JavaSketchIt2 – Using Relational Adjacency Grammars for Visual Syntax Parsing

Filipe M Garcia Pereira
IMMI/INESC-ID/IST/UT
R. Alves Redol, 9, 1000Lisboa
fmgp@mega.ist.utl.pt

Manuel João Fonseca
IMMI/INESC-ID/IST/UTL
R. Alves Redol, 9, 1000 Lisboa
mjf@inesc-id.pt

Joaquim A. Jorge
IMMI/INESC-ID/IST/UTL
R. Alves Redol, 9, 1000 Lisboa
jorgej@acm.org

Abstract

This paper presents a project which expands on JavaSketchit [8], which parsed sketches of user interfaces, identifying gestures by using CALI [6], and then used topological relations and relational constraints [4] to recognize pairs of gestures as widgets in a Java graphical user interface. JavaSketchit was able to export the functional source code for the prototype in Java. Our approach allows users to write down their own visual language and removes the existing limitation of one pair of gestures per widget. The resulting parser became applicable in different contexts besides designing user interfaces. The result is a functional implementation of a parser designed to handle Relational Adjacency Grammars customized by the user..

Keywords

Calligraphic Interface, Visual Parser, Visual Syntax Parsing, Visual Grammar, Relational Adjacency Grammar, Widgets, Gestures, Sketches.

1. INTRODUCTION

Every programmer, computer engineering student, software engineer or interface designer has a natural desire to have some simple and immediate way to translate sketches and interface drafts from paper to functional source code. This concept would always apply to any given programming language they're working on in their projects. To overcome this obstacle, some researchers have been working on calligraphic interfaces for translating sketches to functional interface prototypes.

In many cases the calligraphic interfaces focus on recognizing individual strokes under an application context. In contrast, the approach in this paper presents a parser that takes the sketched shapes one by one, and uses them to recognize widgets according to sets of rules which are represented by topological relations.

2. RELATED WORK

DENIM[4] is among the first to allow the creation of interface prototypes. This was achieved by use of a pattern recognizer to identify both commands and strokes made by the user. The DENIM[4] project is also relevant because it allows multiple pages and a zooming view for the creation of storyboards for the user interface behavior. A great feature in this project was its ability to export the pages created by the user to HTML. The major limitations to DENIM[4] were the problems which were present when the pattern recognizer would not allow a sketch to be used correctly or when the recognizer would correct a stroke not meant to be corrected.

Another approach was JavaSketchit [5]. This used the CALI [3] recognizer to identify which geometrical shape was the closest match to each user drawn sketch, and used the shapes as input tokens. Each shape was crossed with all previously drawn ones to identify which pairs represent widgets and then decide which to keep in the interface prototype. A great feature was that this project would export the prototype to fully functional Java. Still, the project had limitations. These were the fact the user couldn't choose which widgets were part of the visual language, or what gesture pairs to use in each widget and which spatial relations to use.

3. THEORETICAL BACKGROUND

Grammars have been a common concept present in logic since early computation concepts and became a root topic in computational sciences. Usually grammars are described as (N, T, P, S), a quadruple where N is the set of non-terminal symbols, T is the set of terminal symbols, P is a set of valid productions and finally S is the starting symbol. A simple example of a regular grammar definition would be the one presented in Fig. 1 where all the sets and productions are clearly defined.

This information is usually enough for most language concepts and applications, such as textual programming language parsers. However, for the approach presented here, regular grammars do not present a structure which holds enough information, in that they lack a way to specify the constraints needed to describe each non-terminal symbol in detail.

N = { Text, TextField, TextArea, ε }
T = { Line, Rectangle }
S = ε
P: ε → {Text | TextField | TextArea}
Text → {Line} if isHorizontal(Line)
TextField → {Text, Rectangle}
if isInside(Text, Rectangle)
TextArea → {Text, TextField}
if isInside(Text, TextField)
A = { isHorizontal(x), isInside(x,y)}

Figure 1: Example Relational Adjacency Grammar [2]

For example, visual languages are not able to describe spatial constraints such as those that indicate whether a sketch is inside another as in Fig. 3. Towards this goal, we adopted Relational Adjacency Grammars [2] (RAG). A RAG is defined as a quintuple (N, T, S, P, A) where N, T, S and P are the same as in regular grammars and A is a set of constraints which allow to keep the extra information required.

N = { A, B, S }
T = { a, b }
P = {(S→aA), (S→bB), (A→a), (B→b)}
S = S

Figure 2 – Example of a regular grammar definition

In Fig. 5 we demonstrate how this information is used to describe Figs. 2 through 4. With the extra information it becomes possible to unambiguously identify each widget.

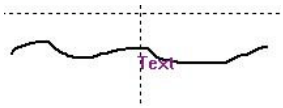

Figure 3 – A sketch representing Text.

