7

Recognition of Hand-Drawn Circuit Diagrams

by

Joanne M. Mikkelson

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degrees of

Bachelor of Science in Electrical Science and Engineering
and Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 1997

Author .....
Department of Electrical Engineering and Computer Science
August 27, 1997

Certified by .....
Paul A. Viola
Assistant Professor
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Recognition of Hand-Drawn Circuit Diagrams

by

Joanne M. Mikkelson

Submitted to the Department of Electrical Engineering and Computer Science
on August 27, 1997, in partial fulfillment of the
requirements for the degrees of
Bachelor of Science in Electrical Science and Engineering
and Master of Engineering in Electrical Engineering and Computer Science

Abstract

This thesis describes a partial implementation of a pen-based recognition system for hand-drawn circuit diagrams. The finished system will become a part of an electronic engineers' notebook. The system incrementally recognizes pen strokes as either wires or circuit elements. These are then connected through a series of rules. A C++ implementation will be described. Discussions include steps to take to complete the project, as well as touching on some general issues which need to be solved to recognize circuit diagrams.

Thesis Supervisor: Paul A. Viola
Title: Assistant Professor

Acknowledgments

First I'd like to thank my advisor, Paul Viola, for giving me the chance to work with him when I wandered by last spring. I've learned quite a lot since then, and he has always been understanding and willing to help.

I should also thank André DeHon and Tom Knight for helping me wander in the right direction.

David Krikorian has been a wonderful source of support during my work on this thesis. He has always been there to listen to my frustrations and complaints, to help me figure out what steps to take when I faced a tricky problem to solve, and to give me emotional support.

I'd also like to mention my friends, who, during the final stages of the thesis, were there when I took the time to look for them. It was always a great boost, helping me through the slow times.

Finally, a special word of thanks must go to Terri Iuzzolino; without her generous actions, I would not be where I am today.

Contents

List of Figures

Chapter 1

Introduction

Engineers and students often draw electrical circuits. Sometimes, when the circuit is finished, a student needs to sketch the transfer function of the circuit. Or, the circuit might be destined for a simulation, and would need to be translated into a non-pictorial form so that it may be understood by a simulation program. Or, even more simply, the circuit just needs to be stored away, so that it can be referred to later.

All of these functions could be carried out much more simply if only the circuit could be drawn on a computer, with the same interface as a piece of paper. The circuit could be drawn freehand with a pen, without requiring the use of a drawing program for which a complex user interface must be learned. The computer would then recognize the parts of the circuit, build up a correct description of the topology, and then send the appropriate representation of the circuit to a mathematical package, a circuit simulator, or a file on a hard disk. Then, an engineer need only draw the circuit once, and it can be used for many different types of analysis, converted into a component of a larger circuit, or printed onto paper for distribution.

This thesis documents a system under development which, when completed, will carry out these tasks. This circuit recognition system is part of a larger project called The Natural Log.

1.1 The Natural Log

The Natural Log is a project to develop an electronic engineers' notebook. It is a pen-based system; input is collected from the user by his drawing with a pen on a display. This display

collects input, and draws a line where the pen has traversed the display, so that the user can see where he has drawn. This interface is similar to the popular small pen-based computers such as the Newton or the Pilot, except that The Natural Log is larger, about the size of a paper notebook.

The Natural Log will be able to understand many kinds of input which an engineer might use. Minimally, text, equations, circuit diagrams, and sketches of functions will be accepted. As in a paper notebook, these different forms of input might be intermixed; text might be illuminated with a graph, equations might augment a circuit, and so forth. The Natural Log will be able to perform many useful actions for each form of input. A circuit might be given to a simulation package. Certain text might lead The Natural Log to do a document search, so that the user may see additional information on what is being written. Equations could be evaluated, simplified, or graphed.

The first step is recognizing the individual parts of the input. The Natural Log must identify strokes drawn by the user as circuit elements, or letters, or axes of a plot. Then, combinations will be built, to form a circuit, a paragraph of text, or an entire equation. These logical units can each be acted upon independently, or combined to produce another larger unit.

1.2 Preview

The rest of this document is devoted to discussing work on the circuit recognition aspect of The Natural Log. This work is not completely independent of other portions of The Natural Log, especially equation recognition. However, much of the work can be considered by itself.

Chapter 2 presents a more complete picture of the goals and progress of circuit recognition in The Natural Log.

In Chapter 3, details about every part of the circuit recognition in The Natural Log are presented. Some of this information is useful to anyone who will continue work on the project. Some is needed to understand the problems and solutions discussed in Chapter 4.

Chapter 4 discusses many things to consider in future work. The implementation of circuit recognition is not yet complete; proposals for completion are listed. More important are the subsequent observations on what the system has demonstrated to be insufficient

approaches, and what better methods can be employed.

Chapter 2

Drawing Circuits with The Natural Log

There are many aspects to understanding a circuit as it is being drawn. Ultimately, The Natural Log will serve as a useful tool in drawing, understanding, and creating circuits. The work discussed in this document takes steps towards achieving these goals.

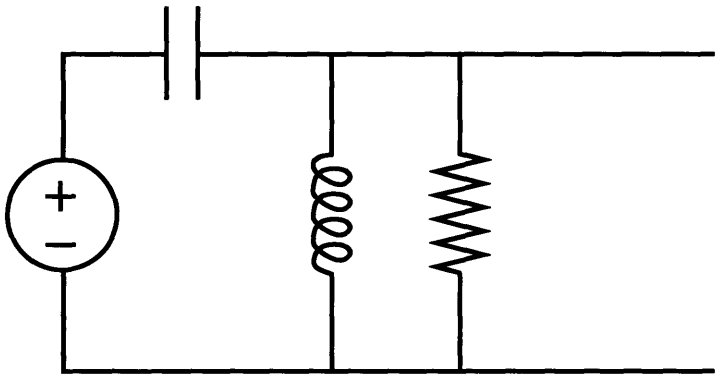

Figure 2-1: A Circuit

2.1 Goals

The first step in recognizing circuits is to recognize the individual circuit elements as they are drawn. Not only is it necessary to recognize single-stroke and multiple-stroke elements, but other lines must be recognized as wires. Wires can be drawn in an arbitrary fashion, so

almost any stroke may potentially be intended to represent a wire. It will be imperative to be able to distinguish a stroke which is a component of an element from a stroke denoting a wire. Success in this part of the circuit recognition problem is very important to a good system. Information about what is currently in the circuit will prove useful for making these distinctions.

After the elements and wires have been distinguished, the next step is to understand how they are intended to be connected together, as a particular assortment of transistors and resistors is not interesting until wired together. The drawing could be a bit sloppy; for example, two wires may not actually meet, or a wire which was supposed to terminate on another may overshoot. The Natural Log must be able to produce an accurate description of the topology of the circuit in such cases. Also important to this step is allowing nodes to be drawn, to connect wires otherwise assumed to simply cross and not connect.

A picture of a circuit alone might communicate enough, as in a drawing of a simple filter, or an amplifier. But elements need to be given values before the exact behavior of a circuit can be determined. The Natural Log will provide a method to input these values. The values might be written directly on the circuit, next to the elements. Then, the numbers might stay, or they might vanish, to reduce visual clutter.

Once the topology of the circuit has been determined and the values of elements are assigned, many interesting actions may be performed. A linear circuit produces a transfer function. The Natural Log could graph it, or suggest a simpler circuit which produces a similar function. A more complicated circuit might be translated into a format suitable for a simulation program such as Spice. The Natural Log might point out floating nodes. Certain element values could be marked as unknown, and constraints placed on others. The Natural Log need only recognize these cases and then use a simulator or search for information which will help determine the values needed.

As part of the interface, The Natural Log needs to be able to draw the circuit in a clear, clean fashion. If a sloppy circuit is drawn, and a neater one results, it will be easier to add to the circuit, or catch errors. It would be easy to connect elements by drawing wires across long distances, but this could be very difficult for a human to understand. The Natural Log, armed with information about the topology of the circuit, could straighten wires and route them around elements. Elements might be moved to a less crowded location. The Natural Log could also present changes to the overall layout which convey additional information

about signal flow through the circuit.

