Connector Semantics for Sketched Diagram Recognition

Isaac J. Freeman and Beryl Plimmer

Department of Computer Science
University of Auckland,
Private Bag 92019 Auckland, New Zealand,

isaac@freeman.org.nz and beryl@cs.auckland.ac.nz

Abstract

Comprehensive interpretation of hand-drawn diagrams is a long-standing challenge. Connectors (arrows, edges and lines) are important components of many types of diagram. In this paper we discuss techniques for syntactic and semantic recognition of connectors. Undirected graphs, digraphs and organization charts are presented as exemplars of three broad classes that encompass many types of connected diagram. Generic techniques have been incorporated into the recognition engine of InkKit, an extensible sketch toolkit, thus reducing the development costs for sketch tools.

Keywords: sketch recognition, sketched diagrams, connector semantics.

1 Introduction

Pen input devices offer an intuitive and informal mode of interaction with a computer which is a viable alternative to the traditional keyboard and mouse. However, the advantages of pen input come at a cost in processing requirements: to fully exploit its potential software must be able to recognize and interpret gestures made by the user, and to convert them into symbolic representations that capture the user's intent. The vast variety of shapes that may be drawn constitutes a considerable challenge for interpretation. Significant progress has been made with text, and robust character recognition is a standard feature in operating systems such as Microsoft's Tablet PC Edition of Windows and Apple's Mac OS X. Recognizing pen-based input only as text, however, removes much of its potential expressiveness and constrains it to a domain in which it is slower and more unwieldy than a conventional keyboard. A user interface that truly embraces the potential of pen input should be able to recognize not only textual input in the form of characters, but also graphical input in the form of diagrams.

Reliable diagram recognition is a precursor to many of the operations that can potentially be supported with intelligent sketch tools; operations such as beautification, translation into other data formats, animation and execution. Progress has been reported in many of these areas, but most projects have been specific to a particular domain, with recognition tailored to the symbols and syntax of one type of diagram. Tools that recognize graphs (Arvo and Novins 2006), UML diagrams (Damm, Hansen et al. 2000; Hammond and Davis 2002; Chen, Grundy et al. 2003), architectural blueprints (Trinder 1999; Do and Gross 2001), or user interface designs (Landay and Myers 1995; Igarashi 2003; Lin and Landay 2003; Plimmer and Apperley 2003; Coyette, Faulkner et al. 2004) each represent a significant advance. There is a good deal of commonality between them that can be brought to bear on the problem of general diagram recognition.

InkKit (Chung, Mirica et al. 2005) is a general diagramming toolkit. It deals with those aspects of diagram recognition common to many domains, and provides an extensible architecture for modules supporting domain-specific features. InkKit is intended to reduce development costs for sketching tools, and to provide an environment in which research can conveniently be conducted into general aspects of sketched diagram support. It includes a well-designed and tested user interface and a powerful example-driven recognition engine. Uniquely, InkKit's recognition engine recognizes both characters and shapes within diagrams, and deals with each appropriately.

The fundamental recognition engine employed by InkKit is based on a similar engine developed for Freeform (Plimmer and Apperley 2003), a user interface sketch tool, and thus it was already well-suited to diagrams depicting layout of user controls in a graphical user interface. Through InkKit's development the set of diagram domains it is capable of recognizing has gradually extended. Some techniques developed for particular domains can be usefully applied to others, and we have aimed to find a suitable level of abstraction for each aspect of recognition.

Many diagrams include shapes such as arcs, edges, and arrows, which indicate connections between other shapes. The frequency with which connectors appear suggested that they should not be treated as domain-specific features, but identified as part of InkKit's generic recognition. While there are numerous examples of sketch tools that recognize connectors within specific types of diagram, this work is unique in describing a general solution to connector recognition.

Copyright © 2007, Australian Computer Society, Inc. This paper appeared at Eighth Australasian User Interface Conference (AUIC2007), Ballarat, Victoria, Australia. Conferences in Research and Practice in Information Technology (CRPIT), Vol. 64 Wayne Piekarski and Beryl Plimmer, Ed. Reproduction for academic, not-for profit purposes permitted provided this text is included.

2 Background

The essential components of a useful sketch tool are: hardware that supports pen input, a paper-like user interface, and a powerful recognition engine. Most suited to sketching are display surfaces that can capture high-quality stylus input. Wacom™ tablets and Tablet PCs meet these requirements, but offer limited display space. Various techniques have been adopted to minimize the problems of small displays, including zooming (Lin, Newman et al. 2000) or a radar window (Damm, Hansen et al. 2000). Larger surfaces such as E-whiteboards increase the available display space, but currently lack the input accuracy of tablets.

