Developing Domain-Specific Gesture Recognizers for Smart Diagram Environments

Adrian Bickerstaffe, Aidan Lane, Bernd Meyer, and Kim Marriott

Monash University, Clayton, Victoria 3800, Australia

Abstract. Computer understanding of visual languages in pen-based environments requires a combination of lexical analysis in which the basic tokens are recognized from hand-drawn gestures and syntax analysis in which the structure is recognized. Typically, lexical analysis relies on statistical methods while syntax analysis utilizes grammars. The two stages are not independent: contextual information provided by syntax analysis is required for lexical disambiguation. Previous research into visual language recognition has focussed on syntax analysis while relatively little research has been devoted to lexical analysis and its integration with syntax analysis. This paper describes GestureLab, a tool designed for building domain-specific gesture recognizers, and its integration with Cider, a grammar engine that uses GestureLab recognizers and parses visual languages. Recognizers created with GestureLab perform probabilistic lexical recognition with disambiguation occurring during parsing based on contextual syntactic information. Creating domain-specific gesture recognizers is not a simple task. It requires significant amounts of experimentation and training with large gesture corpora to determine a suitable set of features and classifier algorithm. GestureLab supports such experimentation and facilitates collaboration by allowing corpora to be shared via remote databases.

1 Introduction

There has been considerable research into the automatic recognition of diagrams as the basis for smart diagrammatic environments (SDEs). These SDEs use structured diagrams as a means of visual human-computer interaction [10]. An example SDE is a smart whiteboard that automatically interprets, refines and annotates sketches jotted down in group discussion. Much of this research has focused on generic diagram interpretation engines based on incremental multi-dimensional parsers. Such parsers can automatically be generated from a grammatical specification of the diagrammatic language [5], greatly simplifying the task of implementing SDEs. The inputs to such a parser are lexical tokens such as lines, rectangles, or arrows. Typically the user composes a diagram from these with an object-oriented drawing editor.

Extending the (semi-)automatic generation of diagram interpreters to support sketching in pen-based environments is a challenging task and the focus of this paper. A generic two-stage approach is taken in which syntax analysis (parsing) is

W. Liu, J. Lladós, and J.-M. Ogier (Eds.): GREC 2007, LNCS 5046, pp. 145–156, 2008.

preceded by lexical analysis (gesture recognition). While some previous projects have used parsing techniques for lexical analysis, decades of research into pattern analysis suggests that feature-based and statistical methods are better suited to this problem [7]. The core challenges tackled in our paper are: (1) to automate as far as possible the development of statistical recognizers for stylus-drawn graphical tokens and (2) to integrate statistical lexical recognition with grammar-based syntax analysis.

The main contribution of this paper is to describe GestureLab, a tool for generating probabilistic gesture recognizers. GestureLab is integrated with Cider [5], a multi-dimensional parser generator for diagram analysis: gesture recognizers generated with GestureLab can be interfaced automatically with an incremental parser generated by Cider. Together these two systems provide a suite of generic tools for the construction of interactive sketch interpretation systems. These tools automate the SDE construction process to a high degree. The viability of the GestureLab-Cider approach is demonstrated in the development of a computer algebra system that interprets stylus-drawn mathematical expressions. The tool recognises algebraic and matrix notation from interactive input and demonstrates context-driven disambiguation.

GestureLab uses Support Vector Machines (SVMs) as the default mechanisms for learning new recognizers. SVMs are a popular approach to supervised learning of wide-margin classifiers because they are well-understood, theoretically well-founded and have shown excellent performance across a broad variety of applications [2,13]. However, standard SVMs perform non-probabalistic two-way classification. A second contribution of this paper is to describe an extension to SVMs that allows GestureLab to generate probabilistic k-way recognizers.

2 GestureLab

Given the huge variety of lexical tokens occurring in different types of diagrams, it is clear that generating an interpreter for a new diagram type requires lexical recognition to be tailored to the gestures of interest. GestureLab (see Fig. 1) is a software tool designed to facilitate rapid development and testing of such domain specific gesture recognizers. Recognizers can be developed entirely within GestureLab without any need for a testbed application and can be coupled to Cider without modification.