There may be different ways for a person to draw a circuit. He may do it in a scattered fashion, or from left to right. Different drawing styles may lend themselves to different ways of presenting the circuit. While some people might appreciate having each part of the circuit redrawn as they proceed, others might prefer instead that the circuit only be recognized and redrawn after several parts of the circuit have been drawn. The Natural Log must be able to recognize and redraw the circuit both incrementally and in batches of strokes.

2.2 Scope of This Work

This set of goals calls for a large body of work. Presented here is the scope of the problems chosen for initial work, and the progress made towards achieving these goals.

The problem of recognizing elements, wires, nodes, and the numbers and symbols describing element values is in itself a complex problem. A way to improve recognition and understanding of elements and wires, as well as recognition of numbers and equations for elements, involves using information about surrounding characters and circuit structure. As there is other closely related work for The Natural Log, recognizing multiple-stroke characters specifically in the context of equation recognition, this portion of the circuit recognition is not dealt with here.

The first simplification of this problem involved deciding that the picture for any element should be drawn only with consecutive strokes. That is, if the user is drawing a capacitor, he cannot draw the first |, then move to another part of the circuit and draw a resistor, then return and draw the second |. If he were to do so, the two halves of the capacitor would be interpreted as something else, most likely two unconnected wires.

As a further simplification, avoiding the initial recognition problem to a large extent, only one element was allowed: a circle. This allowed the topology to be worked on first. The topology recognition does not depend on the number of elements possible, and on only certain pre-defined aspects of what they look like. Hence, additional elements may now be added, without them having complicated earlier work on topology.

Currently, the circuit recognition will connect wires and circles together based on the distance between them. If the user draws a wire, each end of the wire is connected to the

nearest element, if it is within a certain distance of the end of the wire. If an end does not connect to an element, it may be connected to another wire. After this step, any leads of elements or ends of other wires are then connected to the new wire, again, if they are close enough to the new wire. Similarly, if the user most recently drew a circle, any ends of wires close enough to the circle will be connected to the circle.

2.3 A Sample Circuit

A clarification of what The Natural Log will do during the circuit-drawing process can be seen in a simple example circuit. The user starts by drawing a wire, figure 2-2. Only sharp

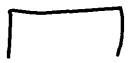

Figure 2-2: The first stroke drawn.

changes in direction will be detected as corners in the wire. This straightens the wires, making the circuit look better. It also reduces the number of separate parts of a wire which need to be dealt with by The Natural Log. Next, an element is drawn, seen in figure 2-3. The end of the first wire is connected to the element, and a lead is created, which is not

Figure 2-3: The first stroke has been redrawn (in grey), and the user has drawn another stroke.

visible in this case. If the user draws a third stroke as in figure 2-4, the element is connected to this newest wire.

In figure 2-5, the middle of the new wire will be connected to the older wire. Adding some more elements and wires in the same manner produces a simple circuit (figure 2-6).

The Natural Log will also route new connections to elements to the right place on the element, attaching them to the lead. If the line in figure 2-7 is now drawn, the wire will be connected to the lead on the element, not another point on the element, as seen in figure 2-8.

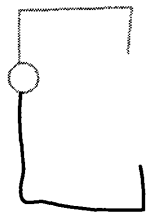

Figure 2-4: The second stroke was redrawn as a circle, again shown in grey. It has been connected to the first wire. The third stroke has also been drawn.

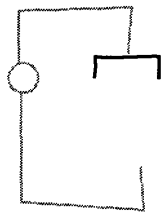

Figure 2-5: Another wire, which should be connected to in the middle.

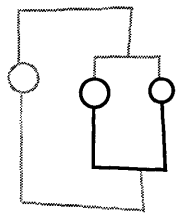

Figure 2-6: Note that the middle of the wire drawn in figure 2-5 has been connected. Shown is the resulting circuit after several more strokes.

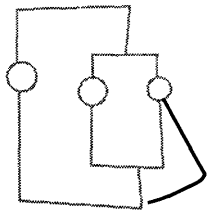

Figure 2-7: The newest stroke.

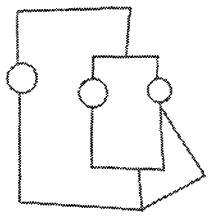

Figure 2-8: The final circuit. Note that the connection was made to the wire below the element, rather than the element itself.

This circuit illustrates many of the abilities of The Natural Log. A full explanation of the process by which this circuit is produced will be seen in Chapter 3, including a walk-through in section 3.7.

Chapter 3

Implementation

In this chapter, most aspects of the implementation of circuit recognition for The Natural Log will be discussed. After a brief overview of recognition in The Natural Log, sections 3.3 and 3.4 discuss the many data types in use. Section 3.5 explains how control is passed between these data types. The algorithms used for circuit recognition are examined in detail in section 3.6. Finally, the way in which the data, control flow, and algorithms fit together is clarified by returning to the example circuit seen in section 2.3, in which the steps taken to recognize and redraw each stroke of the example circuit are presented. Readers only interested in an overview are encouraged to read section 3.7 first.

3.1 Overall Project

Presently, The Natural Log runs in Windows NT. It is written in C++, using the Microsoft Foundation Classes (MFC) libraries.

Each time the user draws on The Natural Log, a stroke is generated. This stroke must then be interpreted in the context of previous strokes for The Natural Log to recognize circuits. The stroke is first recognized as a particular circuit object, for example, a resistor. Then the resistor needs to be connected to other parts of the circuit being drawn, if the user so intended. Then the resistor can be redrawn cleanly, connected properly to the rest of the circuit and drawn with straight line segments. At present, all of this processing is done in one class, `CRecognizer`, which will be discussed in further detail below (see sections 3.4.4, 3.5.3, and especially 3.6). A single instantiation of this class, which will be called the Recognizer, takes each stroke as it is drawn and makes a decision about the

action to take. Should this stroke be taken to be a part of an equation, or as part of a circuit? If the Recognizer labels the stroke as part of a circuit, the precise actions taken are independent of those needed if the stroke were part of an equation.

Making the details of circuit recognition independent of other forms of recognition is a simple division of labor which allows the project to easily develop in several directions simultaneously. This straightforward plan underlies the structure of the Recognizer as seen here.

3.2 Naming Convention

Following the naming convention of MFC, all classes are named with a "C" at the beginning of the name. Throughout the rest of this document, a reference to a class will be to the full name, e.g. `CRecognizer`. A reference to any given instantiation of a class will be referred to by everything after the 'C', e.g. the Recognizer.

3.3 Important Data Types

3.3.1 Strokes

A stroke, as mentioned above, is generated by the user as he draws. It represents primarily an ordered list of screen locations. The first point is the location of the pen down event. Each subsequent point represents a sample of the position of the drawing pen, until the final point, which is the location of the pen up event. The time at which the sample points are collected is not recorded. If the user sets the pen down and waits a long time before moving it, there are no extra samples at the pen down location, and after the fact, one cannot tell by looking at the data in a stroke that the user had waited, instead of moving the pen immediately. This means that since samples are taken only when the pen has moved, the samples can not be assumed to be taken at equal time intervals.

The class used to represent such a stroke is `CStroke`. A Stroke maintains the list of locations in an array. These locations are added to the Stroke one at a time, as the user draws. A Stroke also has members which keep track of the minimum and maximum x and y value of the sampled points, a length (the sum of the Euclidean distances between each point and the next sequential point), and another array, `dist_vec`, which holds the length

computed for each screen location which has been added to the Stroke. In other words, the first element of `dist_vec` is 0, then the distance between the pen down location and the first sample, then the sum of that distance and the distance between the first sample and the second sample, and so forth.

The length information is used to resample the Stroke during recognition. The methods used for recognition are not discussed here,[1] and `CStroke` was developed prior to the addition of circuits to The Natural Log. It is the most basic form of information about the user's input, so it is still vital to work on the circuit recognition problem. A Stroke can be thought of as principally a unit of data. The most important method of `CStroke` is used to add new points to the Stroke during data input.