A digitally supported drawing space should be paper-like to capture the advantages of unconstrained sketching (Goel 1995). Yet it should also provide the advantages of computational support, such as cut, copy, paste, and undo functionality.

Providing intuitive editing support with a stylus is challenging. Researchers have explored automatic grouping (Elrod, Bruce et al. 1992) and the use of functional gestures (Li, Hinckley et al. 2005), but both of these techniques fail badly if recognition is not completely accurate. Storyboards are used by many sketch tools (Bailey, Konstan et al. 2001; Lin and Landay 2003; Plimmer and Apperley 2003) to visualize multiple sketches and create relationships between them.

Most diagrams consist of both text and shapes. Reliable text recognition is now commonly available as an operating system service, but reliable, comprehensive diagram recognition is an on-going challenge. Early shape recognition algorithms required each shape to be drawn in a single stroke (Rubine 1991), an unnatural constraint to place on users. Other shape recognition techniques include joining adjacent strokes before recognition, applying fuzzy logic (Fonseca, Pimentel et al. 2002) and template-matching and identifying connectors first (Kara and Stahovich 2004). With Freeform (Plimmer and Apperley 2003) we provided a user interface for users to define spatial relationships between two ink strokes. The user selected primary and secondary shapes (rectangle, circle, etc.) and a spatial relationship (contains. beside, etc.) from lists to define, for example, a radio button as a small circle that may have text beside it.

Two broad categories of sketch recognition engine can be distinguished according to the stage in the sketching process at which they begin to act on incoming data. *Eager* recognition engines attempt to recognize shapes immediately as they are drawn. This provides instant feedback to the user, and has often been assumed to be a necessary feature of diagram recognition, along with beautification routines that immediately convert rough sketch lines into smooth formal shapes. *Lazy* recognition engines do not attempt to determine what the user has drawn until after the sketch is completed. Studies in the user interface domain (Bailey and Konstan 2003; Plimmer and Apperley 2003) have found that this offers advantages in real world applications. We believe that these advantages carry over to other forms of rapid prototyping, as informal sketched diagrams are most suitable through most of the design process, and formal output is only beneficial at the conclusion. No recognition method is perfect, and while an eager engine draws

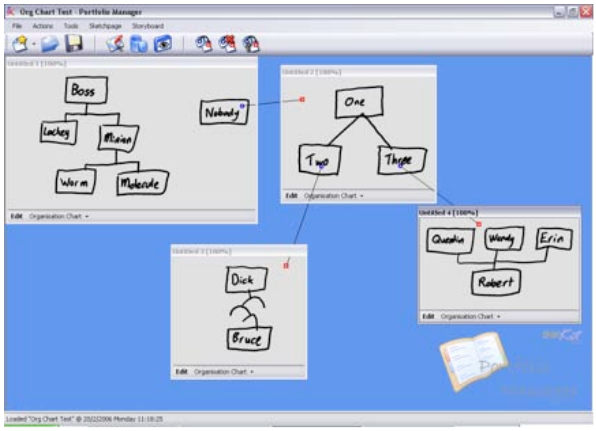

Figure 1: The InkKit Portfolio View, showing multiple linked sketches typically displayed on a large auxiliary display, while windows containing individual diagrams would be opened on a device with appropriate input capabilities, such as a Tablet PC.

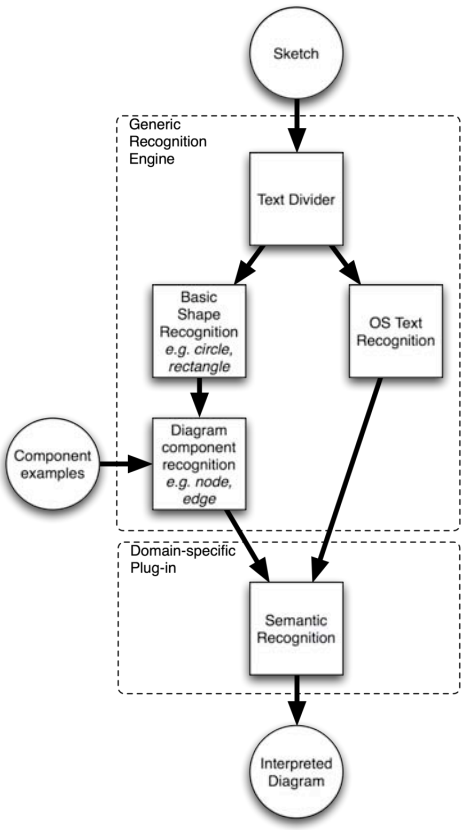

Figure 2: The InkKit recognition process. Sketches are first processed by a generic recognition engine, then passed to a plug-in that performs interpretation specific to a particular type of diagram.

attention to errors while the user is still trying to complete the diagram, lazy engines avoid interruption by delaying correction.