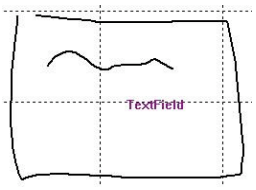

Figure 4 – Set of sketches that represent a TextField

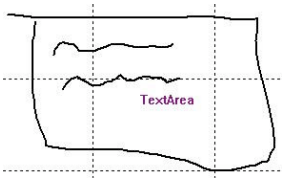

Figure 5: Set of sketches that represent a TextArea

There are modifiers other than "if" as shown in Fig. 5, though for this application only the "if" is utilized for the definition of widgets. Finally, establishing a bridge between the terms in use inside the application and the terms in the grammar:

- N represents the non-terminal symbols which are also known as widgets;
- T represents the terminal symbols which are known as gestures or sketches;
- S is the empty symbol for the reason that the templates hold no valid information until it is instantiated;
- P is the set of valid productions, which are stored as templates after read from a file;
- A is the set of rules which represents the topological relations that can be applied for the validation of widgets when new input is presented.

4. THE APPLICATION

As mentioned, this project focus on the implementation of a parser which uses a RAG to allow a better mechanism for the syntax analysis using sketched inputs, especially when there is no order when drawing the shapes. The architecture of the resulting application is shown in Fig. 6, where is visible how both the sets with non-terminal symbols (widgets) and the valid productions (templates) are supplied. The CALI [3] recognizer receives the sketched shapes and identifies them as gestures by application of fuzzy logic. As a result, the possible gesture results considered for this project are {Line, Arrow, Triangle, Rectangle, Diamond, Circle, Ellipse, Copy, Cross, Move, WavyLine}. After processing the sketch, the gesture with highest probability value is sent to the parser itself. The parser uses the gesture and tries to see if any of the temporary widgets can change its state, followed by generating new widgets from the templates. Finally, generates a list of complete widgets. These last are widgets that have gathered all the gestures they were supposed to collect, and validated all the respective rules These rules can be either relational constraints [2] or topological relations [1,2]. The topological relations considered are either adjacency relations, overlap relations, or metric relations. These cover basic concepts where one gesture is inside another, or intersecting each other, being "to the right" of a gesture, or "to the left", and were even considered all cardinal directions. For the metric relations were considered cases like two gestures having similar areas. In Fig. 7 is shown one example of a "grammar.xml" file where some rules are demonstrated for widget declaration.

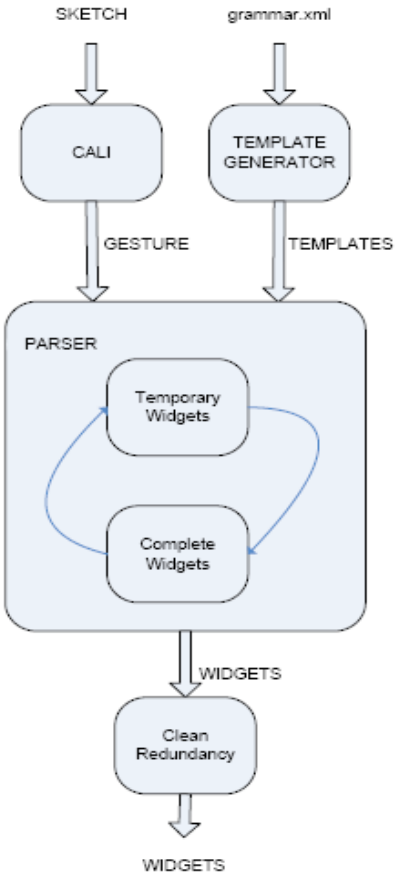

Figure 6: Architecture of the developed application

Notice still in Fig. 7 the presence of the `id` attribute which is used to assure order in how parameters are applied in rule validation. Also the distinction between `isTrue` and `isFalse` tag names for the rules allow to use the NOT operator.

To prevent false positives, routines are called to clean the global widgets lists, by removing widgets that are only parts of the larger ones, also by removing replicas and by ensuring that any gesture is only used once in the prototype.

The parser itself is based on the algorithm shown in Fig. 8, where the parsing mechanism is summarized. **`ParseIncludingCompletes`** takes the transformed input and uses it as input for both temporary widgets and templates. Visible in the code is that for each complete widget found, the routine will make a recursive call using that complete widget. This call allows the user to make use of a widget as piece for another widget declaration. One last feature is that each recursive call will create its own temporary widget list to avoid entering a state of infinite recursive calls.