3.3.2 Paths

An object of class `CPath` is similar to a Stroke, but more powerful. Paths are generated to serve as part of the representation of parts of the circuit. A Path may either be a series of points to connect, like a Stroke, or it may be an arc of a circle. A set of Paths can describe any part of a circuit which contains only line segments and arcs. Some potentially interesting elements of a circuit, like OR gates, can be represented this way, but will not look right, as the curves are not arcs of circles. If necessary, other kinds of curves may be added to `CPath`. Alternately, such a curve may be represented as a series of line segments. This sort of representation, at least for objects containing arcs, is avoided by `CPath`.

An arc is specified by the a Point at the center of the arc, another marking the beginning of the arc, another marking the end of the arc, and a radius. The arc is always assumed to proceed counterclockwise, the direction of positive increasing angle, from the starting Point to the ending Point. If the two points are at the same location, the arc will be interpreted as a circle, not a zero-length arc. If a Path is not an arc, it is a series of locations, connected in order.

A Path keeps track of its own bounding box. It also will compute such useful information as whether a line segment crosses part of the Path, the minimum distance of a point from the Path, etc.

`segmentCrosses(CPoint *segStart, CPoint *segEnd)` returns `TRUE` if the line segment

[1]This work is being carried out by Erik Miller, and will be published at some point in the future.

between **segStart** and **segEnd** crosses the Path at any point. Otherwise, it returns **FALSE**.

findConnectionPoint(CPoint *segStart, CPoint *segEnd) computes locations where the entire line described by **segStart** and **segEnd** intersects with the Path. A pointer to a Point at the location nearest to **segEnd** is returned. If there are no intersections, **NULL** will be returned.

findCrossings(CPoint *segStart, CPoint *segEnd) returns an array of pointers to Points. The Points are the locations at which the line described by **segStart** and **segEnd** intersects with the Path.

pointWithin(CPoint *pt, int howfar) returns **TRUE** if the Point **pt** is on the Path or lies within **howfar** pixels of the Path.

distance(CPoint *pt) returns the Euclidean distance from **pt** to nearest point on the Path.

The following methods may only be called if the Path is an arc. The angles employed here use the standard definitions of 0 angle and positive angle, as seen by the user; however, the window coordinates have increasing y values as one moves towards the user. Thus, the y-axis is flipped, and converting these angles into window coordinates and vice versa requires one more negation than normally seen. **computeAngleOf** and **computeLocationOf** are designed to be used together; if they are always used together, no problems should arise.

computeAngleOf(CPoint *pt) computes the angle of the line through the center point of the arc and **pt**, using the definitions above. The value returned will be between 0 and 2π.

computeAngleOf(double x, double y) is the same as above, except that the x-value and y-value are specified, rather than hidden in a Point. They can also be doubles this way; Points contain only integers.

computeLocationOf(double angle) is the inverse of **computeAngleOf**. This method returns a pointer to a Point which is at the location specified by the radius of the arc and **angle**.

A Path has other methods, needed to use the Path for drawing:

`isArc()` returns `TRUE` if the path is an arc, `FALSE` if the path is a series of line segments.

`boundingBox()` returns a pointer to a Rect describing the bounding box of the Path.

`linePoints()` returns an array of Points representing the Path. The first Point in the array is the start of the Path, and segments should be drawn between each Point and the next. This method can only be used if the Path is not an arc.

The following four methods apply only if the Path is an arc.

`arcCenter()` returns a Point representing the location of the center of the arc.

`arcStart()` returns a Point representing the location of the start of the arc. The arc, as described above, goes counterclockwise from this location.

`arcEnd()` returns a Point representing the location of the end of the arc. The arc proceeds counterclockwise from the starting location to the location returned by this method.

`arcRadius()` returns the radius of the arc.

Finally, there is a set of constructors, methods for specifying the important information for each kind of Path, and for resetting a Path so that new values may be used.

3.3.3 Recognized Objects

The Recognizer needs to be able to identify Strokes as specific objects: the number 2, the letter a, a resistor, a circle, etc. As any of these may be produced by the user, it is useful to have an abstract class for all objects which have been recognized as something specific. This class is `CRecObj`. There are no instantiations of this class; it is merely a parent class for all other kinds of recognized objects. As such, it has a minimal interface; it does not contain any data, but there is a small set of methods.

Each instantiation of a subclass of `CRecObj` (a RecObj) is expected to implement this set of methods.

`inputStrokes()` returns a pointer to a list of the Strokes drawn by the user which are considered part of the RecObj.

`isRedrawn()` returns `FALSE` if the RecObj should be displayed as the Strokes drawn by the user; returns `TRUE` if the RecObj should be displayed in a standard fashion.

The next two methods are related to a very simple method to provide run-time type information. `CRecObj` and each of its subclasses are assigned a unique integer identifier.

`getType()` returns the identifier for the class of the RecObj.

`isKindOf(int)` returns `TRUE` if the integer is the identifier for the class of the RecObj or one of its parent classes. Otherwise it returns `FALSE`.

Calling code should take care not to use `getType` in situations where the return value is tested against the identifier for a parent class. If the object is a subclass, the identifier for the subclass will be returned, and a test for the parent class will fail. Code like

```
switch(obj->getType()) {
  case T_ELEMENT:
```

should be avoided, since the `ELEMENT` type is subclassed (section 3.3.6). The identifiers are assigned via an enum:

```
enum {T_RECOBJ, T_UNRECOBJ, T_WNODE, T_WIRE, T_ELEMENT, T_CIRCLE};
```

There are no other specifications for subclasses of `CRecObj`.

3.3.4 Nodes

Class `CWnode`,[2] a subclass of `CRecObj`, represents a node in the circuit. Every connection between wires and elements in the circuit is maintained by a Wnode. Each connection has two parts: a pointer to the specific RecObj connected to the Wnode, and a pointer to the location of the "other end" of the connected part of the RecObj. The location of the Wnode is expected to be at one end of a line segment representing part of the RecObj, and the second part of the connection is the other end of this line segment. If the Wnode was at one end of a simple L-shaped wire, the "other end" would refer to the location of the corner in the L. A Wnode may have any number of connections to a single RecObj, but it is required that there not be more than one connection for which the "other end" is the same. For elements with arcs in them, this paradigm must be stretched, as arcs do not have straight lines inherent to them. The intention of this representation of connections is to simplify computation of good layout for the circuit. For example, by looking at all the connections of

[2]The W stands for "wire"; there was already a `CNode` class in MFC.

a Wnode, the Recognizer might be able to determine that a certain location for the Wnode would allow each line segment connected to the Wnode to be strictly horizontal or vertical.

Connections are added to Wnodes by calling the `addConnection(CRecObj*, CPoint*)` method. The first argument is the RecObj being connected to; the Point describes the location of the "other end". Connections may be removed based on the location of the "other end" via the `removeConnectionAt(CPoint*)` method, or all connections to a particular RecObj may be removed by the `removeConnectionsTo(CRecObj*)` method. If there are more than three connections, the Wnode is drawn as a dot, •, otherwise it is invisible. A Wnode, unless specified otherwise, has no limit on the number of connections to other RecObjs. A maximum number of connections may be assigned to the Wnode; once that number is reached, the Wnode will not add any new connections. The `isFull()` method returns `TRUE` if the Wnode is in this state. It is advisable for calling code, when changing connections on Wnodes, to remove the old connection first, then add the new connection; otherwise, if the Wnode is full, adding the new connection first would fail.

The Wnode maintains a pointer to a Point which represents the location of the Wnode. This data member is public because it is commonly used by the Recognizer and other RecObjs. The pointer value is always used, so that when the Wnode moves, the change in location takes effect everywhere. Since the pointer is used, there was no purpose in protecting the data and then adding a method to return the pointer. Calling code must be careful to change neither the pointer value nor the data in the Point; Wnodes may only be moved by calling the `moveTo(CPoint)` method.

The remaining data members are: a pointer to a Stroke, for the case where the Wnode was drawn by the user, a pointer to a Path describing how the Wnode looks on the screen, a boolean for the return value of `isRedrawn()` (as described in section 3.3.3), and a pointer to the owner of the Wnode. The owner of the node can be any RecObj in the circuit, and it is that RecObj which is responsible for deleting the Wnode should it be appropriate. Assignment of the owner may be fairly arbitrary; typically if the Wnode's owner is deleted, the owner will change the Wnode's owner to another one of the Wnode's connected objects.