GestureLab recognizers follow the standard approach to statistical gesture recognition: recognition is performed on digital ink which includes position, timing, pressure, and angle data. Statistical summary features such as the total length of the gesture, initial stroke angle, and maximum curvature are extracted from this data and used by a classifier algorithm to predict class labels (gesture types). A recognizer thus consists of a bundle of feature extractors and a classifier algorithm trained on a particular gesture corpus.

GestureLab supports all phases of the recognizer development process: (a) collecting, manipulating and sharing gesture corpora, and (b) automatic training and cross-validation of feature extraction and recognizer mechanisms. In the

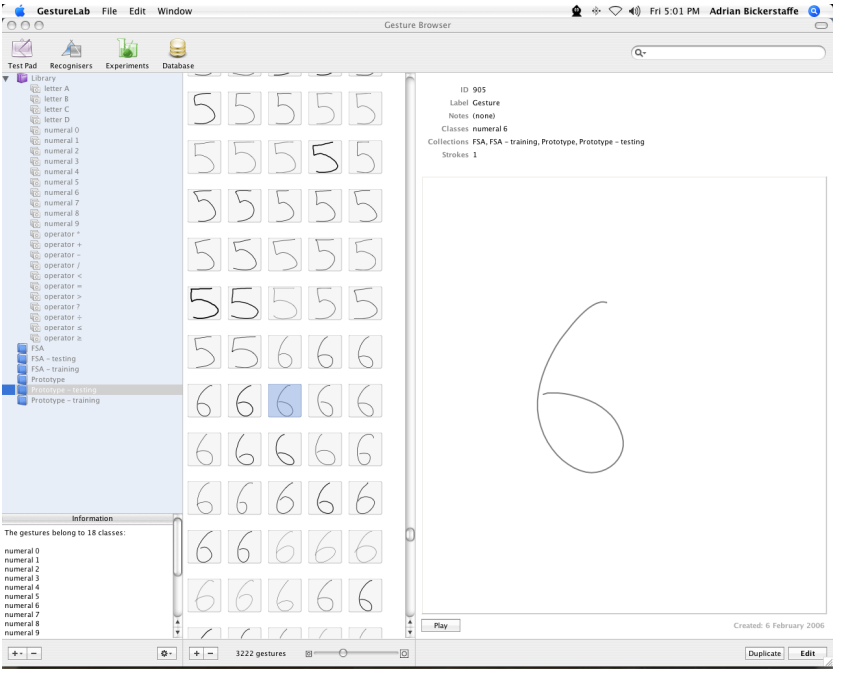

Fig. 1. Main window of GestureLab

event that the built-in feature extraction and recognizer mechanisms are insufficient, GestureLab also allows the developer to readily define (c) new feature extraction mechanisms and (d) new recognizer algorithms.

A core challenge for any two-phase approach which splits lexical and syntactic analysis is that lexical recognition may be ambiguous; contextual information from syntax analysis may be needed to disambiguate the lexical classification. This disambiguation must be delayed until the parsing stage when contextual syntactic information is readily available. To support this, GestureLab generates probabilistic recognizers that return membership probabilities for all possible token classes instead of a single most likely class.

Corpus Management: GestureLab arranges gestures in terms of a library containing named *categories* (or "classes") of gestures and *collections* of gestures selected from these categories. Class membership determines the intended interpretation of a gesture, while collections are named sets for training and testing. Each gesture may belong to any number of named collections and classes. In this way, training and test collections can be created, modified, and deleted without altering the corpora. The library can be accessed and manipulated using an intuitive drag-and-drop interface or via an SQL interface. SQL queries are based on attributes such as collection names, feature data, and experimental results (see below). Gestures can be reviewed visually either as static images or as animations showing the original drawing process.