Other work on toolkits for sketch support includes that of Hammond and Davis (2003) who have taken an approach similar to Freeform: extending the number of components that may be defined, but requiring the user to write rules. Lank (2003), proposes a retargetable framework which, like InkKit, automatically builds recognition rules from users examples and recognizes writing. However, Lank's toolkit requires a significant amount of code to be written for each new diagram type.

2.1 InkKit

InkKit runs on the Windows XP Tablet OS, and utilizes the ability of modern graphics cards to span a desktop across multiple display devices to increase the available area. The user interface consists of windows for editing sketches, and a portfolio window (Figure 1) which shows the all sketches currently in use. While the interface can be used comfortably on a single screen, it has been designed to allow for collaborative development, in which the portfolio window would typically be presented on a large display, and sketches edited on a Tablet PC.

The sketch windows support two modes: ink mode with the standard drawing tools and features commonly found in paint applications, and an edit mode in which they can move and re-size recognized shapes, and correct any components that have been incorrectly identified.

InkKit uses lazy recognition: no attempt is made to interpret a sketch until the user clicks on the 'Recognize' button. We believe that the advantages of eager recognition are minimal for most types of diagrams, and that it often interrupts free sketching. This gives a distinct advantage at recognition time: by choosing when to perform recognition the user implicitly indicates that the sketch is in a state suitable for recognition - it is unlikely that any meaningful symbol will be half-drawn.

A further constraint that improves recognition is provided by user selection of the domain for each sketch. While a portfolio may collect many types of sketches, each sketch belongs to a single domain. At recognition time the set of diagram components to match with shapes is limited to those found in the relevant domain.

Shape recognition in InkKit follows four cleanly separated phases (Figure 2). In the first phase, a divider routine separates text from other diagram shapes. Text is passed to the OS for recognition. The second phase first joins multi-stroke shapes (such as a rectangle drawn as four lines) and then applies Rubine's (1991) single stroke algorithm to recognize domain-independent geometric shapes such as rectangles, triangles, circles and lines. In the third phase, groups of basic shapes are compared with user-drawn examples of domain components. Shapes are matched by pattern-matching visual features, using a variant of Rubine's algorithm. The final phase determines the structure of the diagram from the spatial relationships between recognized components. Component classes may be tagged with properties describing how they interact

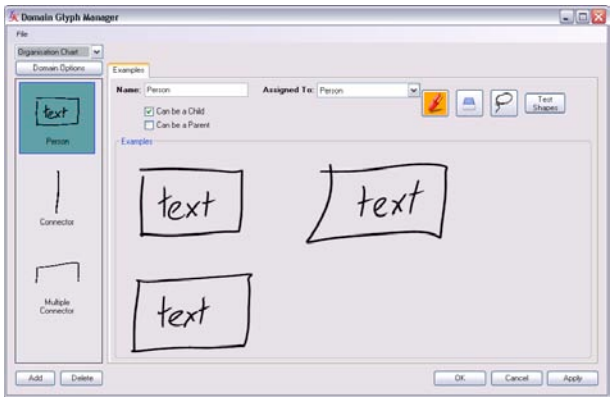

Figure 3: User interface for defining component types. This particular example is the organization chart domain.

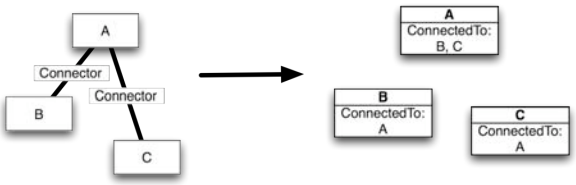

Figure 4: Generic processing of connector objects by a domain plug-in.

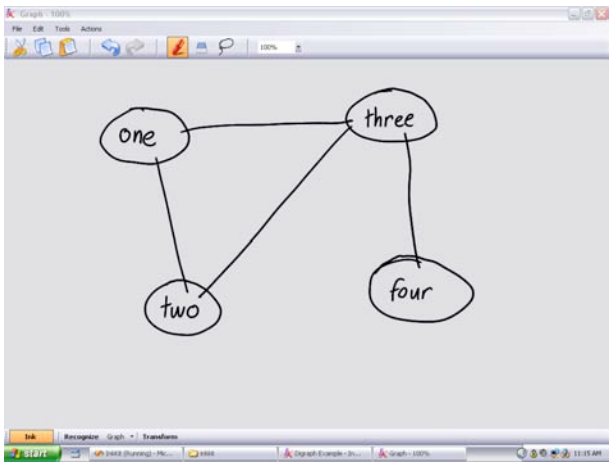

Figure 5: An undirected graph sketched in InkKit

with other components. We aim to keep the set of possible properties to a minimum, and in fact the initial version of InkKit had only two, indicating whether a component might contain other components (parent), or be contained within other components (child). Later in this paper we will discuss the addition of two further properties.