When a Wnode is deleted, it does not delete the Point representing its location. Thus, if nothing else refers to the Wnode's location, the calling code should delete the Wnode's location before deleting the Wnode.

As explained above, a Wnode may only be moved by calling the `moveTo(CPoint)`

method. The Wnode then alerts each RecObj connected to it by calling the object's `nodeMoved` method (see sections 3.3.5 and 3.3.6 for discussion of this method). The return value of `moveTo` is a pointer to a Rect describing the area which needs to be redrawn, because parts of the circuit have changed location. If nothing needs to be redrawn, `NULL` is returned.

To combine two Wnodes `A` and `B`, one should use `A->connectNode(B)`; `A` will assume the connections of `B` and alert each RecObj connected to `B` by calling `changeNode`. The `changeNode(CWnode *from, CWnode *to)` methods should assume that `A` has already assumed `B`'s connections; only data internal to a RecObj should be changed by its `changeNode` method. See sections 3.3.5 and 3.3.6. Like `moveTo`, `connectNode` returns a pointer to a Rect describing the redraw area, or `NULL`.

3.3.5 Wires

Class `CWire` provides the representation of the wires in a circuit. The parent class is `CRecObj`. A Wire can be described as an ordered set of locations; the Wire is displayed by connecting these locations with line segments.

The most critical part of a Wire is the collection of Wnodes and corners which describe the Wire. The endpoints of the Wire are each a Wnode. Between these Wnodes, called the startNode and the endNode, there may be any number of other Wnodes or corners. A corner is a location at which the Wire changes direction. No other circuit elements are connected to the Wire at a corner. Each connection to the Wire must be to a Wnode. During the lifetime of a Wire, a corner may need to be converted to a Wnode to allow a connection at that location. Connections may also be made to a point on a segment between two Wnodes or corners; the Wire achieves this by adding a Wnode there.

The `CWire` class contains a large amount of code; most of it is devoted to properly maintaining this list of Wnodes and corners. A few of the key methods related to this are:

`addNodeAt(CWnode *node, CPoint *where)` adds `node` to the Wire at the location specified by `where`. If `where` is at the same location as a Wnode already in the Wire, the Wnode in the Wire is absorbed by `node` via the `connectNode(CWnode*)` method. If `where` is at the same location as a corner in the Wire, the Wire converts the corner to `node`, adding connections to `node` as necessary. Finally, if `where` falls along a line between two corners or Wnodes A and B, `node` is added to the wire between A and

B, adding connections to **node** to the locations of A and B, and changing connections of A and B if they are Wnodes.

If **node** is at a different location than **where**, the Wire will change to pass through **node**'s location. The return value is either **NULL**, or a pointer to a Rect describing a rectangle to be redrawn, in the case where adding **node** changed the shape of the wire.

changeNode(CWnode *from, CWnode *to) is called by **to** during a call to its **connectTo** method (see section 3.3.4). The Wire will change all its pointers to **from** so that they refer to **to**. Also, any connections in other Wnodes to **from**'s location will be removed, and connections will be added to **to**'s location. As a final step, if the Wire has two references to **to** in sequence, that is, not separated by another corner or Wnode, the Wire will drop one reference. This is to prevent the connections of the Wnodes in the Wire from becoming inconsistent with those implicit in the shape of the Wire. The return value is **NULL**, if nothing needs to be redrawn, else a pointer to a Rect is returned.

nodeMoved(CWnode *node, CPoint *washere) is called by **node** after it has moved. The Point **washere** represents the location of **node** before it moved. The Wire uses this information to re-compute its representation. A pointer to a Rect is returned to describe the area needing to be redrawn.

~CWire() must look at each Wnode in the Wire. If the owner of the Wnode is not the Wire, it should not be deleted, and connections of the Wnode to the Wire are removed. Otherwise, the owner of the Wnode is the Wire. If the Wnode has only connections to the Wire, deletion of the Wire will result in the Wnode having no connections, so the Wnode should be deleted. Otherwise, the connections on the Wnode to the Wire are removed, and the owner of the Wnode is changed to one of the other RecObjs to which the Wnode is connected.

The Wnodes and corners are maintained in a list. Pointers to the startNode and endNode are kept track of separately, and are not in the list of corners and Wnodes.

A Wire maintains a list of Strokes input by the user which describe the Wire, along with a bounding box of these Strokes. The Wire also has a pointer to a Path which describes its

representation. Finally, as with most RecObjs, there is a boolean which is the return value of `isRedrawn()`, specifying whether the Wire should be drawn using the strokes input by the user, or drawn using the Path.

Creating a Wire involves building the wire by successively adding points to the Wire, using either the `addLine(CPoint*)` method or the `addLineAtBeginning(CPoint*)` method. The first Point added to the Wire, an argument to the constructor, becomes the startNode. Each time a Point is added using `addLine` the location of that point becomes the endNode, and the previous endNode is converted into a corner unless endNode has connections to other objects. `addLineAtBeginning`, as its name implies, adds a new startNode, converting the previous startNode to a corner if possible.

3.3.6 Elements

A circuit with only wires and nodes to connect them would not be very interesting. `CElement` is the parent class for all circuit elements. Its parent class is `CRecObj`.

While `CElement` will have subclasses, they are in general for special cases, not to allow a separate class for each kind of circuit element. Providing a class for each element is not difficult for a small number of elements, but as the list of supported elements grows, this becomes more difficult. Also, the user should be able to add an element to The Natural Log's vocabulary. This would be quite difficult if the process involved writing a new class and recompiling the appropriate pieces of The Natural Log. Hence there is a need for a general description of all elements, with specifics for each available at run-time, most likely stored in a file.

A description of a particular circuit element needs to inform The Natural Log what the element should look like when displayed. Each kind of element is assigned a unique integer as an identifier. Of great importance is the location of places on the element to which one can connect wires and other elements, places to which nothing should be connected, and places which need to have something connected. The latter case is provided by adding a Wnode to the element at that location.

When an object of type `CElement` is created, the `setElement(int)` method should be called. The argument is the identifier for the kind of circuit element that the Element should represent. When the Element is redrawn by a call to `setRedrawn`, it creates a list of Paths (Paths are described in section 3.3.2) representing itself, based on the description

for the particular element named by `setElement`. The Element also will add any Wnodes to its representation if they are necessary.

These Wnodes are set up such that they have a maximum number of connections one larger than the number of connections to the Element itself. This allows at most one connection to a Wnode directly on the Element; when a RecObj is to be connected to the Wnode, the Element will create a short Wire, to act as a lead. One end of this wire is connected to the Wnode in the Element, causing it to become full; the other end of the Wire is then free to connect to as many other Elements and Wires as needed. Hence the RecObj connecting to the Element will actually be connected to the other end of the lead. After this connection is made, other connections will be forced to connect to the far end of the lead, as the end connecting to the Element has a Wnode which is full.

In this manner, another RecObj may either connect to a Wnode on the Element, filling the Wnode, or it my connect to the end of a lead of the Element. There remains one other mechanism to connect an object to an Element. Consider a Wire whose startNode can connect to an Element. If the Element has only strictly defined connection locations, the startNode must connect to another Wnode on the Element, whose position is determined by the description of that particular Element. Some Elements, however, can have an unspecified number of connections in unspecified places. An obvious example is a rectangular box, often used to represent an abstraction. Only the user of the abstraction knows how many inputs and outputs the box can have. Another example might be a logic gate, say AND. When the user draws an AND gate, The Natural Log has no way of knowing if there will be only two inputs or ten. The region of an Element which can be connected to in an unspecified manner is a "connectable" Path. If our Wire's startNode can connect to part of a connectable Path, in much the same way as the startNode connecting to another Wire between corners, the Element will add a new Wnode at the location where the startNode would connect. Then, similar to the mechanism for connecting to a Wnode on the Element, a lead is created and connected to the new Wnode. The new Wnode's maximum number of connections is set so that it will be full, and the startNode of the Wire will be connected to the far end of the lead.