GestureLab uses a single database to store corpora and experiment data. Remote access to this database is possible using GestureLab clients connected via the Internet. This makes it possible for geographically distributed research groups and for whole research communities to contribute to shared corpora, and to use this data for recognizer development. Corpora and experiment data are also accessible for other software applications via a versatile text-based import/export facility.

Defining New Classifier Algorithms: In the simplest case, a domain-specific recognizer is built by simply training a generic classifier on a domain-specific corpus. GestureLab uses a probabilistic k-way Support Vector Machine as the default recognizer algorithm (see Section 3). When this is insufficient, specialized classifier algorithms can easily be added by the developer. Classifier algorithms are implemented by writing a new C++ class which inherits from a base recognizer and which implements the recognizer interface defined by virtual functions. This interface comprises functions for training and classifying feature vectors, in addition to saving and loading the recognizer to/from file.

Different applications may need different ink pre-processing such as smoothing or hook removal. The responsibility for any pre-processing rests with the individual recognizers so that each recognizer can process stroke data in a manner which is most suitable for the particular application.

Defining New Feature Extractors: The standard GestureLab distribution includes a set of pre-defined feature extractors, following [12]. These include, for example, the initial angle, maxmimum speed, and total duration of the gesture. However, the features required for effective classification of new gesture sets can vary greatly and so GestureLab provides a flexible mechanism for defining new feature extractors. Feature extractors are defined using a plug-in interface and are implemented by writing a C++ class which inherits from a pre-defined feature class. The interface is straight forward: the extractor receives stroke data and returns the feature as a single real-value. In this way, there are no restrictions on the types of features that can be defined or on the algorithms used to compute these. Several feature extractors can be bundled together as a single feature plug-in module.

Automatic Training and Testing: GestureLab offers full support for automatic recognizer training, testing, and experimentation via an intuitive graphical interface. For an experiment, the designer couples specific feature recognizers and classifier algorithms with chosen gesture collections and can then train and validate the thus defined recognizer automatically. This is particularly useful since training times for some classifiers of large alphabets can be extremely long. GestureLab performs automatic cross validation and can automatically create training and test data sets by randomly sampling from a collection of gestures. All experiment data (including feature values, recognition probabilities, parameters settings, etc.) are stored in the central database and are fully accessible so that experiments can be easily repeated and varied. A versatile experiment report

facility allows the developer to obtain experiment summaries including the overall recognizer accuracy, the number of gestures correctly/incorrectly classified, and the particular gestures which were misclassified. Results can be filtered to display only correct or only incorrect predictions. Gestures contained in the results table can be displayed graphically and replayed as a temporary collection. This is particularly useful for diagnosing causes of misclassification and developing new feature extractors to address the problems found.

GestureLab also supports quicker, less comprehensive testing of recognizers. A "test pad" allows recognizers to be evaluated on a gesture by gesture basis using interactive input instead of a whole gesture collection. The test pad is particularly useful for investigating unexpected recognizer traits.

3 SVM Gesture Recognition in GestureLab

The default classifier algorithm for GestureLab recognizers is the Support Vector Machine (SVM [2,13]). SVMs are chosen because they are well-understood, theoretically well-founded and have proven performance in a wide range of application areas.

Linear SVMs: Basic SVMs are binary linear wide-margin classifiers with a supervised learning algorithm. Let X be a set of m training samples, $x_i \in \mathbb{R}^n$, with associated class labels $c_i \in \{+1, -1\}$. Assuming linear separability, the goal of SVM learning is to find an $(n-1)$-dimensional hyperplane which separates the classes $\{x_i \in X | c_i = +1\}$ and $\{x_i \in X | c_i = -1\}$. Such a hyperplane fulfils $c_i(w \cdot x_i + b) \geq 0$ and corresponds to the decision function $c_{pred}(x) = \text{sign}((w \cdot x + b))$. In general, there are an infinite number of such hyperplanes.

