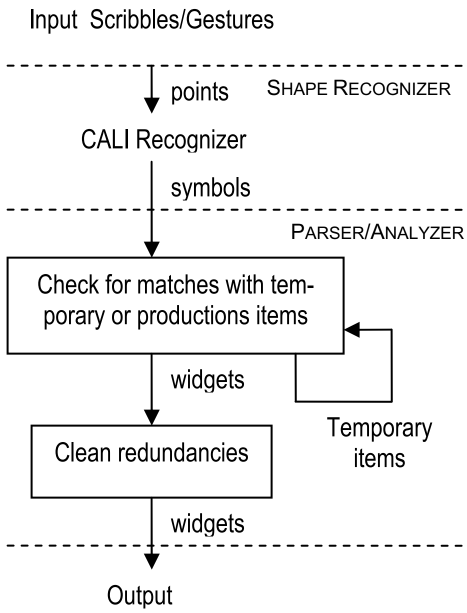

Figure 2: System architecture

Recognized scribbles are sent to a bottom-up parser. Since the order of constituents in a production is irrelevant, we need to search for items to associate using spatial relations and to use a set of heuristics to handle ambiguous productions. Our parser works by constructing partial items. These result from associating productions in the specified grammar with tokens captured by the recognizer. Incomplete *parse items* are stored in a Temporary Widgets List (`TWL`). When a new symbol is recognized by CALI, it gets checked against all elements in the `TWL`, generating new items as partial matches occur. New tokens are also checked against candidate productions in the grammar and may yield complete or incomplete items. A complete item is an instance of a production in which all symbol slots are filled. In the current system, complete items correspond to widgets and get stored in the results list. In contrast, incomplete items still require one or more symbols. A final set of heuristics cleans up both the temporary (`TWL`) and results list eliminating redundant items. The pseudo code below illustrates the way our parsing approach works:

```
Parse(scribble)
   newSym ← Recognize(scribble)
   for each item in TWL
      if item.match(newSym) then
         nItem ← item.reduce(newSym)
         if nItem.isComplete() then
            results.add(nItem)
         else
            TWL.add(nItem)
   for each prod in Grammar
      if prod.match(newSym, item) then
         nItem <- prod.reduce(newSym)
         if nItem.isComplete() then
            results.add(nItem)
         else
            TWL.add(nItem)
   CleanRedundantWidgets()
```

These ideas can be better understood by looking at a simple example. Consider the production `TextField` previously described. Figure 3 shows a possible sketch of a `TextField`.

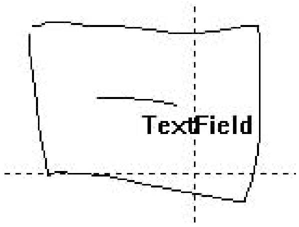

Figure 3: Sketch of a TextField widget (dashed lines are provided for context)

There are two different ways to draw this widget: either we can draw first the `Line` and then the `Rectangle` or the opposite sequence. For simplicity, let us begin with the Rectangle as show in Figure 4. The visual token recognized by CALI gets matched against all items in TWL. There is one incomplete `TextField` widget with a missing `Line` slot.

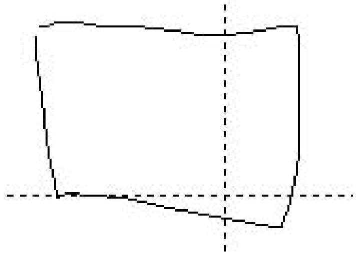

Figure 4: A Rectangle scribble

After the scribble is recognized by CALI as a `Rectangle`, we look in TWL for matching items. However the TWL is initially empty. We then check the Grammar for new entries. We find a matching `TextField` widget and generate the corresponding temporary item. Since the item is not complete (we need an horizontal `line`) we add it to TWL.

Finally, we draw a `Line` inside the `Rectangle` as shown in Figure 5. After it gets recognized by CALI we check the resulting shape against all items in the `TWL`. It matches the `TextField` item described above.

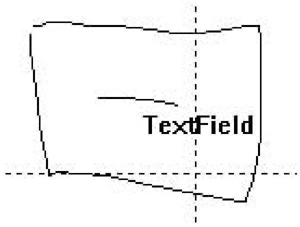

Figure 5: Sequence of two strokes

Since the temporary item thus found verifies both rules as checked by `item.match()`, i.e. `Line` is horizontal and is inside the `Rectangle` symbol in `item`, we reduce a new `item`. This `item` is complete. Thus, it gets added to `results` list, meaning that we have recognized a `TextField` widget. `CleanRedundantWidgets()` gets called afterwards to check whether there are temporary elements in TWL that are no longer needed. It finds the incomplete `TextField` item and checks this against the contents of results. Since the incomplete item matches a completed parse item in results, it gets deleted from TWL.