This architecture minimizes the impact of recognition on the user. They do not need to define explicit rules to train InkKit, only to provide two or three examples of each component to be recognized (Figure 3). In fact, we have found that it is not crucial that the examples be provided

by the same user as the sketch - success of recognition is not strongly affected.

The clean separation of recognition phases allows us to progressively improve the recognition engine, experimenting with alternative approaches and evaluating their effectiveness. Informal evaluations with five novice users achieved approximately 85% accuracy recognizing simple user interface diagrams.

Domain-specific aspects of recognition are supported via a plug-in architecture. A domain is specified by a code module defining its specific diagram components and a set of example components. Further modules may then be written to support output in different formats. For example, currently user interface diagrams can be output as Java source code or as HTML. Graphs and organization charts can be output as either a text description, a bitmap graphic, or constructed as diagrams in Microsoft Word using Microsoft's .Net libraries for Office.

We have successfully developed modules for user interface sketches (Chung, Mirica et al. 2005) which replicate the functionality of Freeform, and add further features such as converting sketches into HTML form code and Java source. Appropriate behaviors such as buttons leading to other forms, and dropdown lists being filled with words are specified by creating links between sketches in the InkKit portfolio view. The result is a simple rapid-prototyping tool for software applications that can generate user interface code from a sketch.

3 Connectors

For many types of diagram, recognizing discrete components of the diagram is not sufficient to correctly interpret meaning. For example, little of value is achieved if a graph is recognized only as a list of edges and nodes. The edges signify relationships between the nodes, and these relationships are crucial to correctly interpreting the graph. There is a large class of diagrams in which connectors represent relationships, but the precise nature of the relationship differs between diagram types. Thus, in the original InkKit model, connecting shapes were not accounted for in the main recognition engine, but left up to authors of individual plug-ins. In this paper, we identify broad classes of connector generic enough to be integrated into a sketch recognition engine.

The meaning of a connector depends on a combination of attributes of its own shape and of the overall sketch layout. The number of shapes connected, their position, or an inherent directionality of the connector may be significant to different degrees in different types of diagram. To explore these ideas we selected three exemplars to implement as InkKit modules: simple undirected graphs, directed graphs, and organization charts. Figure 4 summarizes the approach taken in all three modules: a set of spatially positioned node and connector objects is converted into a set of node objects tagged with their logical relationships to other objects, based on the connector model relevant to the domain.

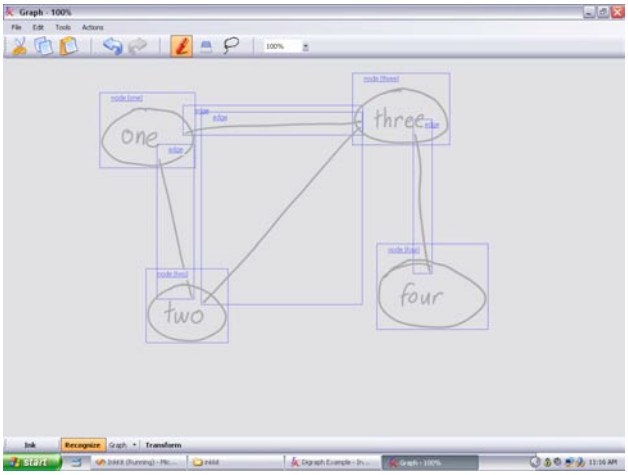

Figure 6: An undirected graph that has been decomposed by the InkKit recognition engine into its node and edge components

```
Graph Contents
Node "three"
has an edge to node "one"
has an edge to node "two"
has an edge to node "four"

Node "one"
has an edge to node "three"
has an edge to node "two"

Node "four"
has an edge to node "three"

Node "two"
has an edge to node "three"
has an edge to node "one"
```

Figure 7: Text output for an undirected graph

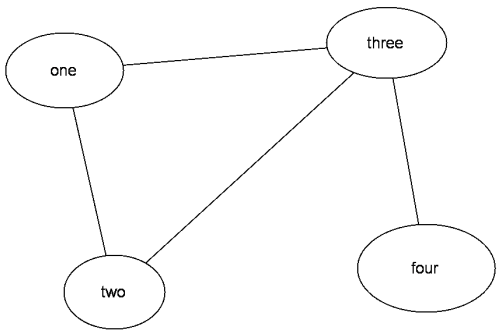

Figure 8: Formal graphical output of an undirected graph from InkKit

3.1 Undirected Graphs

Undirected graphs (Figure 5) provide a base case for connected diagrams. Their edges have no direction, relationships can be determined completely by detecting overlaps - if the end points of an edge fall within two different node shapes, those nodes are connected.