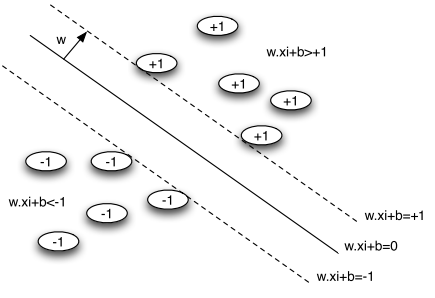

Fig. 2. SVM classification example

SVMs compute the hyperplane that provides the maximum class separation by finding a maximal subset $S \subseteq X$ of so-called support vectors for which w, b can be re-scaled such that $c_i(w \cdot s_i + b) = 1$ for $s_i \in S$. The separation margin perpendicular to the separation hyperplane is $2/||w||$ (see Fig. 2), so that maximizing the margin can be done by solving the quadratic program (QP)

$$\min_{w,b} ||w||^2 \text{ s.t. } \forall i : c_i(w \cdot x_i + b) \geq 1 \tag{1}$$

or its dual

$$\max_{\alpha_i} \sum_{i=1}^{m} \alpha_i - \sum_{i=1}^{m} \sum_{j=1}^{m} \alpha_i \alpha_j c_i c_j {x_i}^T x_j \ s.t. \ \sum_{i=1}^{m} \alpha_i c_i = 0 \wedge \ \forall i : \alpha_i \geq 0 \tag{2}$$

where $w = \sum_i \alpha_i x_i c_i$.

Such a hyperplane cannot be found if the two classes are not linearly separable: some x_i will always be on the wrong side of the plane and QP (1,2) is not feasible. In this case, the aim is to minimize the classification error whilst simultaneously maximizing the margin. This is achieved by introducing a penalty term ξ_i for misclassified samples in the corresponding QP (3) or its dual.

$$\min ||w||^2 + C \sum_i \xi_i \text{ s.t. } \forall i : c_i(w \cdot x_i + b) \geq 1 - \xi_i \tag{3}$$

Non-linear SVMs: The approach described thus far computes only linear classifiers.

In many cases an SVM can, however, separate classes that require non-linear decision surfaces by first transforming the data into some higher-dimensional space in which linear separation is possible. Such transformations can potentially be expensive, but the so-called "Kernel Trick" allows us to side-step the explicit transformation. For a transformation $\phi(\cdot)$, a kernel function $k(\cdot, \cdot)$ computes the dot product of transformed data without explicitly computing the transformation, i.e. $k(x, x') = \phi(x) \cdot \phi(x')$. Of course, kernel functions can only be found for a limited class of transformations $\phi(\cdot)$. Kernel functions provide a general way to apply a linear algorithm (in a limited way) to non-linear problems, provided the crucial computations of the algorithm can be phrased in terms of dot products, as is the case for SVMs.

A non-linear SVM attempts to perform linear separation of the transformed samples $\phi(x_i)$ using the kernel trick [13]. Common kernels include the polynomial kernel $k(x, x') = (x \cdot x')^d$ and the Radial Basis Function (RBF) kernel $k(x, x') = \exp(-\gamma||x - x'||^2)$, $\gamma > 0$. Fig. 3 shows a non-linear classification problem.

GestureLab's default classifier uses RBF kernels and performs a two-dimensional grid-search to optimize the kernel parameters. This search is guided by cross-validation results using all training data relevant to the decision node and 5-fold cross-validation.

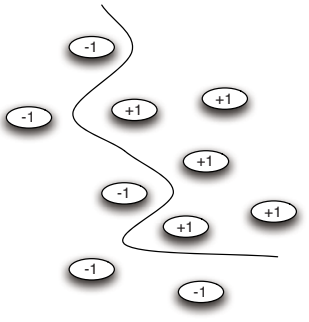

Fig. 3. A non-linear classification problem linearly separable in the transformed feature space