A list of Wnodes directly on the representation of the Element is maintained. Each element of the list is actually an instance of class `CElNode`, which has no methods and five public data members. In this manner, each Wnode is stored with additional information

necessary for changing the location of the Wnode if the representation of the Element changes by being scaled, rotated, or moved. One number informs the Element which Path the Wnode is on. If this Path is an arc, the angle at which the Wnode is found on the arc is provided. If the Path is a series of connected points, two numbers are needed: the first tells the Element which Point on the Path the Wnode is after; the second number is between 0 and 1 and describes the location of the Wnode along the segment between the Point described by the first number and the next Point in the Path. This information allows the Element to move each Wnode as necessary so that it remains at the same location on the Element.

An Element has a list of Paths, and also keeps track of which are connectable. Other data members of an Element include: an angle at which the Element should be drawn, otherwise when redrawn an Element will always have the same orientation; a list of Strokes drawn by the user to represent the Element; a set of Points to specify the bounding box of both the input Strokes and the Paths representing the Element; an integer to keep track of which kind of circuit element is being represented; and, as usual, a boolean specifying whether the Element should be displayed as redrawn or as the input Strokes.

Like `CWire`, `CElement` provides the following three important methods:

`addNodeAt(CWnode *node, CPoint *where)` will add `node` to the element, as described above. The return value is a pointer to a Rect describing the area needing to be redrawn due to changes in the representation of the Element. `NULL` will be returned if no redrawing is needed.

`changeNode(CWnode *from, CWnode *to)` changes all references in the Element to `from` to refer instead to `to`. Here, only the Wnodes on the Element are changed.[3] Then the Element calls `changeNode` for any Wires representing leads, if they contain `from`. A pointer to a Rect is returned if any redrawing is necessary, otherwise `NULL` is returned.

`nodeMoved(CWnode *node, CPoint *where)` updates the each Path representing the Element, since the Wnode has moved. The Point represents the previous location of the Wnode. Either `NULL` or a pointer to a Rect describing a redraw area is returned.

[3]However, in the current implementation, `from` should never be a node on the element.

Circles

One example of a circuit element needing a subclass of `CElement` is the circle. Currently circles are the only circuit elements The Natural Log will recognize. Circles are a special case because one proposed method for quickly achieving good recognition of circuit elements is that the user will draw a circle where an element should be. Then, by drawing the particular element desired inside of the circle, The Natural Log can match these Strokes against only circuit elements.

As this functionality is not present, the `CCircle` class offers little advantage over using the more general methods for creating Elements. An additional data member, the radius, is present. This is primarily used to create the Path.

3.4 Other Data Types

3.4.1 UnRecObj

This subclass of `CRecObj` is employed by the Recognizer to create an object whose class is a subclass of `CRecObj` in cases where the Stroke drawn by the user cannot be recognized as any particular part of a circuit. This is necessary because the Recognizer is required to produce a RecObj for every Stroke drawn by the user.

Since what the Stroke was intended for is unknown, the functionality is as minimally required by `CRecObj`. The `inputStrokes()` method returns a list containing the single Stroke making up the UnRecObj. The `isRedrawn()` method always returns `FALSE`, since an UnRecObj can only be interpreted as the original Stroke. Finally, `isKindOf(int)` and `getType()` behave as expected.

3.4.2 List Classes

The Natural Log uses lists based on the templated list classes provided by MFC. The templated class is `CTypedPtrList`. The templates require two classes specified, the MFC list class to base the list on, and the kind of object stored in the list. The two classes available to base the list on are `CPtrList` and `CObjList`, with the latter rarely used in The Natural Log. Hence, a list containing pointers to Points would be declared as `CTypedPtrList<CPtrList,CPoint*>`.

These classes did not have an assignment operator, and to add one, `CTypedPtrList` was subclassed, creating `CMyPtrList`. This is the class used throughout the system. This class was then subclassed to produce `CStrokeList`, a list containing pointers to Strokes, which is essentially `CMyPtrList<CPtrList,CStroke*>`.

The Stroke Registry

A subclass of `CStrokeList` provides an important service to The Natural Log. This class, `CStrokeRegistry`, has only one instantiation. Each Stroke, as it is created, is added to the StrokeRegistry. This is because Strokes, being essentially low-level data, may be referenced many times, by very disparate objects. Hence, any given object does not know if a Stroke should be deleted. There were two obvious solutions to the difficulty: implement reference counts as a part of `CStroke`, or maintain a list of all Strokes and delete them at the end of the session. Reference counting, as it is not provided in C++, would have made Strokes much more cumbersome to use and provided an avenue for many bugs. The StrokeRegistry implements the second solution, and is simple to use: any time a Stroke is created (unless it is known to be temporary), it is immediately added to the StrokeRegistry. After this, the Stroke requires no further attention. If it is known that specific Stroke will not be referenced, it may be removed from the StrokeRegistry and deleted, but in general this does not occur.

3.4.3 Painter

`CPainter` is a layer which is responsible for communication between the Recognizer and the windows maintained by The Natural Log. It is an abstraction designed to make porting code to another system easier, as the portion of The Natural Log most dependent on the platform and operating system is accessed through a standard interface defined by that of a Painter. The Painter can be expected to draw Strokes and Paths, redraw rectangles, and even draw an Element properly. Extensions to `CPainter` would allow it to control other parts of the user interface.

The Painter is initialized with a pointer to the Recognizer's lists of RecObjs to draw and Strokes to draw, and the window to draw in. The Painter controls many aspects of the output. If, for example, it is desired that lines can be changed to be drawn a different width, it is the Painter which should have a method to change this property.

`redraw()` redraws the entire window.

`invalidateRect(CRect&)` invalidates the window only inside of the rectangle specified, so that only that area will be redrawn.

`redrawAll(CDC*)` will draw every RecObj and Stroke on the drawing context provided. The window will call this method when it has been requested to redraw, which occurs after events such as a window resize.

`getCurrentPen()` returns a pointer to the Pen object currently being used to draw lines.

`drawLine(CClientDC *dc, CPoint &prev, CPoint &curr)` draws a line segment from `prev` to `curr` on the drawing context using the current Pen.

`drawStroke(CStroke*, CDC*)` draws the Stroke on the drawing context, using the current Pen.

`drawArc(CPoint *start, CPoint *end, CRect *bb, CDC *pDC)` draws an arc starting at `start` and going counterclockwise to `end`. The bounding box `bb` determines the radius of the arc.

`invalidateRectangle(CRect, int extraGutter = 0)` is meant to be called by the Recognizer; `invalidateRect` is called with the rectangle specified by the Rect, enlarged in each direction by `extraGutter`.

`invalidateRectangle(CRect*, int extraGutter = 0)` same as the above, but taking a pointer to a Rect.

`drawWire(CWire*, CDC*)` draws the Wire on the drawing context. If the Wire is redrawn, its Path is used, otherwise, the Strokes returned by the Wire's `inputStrokes` method are drawn.

`drawElement(CElement*, CDC*)` same as for `drawWire` except that an Element may have more than one Path, so they are all drawn.

`drawCircle(CCircle*, CDC*)` draws the Circle on the drawing context; generally called by `drawElement`.

`redraw(CRecObj*, int extraGutter = 0)` redraws the RecObj, widening the redraw area in each directly by `extraGutter`.

`drawPathList(CMyPtrList<CPtrList,CPath*>*, CDC *)` is generally called by methods such as `drawWire`; each Path in the list is drawn on the drawing context.

3.4.4 Recognizer

The Recognizer, an instantiation of `CRecognizer`, which has been discussed briefly earlier, does most of the work for The Natural Log. Each Stroke created by the user is handed to the Recognizer, which then must attempt to recognize the Stroke as an wire, single-stroke element, one stroke of many making up a multi-stroke element, and so forth. With support for writing element values directly into the circuit, the Recognizer will also be called upon to distinguish numbers, letters, and other symbols. After this step, the Recognizer must determine if a newly-drawn Wire or Element should be connected to earlier Wires or Elements. Then, if this is the case, the connections are made, and if necessary, the window will be redrawn to display changes to the circuit.

Most of this work is done by private, internal methods. The external interface of `CRecognizer` is fairly simple. As a user draws in The Natural Log's window, Strokes are created. When the pen is set down, the Recognizer's `newStroke()` method is called. This method will create a Stroke, add it to the Recognizer, and return a pointer to the new Stroke, which at this point contains only one Point. As the pen moves, the window collects data about the movement and stores Points in the Stroke. When the user lifts the pen, the Stroke is done, and the window calls `strokeDone()`. This method alerts the Recognizer that the most recently created Stroke is now complete. If appropriate, the Recognizer can then identify the Stroke and connect it to the circuit. The Recognizer will change the appearance of the circuit as necessary.