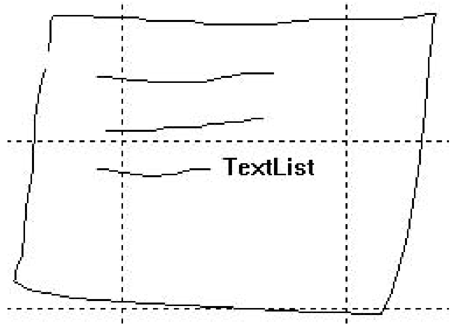

Figure 6: The `TextList` widget

Let's examine a more complex example, using a longer production reule involving also lines and rectangles. The visual representation for the widget `TextList` is shown in Figure 6. The grammar fragment below shows two productions one describing a simple `Text` widget (an horizontal line) and the other a `TextList` widget (three horizontal lines inside a rectangle):

```
<widget type="Text">
  <symbol type="Line" id="1"/>
  <rule name="isHorizontal" arg1="1"/>
</widget>
<widget type="TextList">
  <symbol type="Line" id="1"/>
  <symbol type="Line" id="2"/>
  <symbol type="Line" id="3"/>
  <symbol type="Rectangle" id="4"/>
  <rule name="isHorizontal" arg1="1"/>
  <rule name="isHorizontal" arg1="2"/>
  <rule name="isHorizontal" arg1="3"/>
  <rule name="isInside" arg1="1" arg2="4"/>
  <rule name="isInside" arg1="2" arg2="4"/>
  <rule name="isInside" arg1="3" arg2="4"/>
</widget>
```

We describe how the parser handles the input leading to the desired widget. Focusing on one of the possible permutations, we draw the following elements for this example in the sequence `Line(1)`, `Line(2)`, `Line(3)` and `Rectangle` as scribbles recognized by CALI.

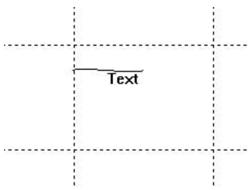

Figure 7: A `Text` widget parsed from the input of the horizontal "Line"

Figures 7 to 10 illustrate this sequence. Let us start by looking at `Line(1)`. At this step the parser checks to see if `Line`(1) is a valid entry for any of the temporary generated symbols and then the production rules in the grammar. Given that it matches all the input constraints for a `Text` widget, it is recognized as such, even though it may originate more temporary items to be recognized as more input is provided.

At this stage the predicate that has been asserted so far `isHorizontal(Line(1))` which causes the recognition of the target widget. With this entry, there isn't any visible progress: the temporary data grow, as well as the previous temporary elements gain more and more information leading them closer to form a complete widget. Figure 8, shows the

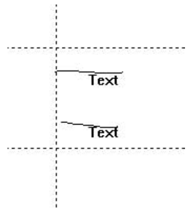

Figure 8: Two `Text` widgets parsed from input of both horizontal `Line` symbols

Figure 8 shows the state after two lines are drawn. Again what is seen and is possible to evaluate given the last input is still `isHorizontal(Line(1))` on the target widget, even though now we gathered two symbols already.

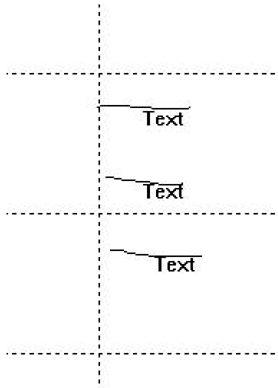

Figure 9: Three `Text` widgets parsed from input of all horizontal `Line` symbols

With this entry, again, there isn't really any visible progress, though the temporary data grow as temporary objects gather enough information to allow the next entry to generate the desired widget:

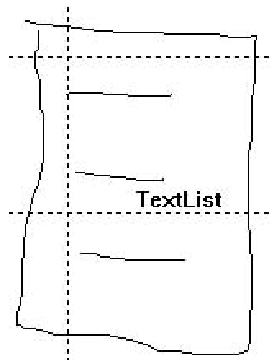

Figure 10: Three `Text` widgets parsed from input of all horizontal `Line` symbols and a rectangle

Figure 10 shows the complete input. The result is achieved when the `Rectangle` is provided to the temporary object list for evaluation. The parser checks that this is a valid entry and most important, it validates the missing rules that have been on hold which are `isInside(Line,Rectangle)` for each `Line` previously provided. We now have a complete item and can assert that a `TextList` was recognized.