There is one last task associated to the widget parsing, which is answering the question "which rules are needed to validate each input in a widget?". In result, a temporary widget can be a valid hypothesis or not. This task is performed in the **`addPiece`** routine as shown in Fig. 9.

```
<widgets>
 <widget type="Label">
  <symbol type="Line" id="1"/>
 </widget>
 <widget type="TextBox">
  <widget type="Label" id="1"/>
  <symbol type="Rectangle" id="2"/>
  <isTrue name="isInside"
          param1="1" param2="2"/>
 </widget>
 <widget type="Button">
  <symbol type="Rectangle" id="1"/>
  <symbol type="Rectangle" id="2"/>
  <isTrue name="isInside"
          param1="1" param2="2"/>
  <isTrue name="haveSimilarAreas"
         param1="1" param2="2"/>
 </widget>
</widgets>
```

Figure 7: Sample contents of the "grammar.xml" file

5. RESULTS

This project has not included usability tests and inquiries since these were the object of existing results reported from the previous project (JavaSketchit). The main task consisted on converting the previous project from Scheme and C to C++. Another task was to allow users to be able to define widgets for use in the parser by means of a grammar specified in external XML files, rather than having it embedded in the code. Also, effort was spent on making the parser work in real time thus allowing users to see widgets being identified while drawing.

The results in Table 1 show that our parser is able to recognize nontrivial elements in real time, which is desirable for interactive applications. The larger values for memory consumption and parsing time are result from having ellipses sketched in the prototype. This happens because they store the greatest amount of data. Despite that, what can be observed is the average parsing time taken per gesture keeps under 0.1s, which is a reasonable value for real time interaction with the developed application.

Table 1: Average results obtained during performance tests

#Gestures per Prototype	Parsing Time per Gesture (seconds)	Memory Consumption (Byte)
9	0,00819	63.984
14	0,01061	120.432
25	0,01890	201.288
35	0,07838	411.328
43	0,05584	335.672

```
parseIncludingCompletes(symbol newInput){
//Check temporary widgets
 if(!(temporaryWidgetList.empty())){
  for each tw in temporaryWidgetList do{
   elems<-tw.checkValidEntry( newInput );
   if( elems.size()>0 ){
    for each tag in elems do{
     newtw<-tw.clone();
     valid<-newtw.addPiece(newInput, tag);
     if( newTemp.isFinnished() && valid ){
      completes.push_back( newTemp );
      parseInclCompletes(nwtw.clone() );
     }
     else{
      if( valid ){
       temps.push_back( nwtw );
 }}}}}}
//Try to generate widgets from templates
 for each tw in templateWidgetList do{
  elems<-tw.checkValidEntry( newInput );
  if( elems.size()>0 ){
   for each tag in elems do{
    newt<-templateWidget.clone();
    valid<-newt.addPiece( newInput, tag );
    if( valid && newt.isFinnished() ){
     completes.push_back( newt );
     parseInclCompletes( newt.clone() );
    }
    else{
     if( valid ){
      temps.push_back( newTemp );
}}}}}}
```

Figure 8: **The main algorithm in the parsing process**

```
widget::addPiece(widget newWidget,
                 tagValue tag){
// The widget can have no rules at all
if(_Rules.empty()){
 missingPieces.remove(newWidget,tag);
 addedPieces.push_back(newWidget,tag));
 if(missingPieces.empty())finished<-true;
 valid<-true;
 return true;
}
//If the widget has rules, know what to
validate
list<rules> rulesToVerify;
for each Rule in localRulesList do{
 if(!Rule.isVerified()){
  if(Rule.canBeVerified(tag,addedPieces){
   rulesToVerify.push_back(Rule);
}}}
result<-true;
for each Rule in rulesToVerify do
 result <- result &&
    Rule.validate( newWidget, addedPieces );
if(result){
 addedPieces.push_back(newWidget,tag);
 missingPieces.remove(newWidget,tag);
 if(missingPieces.empty()){
  finish <- true;
  for each Rule in localRulesList do
   finish <- finish && Rule.isVerified();
 }}
valid<-result;
return result;
}
```

Figure 9: The addPiece algorithm

6. FUTURE WORK

The most relevant work missing in this project is to perform usability tests to gather recent results from real users.

Another interesting development would be a trainable interface to generate grammar elements and supply sets of valid productions to the parser without having to type the widget definitions.

There is also some work to be done to improve the efficiency of the algorithms developed as well as its lower level data structures, which store a lot of redundant information. These would be particularly relevant to allow the whole application viable to use in mobile devices.

7. ACKNOWLEDGMENTS

This work was supported in part by a grant from Portuguese Science Foundation (FCT) POSC/EIA/59022/2004

REFERENCES

[1] Max J. Egenhofer. A Formal Definition of Binary Topological Relationships. In W. Litwin and H. Schek, editors, Third International Conference on Foundations of Data Organization and Algorithms (FODO 89), volume 367 of Lecture Notes in Computer Science, pages 457 472. Springer-Verlag, Paris, France, June 1989.

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

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

[4] 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.

[5] 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