InkKit's recognition engine separates the graph into its edge and node components (Figure 6), and records overlaps. The set of recognized components is passed to the graph module, which iterates over the edges, attaching to each node a list of other nodes to which it is connected. The nodes are then deleted from the logical description of the graph. InkKit output plug-ins produce a simple text description (Figure 7) of the graph, or render it as a formal diagram Figure 8) in a variety of graphics formats. The graphical output is not a simple beautification of the original sketch, but is constructed from the same logical description of the graph as the text output. Further plug-ins to InkKit could be developed to convert this logical description into any other format required.

3.2 Directed Graphs

In a directed graph (Figure 9), incoming arcs are distinguished from outgoing arcs with arrows. The domain module expands on the undirected graph algorithm by identifying the directions of connections.

While processing connectors, the directed graph module distinguishes between the longest stroke of the connector, which is assumed to be the "shaft" of the arrow and any other shorter strokes which are assumed to make up the head. The direction of the arrow is determined according to which end of the longest stroke the shorter strokes are closer to. This approach does not require the user to draw arrows in any particular style, but does have the disadvantage that the shape of the arrowhead does not affect the direction interpreted by the plug-in: a triangle at the top of a vertical line will be assumed to mean that the arrow is pointing upwards, even if the triangle points down. More sophisticated recognition techniques are possible with further development. The directed graph tags nodes with separate lists of incoming and outgoing connections and as before, the connectors are deleted from the logical description of the graph. Figures 10 and 11 show results from text and graphical output modules.

3.3 Organization Charts

Organization charts exemplify a third type of connected diagram, in which the direction of connectors is a property of the sketch itself. Nodes in an organization chart represent people in a hierarchical organization (Figure 12). Superiors are positioned higher in the sketch than their subordinates, so all connectors are directed downwards, and arrowheads are not required.

Organization charts may also include one-to-many relationships, in which a superior has multiple subordinates. They often form tree structures, and the techniques required to recognize them are applicable to

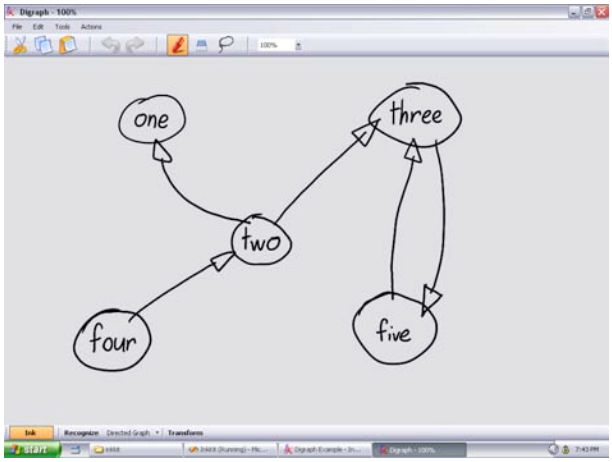

Figure 9: A directed graph sketched in InkKit

```
Directed Graph Contents
Node "three"
has an incoming arc from node "two"
has an incoming arc from node "five"
has an outgoing arc to node "five"

Node "one"
has an incoming arc from node "two"

Node "two"
has an incoming arc from node "four"
has an outgoing arc to node "three"
has an outgoing arc to node "one"

Node "five"
has an incoming arc from node "three"
has an outgoing arc to node "three"

Node "four"
has an outgoing arc to node "two"
```

Figure 10: Text output for a directed graph

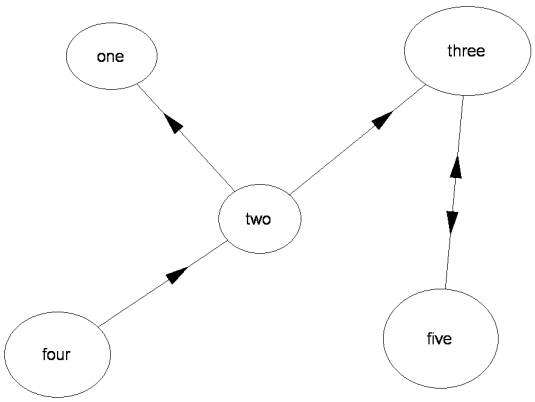

Figure 11: Formal output of a directed graph from InkKit

other tree diagrams. However, organization charts encompass a broader class than trees, as a subordinate may sometimes have more than one superior. So connectors must be able to depict many-to-many relationships.