Some data members of interest are `m_recObjList`, a list of all RecObjs produced by the Recognizer; `m_inputStrokes` a list containing all Strokes drawn by the user, in the order drawn; and `m_outputStrokes`, yet another list, containing only Strokes which the Painter should display. A short list of Strokes, `m_currentStrokes`, holds a number of recently drawn Strokes, in order, which are expected to be used for Element recognition. See section 3.5.3 for a description of the use of this latter list.

Section 3.5.3 also discusses the many steps taken by the Recognizer in processing a Stroke, and section 3.6 describes the algorithms used.

3.4.5 Window Classes

Descriptions of The Natural Log to this point have referred to "the window", which the user draws in. A general description is desired, as the details of the user interface have changed over time, and will continue to do so. The Natural Log has been designed with this in mind. The user interface need only construct Strokes and use the appropriate methods of `CRecognizer` and `CPainter`, and the recognition process will continue to work.

The user interface for The Natural Log differs from the simpler one used for development of the circuit-drawing portion of the Recognizer. Since both provide the same information to the Recognizer, the interface has been referred to more generally as "the window".

A short description of the window classes provides a sense for the structure of the windows. The user interface used for development is taken largely from an example program provided by MFC. When the application is started, an instantiation of `CScribble` opens the first window, a MainFrame, which contains the menus and tool bar. Then, when a new document is opened (this happens by default on startup, but the user can create a new document afterwards), a ChildFrame is created. It is filled with a ScribbleView, which deals with representing a ScribbleDoc. The ScribbleView is the key component; it is the view which collects input from the user, including pen down and up events and movements of the pen, in short, the ScribbleView collects Strokes for the Recognizer. If the Recognizer changes the representation of something being displayed, it effects this change through the Painter. When window events such as redraws occur, the ScribbleView dispatches drawing to the Painter, providing the drawing context (a `CDC` or appropriate subclass).

Most of this complication is handled by the user interface window classes and the Painter. A newer interface for The Natural Log, which does not use the example code, does not support multiple documents. This reduces the number of windows necessary, as the view can inhabit the equivalent of the MainFrame. This interface is actually "dialog-based," meaning that the classes used are somewhat different, but the overall structure is very similar.

3.4.6 Other Kinds of Recognized Objects

As functionality is added to The Natural Log, other kinds of recognized objects will need to be added. Another grouping of RecObjs distinct from parts of a circuit would include characters. Numbers, too, would be RecObjs. Perhaps these would both be subclasses of a more general Glyph, whose parent class would be then `CRecObj`. Other symbols, such as μ, might be glyphs as well.

Combinations of RecObjs would also be RecObjs. A circuit example might be an amplifier, built of transistors and wires. An equation might contain a fraction, built of two number glyphs and an appropriately placed line.

While none of these RecObjs are to be seen in the circuit recognition discussed here, it can be seen that the Recognizer will eventually manage and combine a variety of different objects.

3.5 Overview of Control Flow

The broadest view of the flow of program control in The Natural Log starts at the drawing window. This window waits until the user has drawn a Stroke, and then passes control to the Recognizer. The Recognizer does not return control to the window until it is done processing the Stroke. As part of this processing, the Recognizer may use methods of the Painter, which in turn may return control to the window for specific tasks.

Control flow between the window classes, the Recognizer, and the Painter is not significantly more complex than in this short description. The most complex control is to be found within the Recognizer itself, as will be seen below.

3.5.1 User Input

Nothing interesting happens until the user draws a Stroke. It is with a pen down event that circuit recognition starts. A pen down event in the drawing region marks the beginning of a Stroke. To create the Stroke, the window calls the Recognizer's `newStroke()` method (see section 3.4.4). This method will return a pointer to a Stroke to be filled out by the window. At this point, the window samples the locations crossed by the pen, until the pen is lifted, generating a pen up event.

As the user moves the pen, a line is drawn between each new location sampled and the

previous location, so that the user may see where the pen has traversed the window. The window requests a pointer to a drawing context (a subclass of the MFC `CDC` class) on which the lines are being drawn. Then, the window dispatches the drawing to a Painter, which operates on the drawing context. After the line is drawn, control returns to the window when the pen is moved again.

When the user has lifted the pen, the Stroke has been populated with data representing the movement of the pen, and the Recognizer may begin to recognize the circuit element represented by the Stroke. The window passes control to the Recognizer for this purpose through the `strokeDone()` method. Control will not be returned to the user input portion of the window until the Recognizer has completed its work.

3.5.2 Output from the Recognizer

The Recognizer receives program control from the window, as discussed above. After it processes the most recent Stroke, it may be necessary to redraw part or all of the window, to reflect changes made to parts of the circuit. These changes may be a simple as redrawing an Element or Wire in a clean, rectified form. They may also be due to more complex processes, such as the Recognizer connecting Wires and hence changing their appearance because ends or corners have been moved. All of these changes in the window are achieved through methods of `CPainter`. These methods need to change some part of the state of the window. Hence control is passed from the Recognizer to the window through the Painter.

When added, other forms of output, such as writing a transfer function to a file, will not necessarily pass control to the Painter, or even result in control leaving the Recognizer at all.

3.5.3 Control Flow in the Recognizer

When the `strokeDone()` method of the Recognizer is evaluated, the Recognizer will determine which environment is the best in which to perform recognition. Examples of such modes are: circuits, equations, text, etc. When The Natural Log is performing circuit recognition, the Recognizer's private `recognizeCircuitMode()` method is called.

Recognizing the Strokes as Parts of a Circuit

In `recognizeCircuitMode`, the Recognizer must first identify the circuit element represented by the most recent Stroke. When the methods for this recognition, developed separately, are folded into the circuit recognition, several possible interpretations of each of the most recent Strokes will be produced, along with information about the likelihood of each. This will allow recognition of Elements requiring multiple Strokes. If the most recent Stroke is part of a multi-stroke Element, the Recognizer will determine whether that interpretation of the Strokes in the Element allocates a Stroke to more than one Element. If so, an alternate interpretation of the affected Elements must be produced.

However, as only one single-stroke Element is included in the circuit recognition at present, there is no need at present to compute whether an Element containing the most recent Stroke has overlapping Strokes, as overlaps are not possible. Hence the Recognizer continues on to the next portion of the circuit recognition process.

Another effect of the fact that full recognition is not yet implemented is also manifested in the way in which the most recent Stroke is identified. In order to provide a simple alternative to the full recognition, the Stroke is first tested to see if it represents a node in the circuit drawn by the user (a •, following convention). If so, appropriate action is taken, although nodes are not yet recognized. If not, the Stroke is tested to see if it is an Element, a circle in this case. If it is, the Recognizer connects the newly-recognized Element. Otherwise, the Stroke is recognized as a Wire. Since almost any Stroke may be a Wire, it is very likely that this final recognition will succeed. If the Stroke is a Wire, it is also connected into the circuit. Finally, if the newest Stroke has not been recognized as any of these three major parts of a circuit, it is converted into an UnRecObj (see section 3.4.1). When full recognition is added, only one step is needed for recognition, and decisions about whether the most recent Stroke is a node, part of an element, or a wire will be based on the interpretations returned and on the interpretations of previous Strokes instead of on a failure to recognize the Stroke as a particular object.

Each attempt to recognize the most recent Stroke results in a call to a method. These methods have no arguments, and will look at the list of recent Strokes maintained by the Recognizer, `m_currentStrokes`. They may only return a new instantiation of the appropriate kind of circuit object if the *newest* Stroke is included in set of the Strokes which

produced said object. Otherwise, `NULL` is returned. The algorithms for recognition will be seen in section 3.6.

After Recognition

The steps taken by the Recognizer after the recognition step depend on what was produced by identifying the part of the circuit represented by the most recent Stroke. In general, the object is redrawn. Then, if appropriate Elements or Wires are nearby, the object is connected into the circuit. Then the window is told to redraw areas so that the new representation of the circuit will be seen. Finally, the RecObj produced by recognition is returned to `strokeDone`.