Multiclass SVMs: Standard SVMs are binary classifiers and it is not at all straightforward to use these for multi-way classification. The standard techniques to build k-way SVMs are one-against-all [4], one-against-one [4], and DAGSVM schemes [11]. A one-against-all classifier requires k SVMs for a k-class problem, where the i^{th} SVM is trained using all samples from the i^{th} class versus all other samples. A sample is classified by evaluating all k trained SVMs and the label of the class for which the decision function is maximum is chosen. The one-against-one scheme trains $\frac{k(k-1)}{2}$ classifiers derived from pairwise comparison of target

classes. A prediction is made by evaluating each SVM and recording "votes" for the favored class; the class with the most votes is selected as the predicted class. Both methods suffer from very long training times and this issue is further compounded for large data sets such as our corpus of over 10000 gestures. Furthermore, there is no bound on the generalization error of one-against-all schemes, and one-against-one schemes can tend to overfit.

The DAGSVM scheme is more complex. The decision DAG is created by viewing the problem as a series of operations on a list, with each node eliminating one element from the list. Given a list initialized with all class labels, the root node is formed using the training data corresponding to the first and last elements of the list. A decision can now be made which will eliminate one of the two classes being compared. The eliminated class is removed from the list and the DAG proceeds to form a child node again using the first and last list elements. The formation of child nodes in this manner occurs for both decision paths and continues until only one element remains in the list. The DAGSVM will consequently comprise $\frac{k(k-1)}{2}$ nodes and achieve predictions by evaluating $k-1$ of these nodes. Note that each final node can be reached using more than one pathway from the root node and thus, acyclic graph structure is exhibited. The problem of lengthy training times also applies to the DAGSVM schema since, like one-against-one, it requires training $\frac{k(k-1)}{2}$ decision nodes. The performance of a DAGSVM also relies on the order in which classes are processed, and no practical method is known to optimize this order.

We believe a better approach is to reduce the set of possible classes at each decision node and take relative class similarity into account during the construction of the decision tree. We construct the decision tree as a Minimum Cost Spanning Tree (MCST) based on feature distances. Each of the leaves corresponds to a target class and the interior nodes group classes into progressively more disjoint sets. For each internal node in the MCST an SVM is trained to separate all samples belonging to classes in its left subtree from those in the right subtree. Fig. 4 contrasts the DAGSVM and MCST-SVM approaches for a four class example.

The MCST recognizer scales features to $[-1, 1]$ and computes a representative feature vector for each class. The representative feature for a given class is the centroid of all samples belonging to that class. Euclidean distances between all

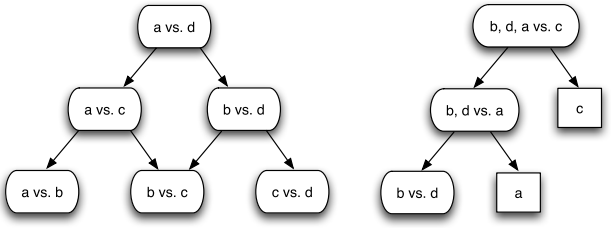

Fig. 4. DAGSVM (left) vs. MCST (right) structure

unique unordered pairs of representative vectors are calculated, and from these distances (or "edge weights") an MCST is constructed (in polynomial time) using Kruskal's algorithm [6]. Average-linkage and complete-linkage versions of the decision tree have also been implemented.

The MSCT recognizer requires $k-1$ nodes for a k-class problem and a maximum of $k-1$ decisions for a prediction. MCST recognizers have a core advantage over the other schemas since they discriminate between classes based on class similarity. Furthermore, training time is reduced because only $k-1$ SVMs must be trained.

Probabilistic SVMs: A standard SVM provides only a non-probabilistic class prediction ("best guess"). As explained earlier, probabilistic predictions are required to perform context-based syntactic disambiguation. The MCST approach facilitates inference of probability distributions for prediction errors during the training phase in a simple manner: after completing the training of all recognizer nodes, a test prediction for each training sample is made and the frequencies of predicting class c_i for a data item of true (known) class c_j are tabulated. From these, maximum likelihood probability distributions are computed for each leaf node of the SVM decision tree.

Coupled with a standard set of feature extractors, the probabilistic MCST-SVM recognizers produce state-of-the-art recognition rates [3].