A person will typically be drawn as a box containing text, although any other shape may be substituted if the user provides suitable example sketches. Rather than require the user to create examples for every possible type of connection, we allow a relationship between people to be represented by a chain of one or more connectors. For example, the relationship between a superior and two subordinates may be made up of a straight vertical line connected to an inverted-U shape. InkKit can recognize multiple versions of the same component, so that direct straight-line connectors and multiple connectors may be considered to be two different examples of a connector component. In fact, the constraint of working with a specific domain allows InkKit to interpret as a connector any shape that is not a person, which allows the user a great deal of freedom. As before, InkKit's main engine recognizes an unordered list of shapes, both Persons and Connectors (Figure 13).

Each connector is processed by the domain module in a manner similar to that for directed graphs: persons at its top end have persons from the bottom end added to their inferiors list, and vice versa. As connectors may be chained, it is necessary to detect for each connector not just the persons at its superior and inferior ends, but also any other connectors that may lead on to further persons. To handle this requirement, connectors are tagged with superior and inferior lists of the same type as those for person objects, and processed as persons until all their connections have been assigned to actual person shapes. Processing proceeds through all connectors in a single pass, and the result is a list containing only persons annotated with their relationships, which is passed to domain modules for output (Figures 14 and 15). This is the most complex of the connector algorithms we have developed, and pseudocode is provided in figure 16.

4 Generalizing connector recognition

From these three modules we identified the core syntactic and semantic requirements for intelligently recognizing connectors, and identified common functionality to integrate into the final phase of the core recognition engine. This code identifies the components at the endpoints of a connector and tags each component with the connector information. With this information the domain specific modules can trace the paths between components and apply path information as appropriate. By providing generic connector information the amount and complexity of code required in interpreter modules for connected diagrams is reduced.

Once the generic connector code was integrated into InkKit we re-engineered the modules. The graph, digraph and organization chart originally required 337,

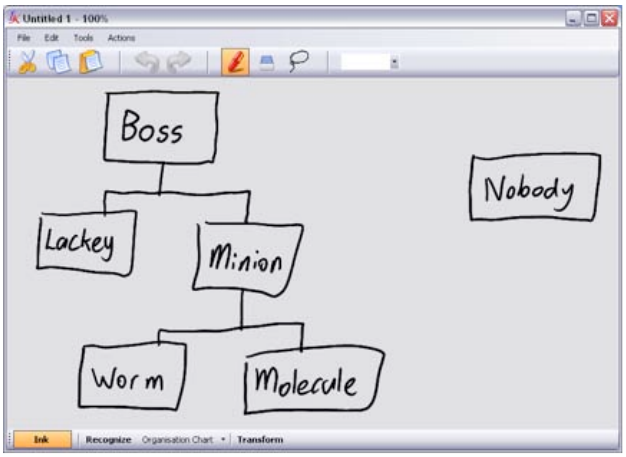

Figure 12: An organization chart sketched in InkKit

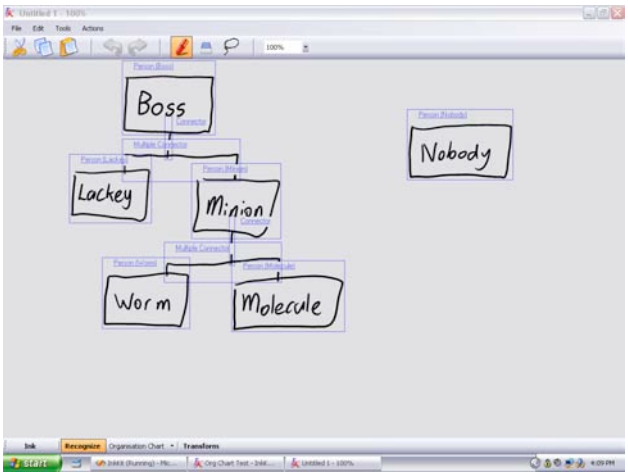

Figure 13: An organisation chart that has been decomposed by the InkKit recognition engine into its person and connector components

```
Organization Chart Contents

Person "boss"
is a superior of "Lackey"
is a superior of "Minion"
has no superiors

Person "Lackey"
has no inferiors
is an inferior of "boss"

Person "Minion"
is a superior of "Worm"
is a superior of "Molecule"
is an inferior of "boss"

Person "Worm"
has no inferiors
is an inferior of "Minion"

Person "Molecule"
has no inferiors
is an inferior of "Minion"

Person "Nobody"
has no inferiors
has no superiors
```

Figure 14: Text output for an organization chart

598 and 443 lines of code respectively. These have been reduced to 320, 439 and 373 lines respectively, with a corresponding reduction in complexity and no effect on reliability. A corresponding reduction in development effort should apply to any module that exploits InkKit's new generic connector routines.