If the most recent Stroke is a hand-drawn node, a Wnode is produced which then needs to be connected to the Wires which cross the area covered by the Wnode. This will connect Wires which crossed but were not connected. This code has not been written, but is straightforward.

If the most recent Stroke is an Element, the Element must be connected into the circuit. First, the Element is redrawn. This involves passing control to the Element by calling its `setRedrawn(BOOL)` method with `TRUE`. The Element will then generate as many Paths as needed to represent itself, scale a representation of the particular element in question, and set the Paths up to produce the redrawn representation of the Element. After the Element performs this task and any other modifications needed, a pointer to a Rect is returned. This Rect represents the area needing to be redrawn so that the changes to the Element are visible. The Rect is added to a list of areas to be redrawn. Then `adjustElement(CElement*)` and `adjustOtherObjsToElement(CElement*)` are called. Each produces a list of Rects representing areas needing to be redrawn because part of the element has moved, or a Wnode was added, etc. These Rects are collected into a list, and given to the Painter. When the Painter is finished redrawing the window, the Recognizer returns the Element to `strokeDone`.

If a Wire is produced by the recognition step, similar steps are carried out. As before, the Wire is redrawn by a call to `setRedrawn(BOOL)`. The Wire generates a single Path as its representation, and returns a pointer to a Rect representing the redraw area. Now `adjustWireEnds(CWire*)` and `adjustOtherObjsToWire(CWire*)` are called. The list of Rects is collected and sent to the Painter. Finally, the Wire is returned.

The Adjustment Methods

The adjustment methods `adjustElement`, `adjustWireEnds`, `adjustOtherObjsToElement`, and `adjustOtherObjsToWire` follow the same basic plan. Each cycles through the entire list of RecObj which have been generated by the Recognizer; this list is the data member `m_recObjList`. Each RecObj in the list except the most recently created one (which is assumed to be the Wire or Element being adjusted[4]) which is close enough to the Wire or Element is then passed to another Recognizer method, `getConnInfo`. This method will provide information about how the Wire or Element may be connected to the RecObj. See section 3.6.3 for more detail.

Each adjustment method uses this information to either choose the best connection, using `chooseBestConn` (section 3.6.2), or to connect several RecObjs to the Wire or Element. As mentioned earlier, each method returns a list of pointers to Rects; these rectangles represent the areas needing to be redrawn in order to reflect changes made to the representation of the circuit due to the connection process.

Back to `strokeDone`

After the most recent Stroke has been recognized, identified as part of an Element or Wire, and connected to other objects in the circuit if possible, and the circuit has been redrawn, control is returned to the `strokeDone()` method. At this point, the Recognizer can determine if the list of recent Strokes, `m_currentStrokes`, can be shortened. Strokes old enough to not be recognized as part of a new Element may be dropped from the list, in the order they were added. It is possible that re-interpretation of newer Elements may propagate backwards to cause re-interpretation of older Strokes, but in general, this should not happen for a significant number of Strokes. Taking care to not drop any Strokes of an Element until all Strokes in the Element are too old should reduce the chance of this occurring, and in the event it does happen, the Recognizer can then turn to the list of all Strokes, `m_inputStrokes`, to get more information.[5]

After this manipulation of `m_currentStrokes`, the Recognizer has completed its work for the most recent Stroke, and the new RecObj is returned. The window is once again in

[4]If it isn't, then something which should connect might not; adjusting a Wire or Element to itself will not cause any ill effect.

[5]With inclusion of the rest of The Natural Log, this particular method for dealing with multiple-stroke elements might be replaced.

control, ready to collect more user input.

3.6 Algorithms Employed in the Recognizer

Several important methods of the Recognizer remain to be discussed; these methods implement such things as production of information about possible connections, recognition of circles, and so forth. Before looking at the rules used to interpret the topology of circuits, it will be useful to briefly consider the rules the user follows as he draws the circuit.

3.6.1 Rules for People

There are a few basic rules, which when followed, make a circuit easier to understand. An obvious rule is that wires should not cross through circuit elements; wires should be drawn around the element. Circuits are often considered clearer if they are laid out such that corners in wires are right angles wherever possible, and elements are presented in locations which provide information about their purpose. By convention, two wires which cross are assumed to not be connected. If a node (represented by a •) is drawn at a place where wires cross, the wires are then considered to be connected together at that point. Preferably, when a wire is drawn crossing another but should not be connected to other wire, a half of a loop is drawn at the crossing point. An example is presented in figure 3-1.

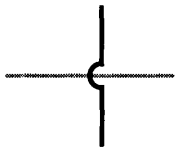

Figure 3-1: The loop denotes that the black wire is not connected to the grey wire.

People draw circuit diagrams in which the lines are crooked, and wires which should be connected to elements do not quite touch. Sometimes the end of a wire goes past the connection point. Cases like these, where parts of the circuit should be connected but the pen has not crossed the same point, do not affect the ability of people to understand that the connection was intended by the person who drew the circuit. The Natural Log must be able to deal with sloppy diagrams as well.

It is these situations which are of the most concern to The Natural Log. The most crucial portion of the circuit recognition process is to produce an accurate representation of what the user intended. While The Natural Log should be able to produce diagrams with straight lines and right angle corners, with no wires passing through elements, this is a separate problem involving a fair amount of computation for simply producing a nice output as seen by the user. Even better, The Natural Log might present a better layout for the circuit. These features may be developed and added incrementally, but they rely on information about the topology of the circuit.

Therefore, the goal of greatest concern is to create a mechanism for producing accurate decisions about how the parts of the circuit drawn by the user are intended to be connected.

3.6.2 Rules for Making Connections

Decisions about whether to connect two Elements or Wires are made based on several metrics. The most important is the requirement that the distance between the two locations to be connected is below the threshhold used to determine if a circuit object is close enough to another. After a connection under consideration passes this first requirement, others might come into play. One example of such a constraint requires that if the end of a Wire is being connected to another circuit object, the change in angle of the final segment of the Wire due to making the connection cannot be too large. Elements may have Wnodes which are full, disallowing any new connections. It should be noted that all of the constraints currently employed by The Natural Log are based on a small portion of the circuit. Hence, The Natural Log produces only a localized solution to the problem of deducing connections.

How the Recognizer Identifies Possible Connections

As mentioned in section 3.5.3, `getConnInfo` produces information about a possible connection between a Wire or Element and another circuit element. This information is stored in a struct internal to the Recognizer, `CONNECT_INFO`. How the Recognizer uses the `CONNECT_INFO`s produced by `getConnInfo` will be discussed shortly, in section 3.6.3.

When used to determine possible connections to a Wire, `getConnInfo` takes three arguments. In order to clarify the arguments, imagine that a Wire has just been drawn, and `adjustWireEnds` is testing to see if the startNode of the Wire can connect to another object in the circuit. The first argument to `getConnInfo` is a pointer to the Wire. The second

argument is a pointer to the particular Wnode on the Wire which is under consideration for connection to another part of the circuit, in this case, the startNode. The third argument is a pointer to the RecObj to which the Recognizer is attempting to connect the startNode. This is generally a Wire or an Element. `getConnInfo` has two overloads; one has a pointer to a Wire as the first argument, and the other has a pointer to an Element as the first argument.

In order to shorten `getConnInfo` and separate the algorithms for Wires and Elements, two methods, `getConnInfoWire` and `getConnInfoElement`, are called by `getConnInfo`. The former is used when the RecObj is a Wire, the latter if the third argument is an Element. Otherwise, these two methods have the same arguments and overloads as `getConnInfo` itself.

A large part of the code in `getConnInfoWire` and `getConnInfoElement` can be condensed into a simple explanation of the rules followed in connecting Wires and Elements to any other object and the rules followed in allowing an object to connect to a Wire or to an Element. The first test which must be passed requires that the Wnode passed as the second argument must not already have a connection to the RecObj passed as the third argument. If the Wnode is already connected, `getConnInfo` will return `NULL`. Otherwise, the following rules are used.