4 Cider

Syntactic recognition is provided by Cider. Only a quick overview can be given here, for more details see [5]. Fig. 5 shows the components that comprise the Cider toolkit and how these components are used in the creation of an application. The white boxes indicate components of Cider; cross-hatched boxes indicate optional components that can be tailored to extend the capabilities of the toolkit; shading indicates components that must be created by the application developer.

Cider automatically generates a parser for a visual language from a grammatical specification of the visual language's syntax. Parsers produced by Cider are fully incremental which means that users can add, delete, or modify components of a diagram at any time and that the interpretation engine automatically maintains a consistent interpretation of the diagram state. Furthermore, the ability to specify syntactic transformations provides a powerful mechanism for encoding diagram manipulations and user interactions. Cider compiles the grammar and transformation specifications into libraries that can then be used as domain-specific diagram interpretation engines by an application.

Both the syntax and transformation rules are specified using Constraint Multiset Grammars [9], a kind of attributed multi-set grammar. As a simple example, consider the following production which defines a division term `t` as composed of two numerals `a` and `b` with a horizontal division line:

```
t:Term     ::= l:Line
               exist a:Numeral, b:Numeral
                  where immediately_above(a.bbox, l.bbox) and
                     immediately_below(b.bbox, l.bbox) and
                     horiziontally_centered(l.bbox, a.bbox) and
                     horizontally_centered(l.bbox, b.bbox)
               { t.value = a.value / b.value }
```

Importantly, Cider supports structure preserving diagram manipulation. This means that specifications can be written so that once a syntactic diagram component has been recognized, the syntactic structure is automatically maintained when the user manipulates one of the component constituents. For example, when a fraction has been recognized in a mathematical expression and the user extends the denominator, the fraction line can automatically be extended; when the fraction line is dragged, numerator and denominator terms can be dragged with it, etc. This is achieved by using a constraint solver in the diagram processor to automatically update attribute values of tokens so that the specification remains consistent with the visual state of the diagram.

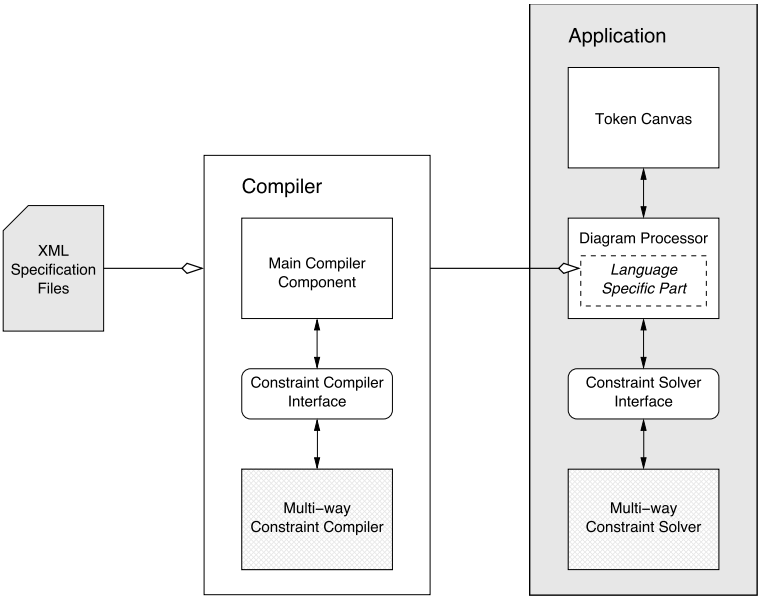

Fig. 5. Cider Architecture

5 System Integration

Building an SDE with GestureLab and Cider: Cider and GestureLab provide a powerful tool suite for building pen-based SDEs. An SDE created with GestureLab and Cider has three main components: a graphical front-end where

