Generating Domain Specific Sketch Recognizers From Object Descriptions

Tevfik Metin Sezgin MTSEZGIN@AI.MIT.EDU
MIT Artificial Intelligence Laboratory, 200 Technology Square, Cambridge MA, 02139 USA

1. Introduction

Sketch understanding has been recognized as a vital component in intelligent design spaces and an enabling technology in natural human-computer interaction. Unfortunately there is little computer support for sketching. We describe a system that compiles domain specific knowledge about objects in a particular domain into recognizers for those objects, removing the need to handcode individual recognizers.

```
(define AndGate
  (components
    ( Line input_A )
    ( Line input_B )
    ( Line output )
    ( Line vertical_line )
    ( GeneralPath arc ) )
  (constraints
    ( parallel input_A input_B )
    ( above  input_A input_B )
    ... )
)
```

Figure 1. Typical input for our system.

2. System Description

As noted in (Davis, 2002), we are working on a generic multi-domain sketch recognition architecture. This sketch recognition system will differ from existing architectures in many aspects, including a language for describing shapes, mechanisms for learning new shapes, and a blackboard based recognition architecture with top-down and bottom-up recognizers. Here we describe a system, a part of the larger architecture, that takes descriptions of domain specific objects in a format as specified in (Hammond, 2002), and generates Java code that when compiled produces a separate class file representing each object in the domain. The generated code also functions as a bottom-up recognizer that acts as a knowledge source in the larger blackboard based architecture. This automatic code generation scheme removes the need to manually write a separate recognizer for each object in the domain.

2.1 Input

Fig. 1 illustrates how a typical input to our program looks like. The object description has a Lisp-like syntax. It specifies what primitives form the object, and the constraints that must be satisfied.

2.2 Output

Our system reads in the input file and generates a separate class for each object described. The generated class contains member fields corresponding to the components forming the object. Upon successfully recognizing an object, these fields are set to point to the actual primitives on the sketching surface. Fragments of the output for the object description in Fig. 1 can be seen in Fig. 2.

2.3 Code generation

The parser for reading the input files is written using javacc (Sun's Java compiler compiler). When an input file containing object descriptions is read, an abstract syntax tree is built. The AST has nodes for components, constraints of different types, and several other declarations such as renaming statements which provide a mechanism for referring to objects with a different name – similar to let in Lisp. The AST is then processed to generate Java code for individual recognizers. In addition to member fields corresponding to components of the object, the generated code includes a series of nested for loops and if statements for cycling through available strokes in the sketching surface, generating permutations of strokes and testing if any of these permutations satisfy the constraints. If a particular subset of the strokes in the sketching surface satisfies all the constraints, then a recognition is signalled to the blackboard. If only a subset of the constraints are satisfied, a *partial recognition* is generated as explained in the next section.

3. Generating Partial Recognitions

When the recognition routine can't find a full recognition but only a subset of the constraints are satisfied, we still need to notify the blackboard that there are a number of strokes that *almost* form a well-formed object but some components are missing. For example, if we are recognizing squares and we only have three edges that satisfy the

```
public class AndGateParser {
  Line horizontal_line1; // Members

  public void recognizeAndGate( SpatialDatabase d,
                                Rectangle focus, boolean cons[]){
    ArrayList input_A_instances; // Component instances
    ...
    Object objects[]
    objects = database.getObjects( focus );
    // Resuming partial recognition
    if ( !constraint0 || !constraint1 || !constraint2 ) {
      // Checking constraints
      for ( int i2 = 0; i2 < input_A_instances.size(); i2++ ) {
        for ( int i4 = 0; i4 < input_B_instances.size(); i4++ ) {
          if ( cons0 && cons1 && ... && cons8 ) {
            // Recognized a full object
            input_A = input_A_instance;
          } else {
            // Report partial recognition to the blackboard
          }
        }
      }
    }
  }
}
```

Figure 2. Structure of the Java source generated by our system.

constraints of being part of a square, we notify the blackboard of this fact so that other knowledge sources (for example the top-down knowledge sources) have this information. We define a partial recognition as a group of strokes that satisfy all the constraints that they participate in. A group of strokes participate in a constraint if the constraint refers to both strokes. For example, if a constraint declares that strokes A and B are supposed to be parallel, they are said to participate in this constraint. So in all partial recognitions including A *and* B, this constraint must be satisfied.

The concept of partial recognitions can be illustrated better as a graph problem. Imagine a graph where the components are represented as nodes, and the constraints between components are represented by edges. Furthermore, the edges carry truth values: true if the constraint is satisfied, false otherwise. Fig. 4 shows a situation for an object with five components. Green edges are satisfied constraints, and red edges are unsatisfied. In this setup, a partial recognition is a subgraph where all the edges between the nodes are compatible with each other (i.e. the constraints they participate in are satisfied). In this case, the partial recognition includes nodes with green borders. Simplified pseudocode for an algorithm computing partial recognitions is in Fig. 4. Here, the input arguments are respectively the current partial recognition, list of components to be considered and the list of components that are known to conflict with the current partial recognition.

4. Related Work

There are existing systems that use object descriptions for sketch recognition. Our system differs from those systems by actually generating Java code which can be compiled into Java binaries and executed efficiently as opposed to referring to the object description file during the sketching

```
ComputePR( List PR, List R, List P )
{
  if ( R is empty )
    return PR;

  choose neighbor node r from R
  if ( r.isCompatibleWith( PR ) ) {
    pr = ComputePR( PR+r, R-r, P )
  } else {
    pr = ComputePR( PR, R-r, P+r )
  }

  return pr;
}
```

Figure 3. Simplified pseudocode for the PR algorithm.

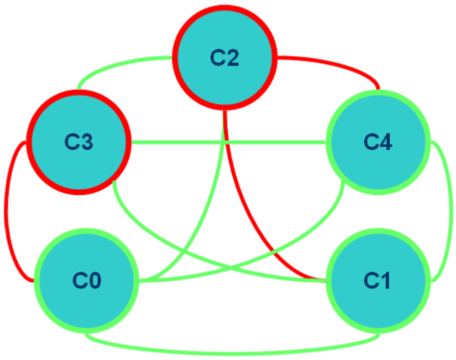

Figure 4. The partial recognition graph.

process dynamically. In addition Java's support for loading classes makes regenerating and reloading class files possible in a dynamic fashion without sacrificing the advantages of using compiled binaries. Finally, the partial recognition representation we introduce gives us a systematic framework to deal with partially drawn objects.

5. Future Work

Future directions include improving the speed and accuracy of the generated recognizers by incorporating knowledge about how objects are actually drawn by users. We are also looking at ways of automatically incorporating optimizations in the generated code.

Acknowledgements

I would like to thank my thesis advisor Prof. Randall Davis for his supervision.

References

Davis, R. (2002). Sketch understanding in design: Overview of work at the mit ai lab. *AAAI Spring Symposium: Sketch Understanding*.

Hammond, T. (2002). A domain description language for sketch recognition. *Proceedings of 2002 SOW*.