4. HEURISTICS USED

Given the parsing techniques described above, the remaining issues are redundancy and ambiguity.

The redundancy is most notably seen when generating temporary objects from grammar productions due to possible permutations for more complex widgets. An example occurs in the `TextList` where we have three lines which obey to the same exact rules and in the same position when parameters for binary rules. Associating items in pairs would lead to many redundant temporaries. The resulting idea was to identify such occurrences from the start and when checking for valid entries, identify only one of these pairs as relevant and not consider others.

Ambiguity arises when recognizing and defining which widgets are complete. This causes immediately the appearance of multiple widgets with a different number of pieces gathered among the input symbols. From the example in Figure 10, we could have identified three occurrences of `Text` and at least one occurrence of `TextList`. We solve this by picking the widget with the largest count of input pieces and keeping the others as alternative results.

5. FUTURE WORK

There are some obvious limitations to the current status of the project. One example is container widgets. For these the recognition process might cause collisions with the widgets inside them and generate undesired results, such as an exponential growth in the number of temporary items. This will be dealt with in the near future.

Another limitation comes from the apparent simplicity of the input language/grammar file, which is only a first stage, given that it still wouldn't allow defining the widget of Figure 10 in the way shown below:

```
<widget type="Text">
  <symbol type="Line" id="1"/>
  <rule name="isHorizontal" arg1="1"/>
</widget>
<widget type="TextList">
  <widget type="Text" id="1"/>
  <widget type="Text" id="2"/>
  <widget type="Text" id="3"/>
  <symbol type="Rectangle" id="4"/>
  <rule name="isInside" arg1="1" arg2="4"/>
  <rule name="isInside" arg1="2" arg2="4"/>
  <rule name="isInside" arg1="3" arg2="4"/>
</widget>
```

The main differences between this and the previous examples lie in that the latter production references *non-terminal* items (`widgets`) instead of symbols. Indeed currently we can only parse a conceptual two level hierarchy of syntax rules.

As a final remark, this approach still lacks the mechanisms to suggest when to align groups of widgets, or to allow the users to create their own symbols or even learn design patterns from the users. These problems require a top-down approach while what we have implemented so far is a bottom-up technique. We plan to address these issues in the near future.

6. REFERENCES

[Caetano02] Caetano, A., Goulart, N., Fonseca, M. and Jorge, J.: JavaSketchIt: Issues in Sketching the Look of User Interfaces. In Proceedings of the 2002 AAAI Spring Symposium - Sketch Understanding, pages 9-14, Palo Alto, USA, 2002. http://immi.inesc.pt/publication.php?publication_id=40

[Coyette04] Adrien Coyette, Stéphane Faulkner, Manuel Kolp, Quentin Limbourg, Jean Vanderdonckt, "SketchiXML: Towards a Multi-Agent Design Tool for Sketching User Interfaces Based on UsiXML". In: Proc. of 3rd Int. Workshop on Task Models and Diagrams for user interface design TAMODIA'2004 (Prague, November 15-16, 2004), Ph. Palanque, P. Slavik, M. Winckler (eds.), ACM Press, New York, 2004, pp. 75-82. http://www.isys.ucl.ac.be/bchi/publications/2004/Coyette-TAMODIA2004.pdf

[Fonseca00] M.Fonseca and J.Jorge. CALI: A Software Library for Calligraphic Interfaces. INESC-ID, available at http://immi.inesc-id.pt/projects/cali/, 2000. http://immi.inesc.pt/projects/cali/papers/9epcg2k.pdf

[Jorge94] Joaquim A Jorge, Parsing Adjancency Grammars for Calligraphic Interfaces, Phd Thesis, Rensselaer Polytechnic Institute, Troy, NY, 1994.

[Lin01] James Lin, Mark W. Newman, Jason I. Hong, James A. Landay, "DENIM: An Informal Tool for Early Stage Web Site Design." Video poster in Extended Abstracts of Human Factors in Computing Systems: CHI 2001, Seattle, WA, March 31-April 5, 2001, pp. 205-206. <http://dub.washington.edu/projects/denim/pubs/denim-chi-2001-video.pdf>

[Landay95] James A. Landay and Brad A. Myers, "Interactive Sketching for the Early Stages of User Interface Design." In Proceedings of Human Factors in Computing Systems: CHI 95, Denver, CO, May 1995, pp. 43-50. <http://www.cs.berkeley.edu/~landay/research/publications/storyboard-tr/storyboard.html>