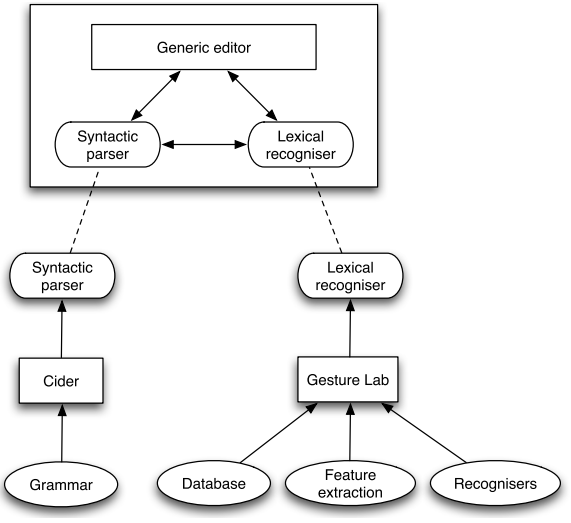

Fig. 6. System components

input is drawn, the Cider runtime environment, and the recognition engine. The recognition engine consists of the lexical recognizer generated by GestureLab and the syntax recognizer (i.e. parser) generated by Cider (see Fig. 6). The graphical front-end is either provided by the application or a generic graphical editor.

The first step in building such an SDE is to bundle the gesture recognizers developed in GestureLab as a single static library. The second step is to develop a Constraint Multiset Grammar that parses the targeted diagram language. Using Cider, this grammar is compiled into another static library that encapsulates the syntactic recognition engine. The application must then be set up to communicate with Cider through the Cider controller API. Primarily, the controller handles the addition, modification and deletion of gestures on the front-end side, as well as the addition of non-terminals (recognition of syntactic components), and their modification and deletion. Non-terminals are added, modified and deleted by Cider as the consequence of a production application or in reaction to a structure preserving manipulation. Communication between the front-end and Cider occurs in terms of requests made to Cider from the front-end. Asynchronous responses from Cider are handled by registering callback functions for events such as the creation or removal of a non-terminal symbol, or attribute changes.

Handling Lexical Ambiguity: Interpretation by the recognition engine is a two-phase process: first, the lexical recognizer provides a probabilistic classification of the gesture; second, resolution of lexical ambiguity is delayed until parsing so as to allow the use of contextual information. For instance, consider a toy problem in which we want to recognize divisions written in TeX in-line style, such as “4 / 5” where numbers always comprise a single digit (the extension to multi-digit numbers is simple).

The grammar contains a "Gesture" terminal symbol in addition to "Numeral", "Operator", and "Term" non-terminal symbols. All symbols have bounding box attributes. The "Numeral" and "Term" non-terminals require a further integer attribute to store their numerical value.

The problem that we encounter is that there can be ambiguity on the lexical level when classifying a vertical line: depending on the angle of the line, it may represent the numeral "one" or a division operator. At some angles the interpretation will be ambiguous and the classification must be delayed until sufficient syntactic context is available to disambiguate in the parser. A vertical line must be a division operator if there are numeric operands to its left and right, whereas the line must be a numeral with value "one" if there is no numeral immediately to the left or right of it. This syntactic disambiguation is taken into account in the following CMG grammar fragment:

```
n:Numeral  ::= g:Gesture where ( most_likely(g, zero) )
               { n.value := 0, n.bbox = Gesture.bbox }
n:Numeral  ::= g:Gesture where ( most_likely(g, line) )
               not exists ( n:numeral )
                  where ( immediately_left_of(n.bbox, g.bbox) or
                          immediately_right_of(n.bbox, g.bbox) )
               { n.value := 1, n.bbox = Gesture.bbox }
...
n:Numeral  ::= g:Gesture where ( most_likely(g, nine) )
               { n.value := 9, n.bbox = Gesture.bbox }

d:Operator    ::= g:Gesture where (most_likely(g, line) )
               exists ( n:numeral )
                  where ( immediately_left_of(n.bbox, g.bbox) or
                          immediately_right_of(n.bbox, g.bbox) )
               { d.bbox = g.bbox }
t:Term     ::= o:Operator
               exist a:Numeral, b:Numeral
                  where immediately_left_of(a.bbox, o.bbox) and
                     immediately_right_of(b.bbox, o.bbox) and
                     vertically_centered(o.bbox, a.bbox) and
                     vertically_centered(o.bbox, b.bbox)
               { t.value = a.value / b.value }
```

The attribute constraints will be automatically processed with default tolerances, however tolerances can be set explicitly and arbitrary attribute tests can be implemented as user-defined functions.