5 Discussion and further work

In this paper we have described the requirements for three different types of connectors; simple point-to-point connections (graphs), and two types of directional connectors, those governed by the connector syntax (arrows) and spatially inferred relationships (trees). The exemplars also demonstrate the full range of cardinality requirements, one-to-one, one-to-many and many-to-many. Developing these exemplars allowed us to identify the generic requirements for connectors in diagrams and to integrate this code into the InkKit diagram toolkit.

The simplicity of these connector algorithms belies the complexity of the relationships that they are capable of recognizing. Assuming InkKit correctly separates components, the connector algorithms can reliably be applied to almost all connected diagram types.

There are limitations on the recognition of direction of arrows. Our simplistic approach is to recognize the longer stroke as the shaft and shorter strokes as an arrowhead that is assumed to be pointing towards the end to which it is closer. This has been sufficient for us to explore the semantics of arrow directed connectors, however many diagrams use different techniques for indicating directionality, and InkKit might usefully be extended to incorporate them.

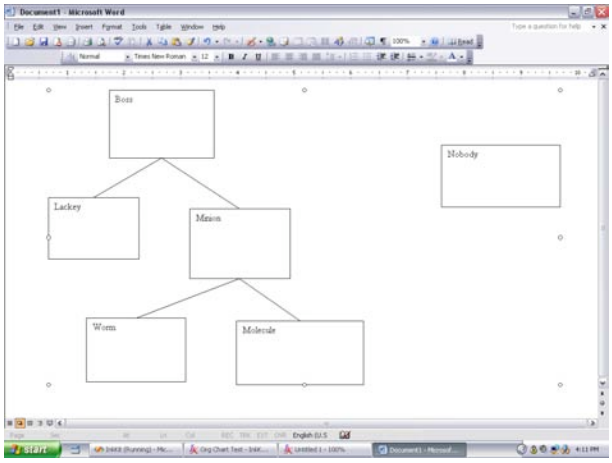

Figure 15: Output from an organization chart, built as a diagram in Microsoft Word

UML Class diagrams would serve as a useful exemplar for further development of connector recognition. They require that connectors be labeled to represent associations and constraints, and that they incorporate annotations at each end to represent cardinality. In addition, UML Class diagrams may incorporate several types of connector in the same diagram with different behaviors for each: inheritance relationships must be interpreted differently from composition relationships. InkKit is well suited to recognizing these different connector types as separate components, and processing each differently, but implementing all of these would be useful to further elucidate the general and specific features required for diagram and connector recognition.

InkKit's modular architecture and separation of generic recognition from domain-specific interpretation significantly reduces the effort required to develop new

```
for each shape in shapesList returned by recognition engine

    if shape is a Connector

        for each connectedShape in shape's list of connected shapes
            if connectedShape is lower on page than shape
                shape.AddSuperior(connectedShape);
            else
                shape.AddInferior(connectedShape);

        for each superiorShape in shape's list of superiors
            if superiorShape is a Connector
                for each inferiorShape in shape's list of inferiors
                    superiorShape.AddInferior(inferiorShape);
            else if superiorShape is a Person
                for each inferiorShape in shape's list of inferiors
                    if inferiorShape is a Person
                        superiorShape.AddInferior(inferiorShape);

        for each inferiorShape in shape's list of superiors
            if inferiorShape is a Connector
                for each superiorShape in shape's list of superiors
                    inferiorShape.AddSuperior(superiorShape);
            else if inferiorShape is a Person
                for each superiorShape in shape's list of superiors
                    if superiorShape is a Person
                        inferiorShape.AddSuperior(superiorShape);

        remove shape from shapesList
```

Figure 16: Pseudocode for processing InkKit recognition results for an organisation chart.

sketch recognition tools, and opens many avenues for further development. The integration of basic understanding of connections into the core recognition further reduces the development effort. The domain specific techniques applied to undirected graphs, directed graphs and organization charts in this paper are applicable to a very wide range of diagram types, and the three discussed in this paper require only minor modifications to support recognition of genealogy charts, Feynman diagrams, Entity-Relationship diagrams, inheritance trees and others from many fields.