Case study: The GestureLab-Cider pair has been used to create a pen-based front-end for a computer algebra system. This system interprets stylus-drawn mathematical expressions and handles fractions, exponentials, basic arithmetic, and matrices. The case study demonstrates the viability of the GestureLab-Cider approach for generating domain-specific SDEs. Note that due to the incremental nature of Cider parsers, expressions can be written in any order of symbols and can arbitrarily be modified; a consistent interpretation will automatically be maintained at all times.

6 Conclusions

This paper has described GestureLab, a tool designed for building domain-specific gesture recognizers, and its integration with Cider, a grammar engine for parsing visual languages that use GestureLab recognizers. Together these two systems form a suite of generic tools for the construction of interactive sketch interpretation systems. These tools automate the SDE construction process to a high degree.

GestureLab has been specifically designed to facilitate collaboration between researchers, allowing gesture corpora to be stored and shared via remote databases either locally or via the Internet. The software has been released into the public domain at `http://www.csse.monash.edu.au/~adrianb/GL/Home.html`. GestureLab aims to provide synergy between different research efforts by facilitating the sharing of corpora and recognizer reference implementations. Cider is also available upon request.

References

1. Chang, C., Lin, C.: LIBSVM: a library for support vector machines (2001), `http://www.csie.ntu.edu.tw/~cjlin/libsvm`
2. Cortes, C., Vapnik, V.: Support-vector network. Machine Learning 20, 273–297 (1995)
3. Garain, U., Chaudhuri, B.B.: Recognition of online handwritten mathematical expressions. IEEE Transactions on Systems, Man, and Cybernetics - Part B 34(6), 2366–2376 (2004)
4. Hsu, C.W., Lin, C.J.: A comparison of methods for multi-class support vector machines. IEEE Transactions on Neural Networks 13 (2002)
5. Jansen, A.R., Marriott, K., Meyer, B.: Cider: A component-based toolkit for creating smart diagram environments. In: International Conference on Distributed and Multimedia Systems, Miami (September 2003)
6. Kruskal, J.B.: On the shortest spanning subtree and the traveling salesman problem. Proceedings of the American Mathematical Society 7, 48–50 (1956)
7. Liu, W.: On-line graphics recognition: state-of-the-art. In: Lladós, J., Kwon, Y.-B. (eds.) GREC 2003. LNCS, vol. 3088, pp. 291–304. Springer, Heidelberg (2004)
8. Lorena, A.C., de Carvalho, A.C.P.L.F.: Minimum spanning trees in hierarchical multiclass support vector machines generation. In: Ali, M., Esposito, F. (eds.) IEA/AIE 2005. LNCS (LNAI), vol. 3533, pp. 422–431. Springer, Heidelberg (2005)
9. Marriott, K., Meyer, B.: On the classification of visual languages by grammar hierarchies. Journal of Visual Languages and Computing 8(4), 374–402 (1997)
10. Meyer, B., Marriott, K., Allwein, G.: Intelligent diagrammatic interfaces: state of the art. In: Diagrammatic Representation and Reasoning, pp. 411–430. Springer, London (2001)
11. Platt, J.C., Cristinini, N., Shawe-Taylor, J.: Large margin DAGs for multiclass classification. Advances in Neural Information Processing Systems 12, 547–553 (2000)
12. Rubine, D.: Specifying gestures by example. Computer Graphics 25(4), 329–337 (1991)
13. Schölkopf, B., Smola, A.: Learning with kernels. MIT Press, Cambridge (2002)