InkKit also has potential for development of output methods more specific to each type of diagram. Graph plug-ins could be augmented with algorithms to determine whether they are connected, or any of a variety other common algorithms from graph theory. Since InkKit produces output from logical representations, there is also potential for adding algorithms to "untangle" complicated sketches.

6 References

Arvo, J. and K. Novins (2006). Appearance-preserving manipulation of hand-drawn graphs. Graphite, ACM,61-68

Bailey, B. P. and J. A. Konstan (2003). Are Informal Tools Better? Comparing DEMAIS, Pencil and Paper, and Authorware for Early Multimedia Design. CHI 2003, Ft Lauderdale, ACM,313-320

Bailey, B. P., J. A. Konstan and J. V. Carlis (2001). DEMAIS: Designing Multimedia Applications with Interactive Storyboards. ACM Multimedia,pp. 241-250

Chen, Q., J. Grundy and J. Hosking (2003). An E-whiteboard application to support early design-stage sketching of UML diagrams. Human Centric Computer Languages and Environments, Auckland, NZ, IEEE,219-226

Chung, R., P. Mirica and B. Plimmer (2005). InkKit: A Generic Design Tool for the Tablet PC. CHINZ 05, Auckland, ACM,29-30

Coyette, A., S. Faulkner, M. Kolp, Q. Limbourg and J. Vanderdonckt (2004). SketchiXML: towards a multi-agent design tool for sketching user interfaces based on USIXML. Proceedings of the 3rd annual conference on Task models and diagrams, Prague, Czech Republic, ACM Press,75-82

Damm, C. H., K. M. Hansen and M. Thomsen (2000). Tool support for cooperative object-oriented design: Gesture based modelling on and electronic whiteboard. Chi 2000, ACM,518-525

Do, E. Y. L. and M. Gross (2001). "Thinking with Diagrams in Architectural Design." Artificial Intelligence Review(15): 135-149.

Elrod, S., R. Bruce, R. Gold, D. Goldberg, F. Halasz, et al. (1992). "Liveboard: A large interactive display supporting group meetings, presentations and remote collaboration." CHI '92: 599-607.

Fonseca, M. J., C. e. Pimentel and J. A. Jorge (2002). CALI: An Online Scribble Recognizer for Calligraphic Interfaces. AAAI Spring symposium on Sketch Understanding, IEEE,51-58

Goel, V. (1995). Sketches of thought. Cambridge, Massachusetts, The MIT Press.

Hammond, T. and R. Davis (2002). Tahuti: A Geometrical Sketch Recognition System for UML Class Diagrams. 2002 AAAI Spring Symposium on Sketch Understanding

Hammond, T. and R. Davis (2003). LADDER: A Language to Describe Drawing, Display, and Editing in Sketch Recognition. IJCAI,12-19

Igarashi, T. (2003). Freeform User Interfaces for Graphical Computing. 3rd International Symposium on Smart Graphics, Heidelberg, Germany, Springer,39-48

Kara, L. B. and T. F. Stahovich (2004). Hierarchical Parsing and Recognition of HandSketched Diagrams. UIST '04, Santa Fe, New Mexico, ACM Press,13 - 22

Landay, J. and B. Myers (1995). Interactive sketching for the early stages of user interface design. Chi '95 Mosaic of Creativity, ACM,43-50

Lank, E. H. (2003). A Retargetable Framework for Interactive Diagram Recognition. ICDAR, IEEE,185- 189

Li, Y., K. Hinckley, Z. Guan and J. A. Landay (2005). Experimental analysis of mode switching techniques in pen-based user interfaces. SigChi 2005, Portland, Oregon, USA ACM,461-470

Lin, J. and J. A. Landay (2003). Damask: A Tool for Early-Stage Design and Prototyping of Cross-Device User Interfaces. CHI 2003 workshop on HCI Patterns: Concepts and Tools, Fort Lauderdale, Florida

Lin, J., M. W. Newman, J. I. Hong and J. A. Landay (2000). Denim: Finding a tighter fit between tools and practice for web design. Chi 2000, ACM,510-517

Plimmer, B. E. and M. Apperley (2003). Software for Students to Sketch Interface Designs. Interact, Zurich,73-80

Rubine, D. (1991). Specifying gestures by example. Proceedings of Siggraph '91, ACM,329-337

Trinder, M. (1999). The computer's role in sketch design: A transparent sketching medium,. Computers and Building, CAAD futures 99, Atlanta,227-244