A Wire Connecting to Another Object Only the endpoints of the Wire may connect to another object. For a given Wnode representing one of the two endpoints, several conditions must be fulfilled to make a valid connection. For clarity, call this Wnode the startNode. First, the startNode must be close enough to the location of the connection. This is determined, as are all closeness measures, by the Euclidean distance between these two locations and a threshhold. If the potential connection on the other object is at a Wnode on the object, then that Wnode must be able to add all of the connections of the startNode without becoming full before all of the connections are added. Each potential connection on the object satisfying these conditions is then possible as far as the other object is concerned; the remainder of the conditions on the connection are due to the Wire itself. The angles of the segments of the Wire which terminate at the startNode are not allowed to change by more than 90° if the connection were to be made.[6] Finally, if making the connection would cause the Wire to cross itself, the connection is not allowed. If there

[6]This restriction could be relaxed in a number of different ways.

are more than one valid connections of the startNode to the other object, the connection point which is closest to the location of the startNode is chosen. If more than one connection point is at the closest distance, there is an ambiguity about the intention of the user, and no connection should be made; `getConnInfo` will return `NULL`.

An Element Connecting to Another Object Only Wnodes on the Element may connect to the other object. Wnodes on the Element's end of a lead should be full, and will not generate valid connections. For Wnodes on the far end of a lead, the rules for connection are identical to those listed above for a wire connecting to another object. Wnodes on the Element for which a lead has not yet been generated[7] are not part of any line segment and are expected to not move. Hence the connection rules are simpler. In the case where the Wnode on the Element is being tested for a connection to a different Wnode on the other object, the latter Wnode would be moved to the location of the end of a newly-created lead if connected to the Element. This eliminates all requirements on the Element except that the two Wnodes are close enough. The other object may have additional rules. In any other case, a line segment is needed to determine the location on the other object which should be connected to. A Point representing an initial guess for the location of the end of a lead is created, and the segment thus described by this point and the location of the Wnode on the element is then subjected to the same rules as for segments at the end of Wires listed above. Note that some Elements may not have any Wnodes in the initial representation; in this case no connections will be possible until Wnodes are added by another object connecting to the Element.

Another Object Connecting to a Wire Objects attempting to connect to a Wire should first attempt to connect only to the Wnodes on the Wire. This is because if the user draws a circuit object with an end very close to a node in the circuit, he probably wanted the two objects to connect. If the object connecting to the Wire cannot generate any valid connections to any Wnode on the Wire, the object should then attempt to connect to the corners of the Wire. It is common to draw a wire with corners in a circuit and then later connect another part of the circuit to these corners. If no connections to any corners are possible, then the object may try to connect to part of the Wire between corners and

[7]These Wnodes necessarily have been created by the Element evaluating the `setRedrawn(BOOL)` method.

Wnodes. This is generally achieved by describing a line which might intersect with the Wire and then testing the intersection point to find out of it is a possible connection point. The object connecting to the Wire will call the `findConnectionPoint(CPoint *segStart, CPoint *segEnd)` method of the Path representing the Wire. **segEnd** is always the location of the Wnode which was the argument to `getConnInfo`. The Path will then compute the closest point to **segEnd** which is on the Path and on the line described by **segStart** and **segEnd**. If this line does not intersect with the Path, `NULL` is returned. If a Point is returned, the object connecting to the Wire must then make sure that the distance between this crossing point and the location of **segEnd** is smaller than the threshhold. If so, the final requirement is that the crossing point is either between **segStart** and **segEnd** or past **segEnd**, that is, the crossing point must be on the line described by

$$\begin{pmatrix} \texttt{segStart->x} \\ \texttt{segStart->y} \end{pmatrix} + a \begin{pmatrix} \texttt{segEnd->x} - \texttt{segStart->x} \\ \texttt{segEnd->y} - \texttt{segStart->y} \end{pmatrix}$$

where a is positive.

Another Object Connecting to an Element The rules here are very similar to those for connecting to a Wire. The object should first try to connect to the Wnodes on the Element. If there are no valid connections, the object should try to connect to the connectable Paths of the Element. As for connecting to a Wire, a line is described, and `findConnectionPoint` is called for each Path. The same requirements on crossing points are imposed. Finally, if more than one Path on the Element may be connected to, the connection which is closest to **segEnd** will be returned. If more than one potential connection is at the smallest distance, the connection is ambiguous and `getConnInfo` will return `NULL`.

Choosing the Best Connection

The Recognizer, during adjustment of the circuit, may potentially collect up a large number of `CONNECT_INFO`s for a given Wire or Element. In cases where only one connection should be made, one of these `CONNECT_INFO`s must be chosen as the "best" connection possible. The `chooseBestConn(CONNECT_INFO *ci1, CONNECT_INFO *ci2)` method of the Recognizer serves this purpose. A pointer to the better connection of the two is returned.

If one of the `CONNECT_INFO`s describes a connection to an Element, and the other does

not, then the CONNECT_INFO for the Element is returned. Next, if one of the connections is to a Wnode and the other is not, the CONNECT_INFO for the connection to the Wnode is returned. Then, if both connections are to Wires, and one CONNECT_INFO is for a corner, and the other is not, the connection to a corner is preferable and is returned. The final metric is related to the distances stored in the CONNECT_INFO. There are two numbers: one is the distance of the Wnode attempting to make the connection from the connection point on the object being connected to, and the other is the distance by which the connection point on the object being connected to would have to move when the connection is made. This latter number is usually zero. It is desired that the object being connected to need move as infrequently as possible, so a simple weighting of distances is used. The CONNECT_INFO for which the sum of the first number and three times the second is smallest is returned.

3.6.3 The Adjustment Methods

`adjustElement(CElement*)` and `adjustOtherObjsToElement(CElement*)` are called by the Recognizer when an Element has been drawn. `adjustWireEnds(CWire*)` and its companion `adjustOtherObjsToWire(CWire*)` are used when a Wire has just been created. These four adjustment methods contain much of the logic regarding when Elements and Wires may be connected. As seen in section 3.5.3, they call `getConnInfo` and `chooseBestConn`. Which results are used when varies among the four adjustment methods, but otherwise they are all similar.

Here, `adjustWireEnds` is discussed in detail; differences between this method and the others will be noted later.

A Detailed Look at `adjustWireEnds`

`adjustWireEnds` attempts to connect the Wnode at each end of the Wire to other objects in the circuit. Arbitrarily starting with the startNode of the Wire, the Recognizer walks the list of RecObjs, `m_recObjList`, testing each RecObj, first to see if it is a circuit object, as a Wire should not attempt to connect to other kinds of RecObjs, such as numbers; then to determine if the startNode is close enough to the RecObj. The latter is determined by a simple threshhold on the Euclidean distance between the startNode and the nearest part of the RecObj. If the RecObj in question is a Wire, the Path representing the Wire can provide this information through the `pointWithin(CPoint *pt, int howfar)`

method (see section 3.3.2). When given the location of the startNode and the distance threshhold as arguments, this function will return `TRUE` if the startNode is close enough to the Wire. If instead the RecObj is an Element, a simpler metric is used. While a Wire always maintains a Path for its representation, an Element only generates Paths when it is redrawn. Furthermore, if the Element has open spaces, like a circle does, the startNode of the Wire might be inside of the Element and also too far from the lines making up the Element. Thus the startNode is close enough to the Element if it is either inside of the bounding box of the Element, or close enough to the bounding box. The Recognizer will use its private `isNear(CPoint*, CRect*)` method to determine this, passing the result of a call to `CElement::boundingBox()` as the second argument.

After another Element or Wire has been determined to be close enough, another important method of the Recognizer is called: `getConnInfo`. As explained in section 3.6.2, this function takes three arguments: the first is a pointer to the Wire whose ends are being adjusted, the second is a pointer to the Wnode of the Wire under consideration (at this point, the startNode), and the third is a pointer to a RecObj, which is the Wire or Element close enough to the startNode. If it is possible to connect the startNode to the RecObj, `getConnInfo` returns a pointer to a `CONNECT_INFO` struct. Otherwise `NULL` is returned. A `CONNECT_INFO` contains the information necessary to make a decision on an optimal connection for the startNode and to actually connect the startNode to the RecObj.

If `getConnInfo` does not return `NULL`, the `CONNECT_INFO` is added to a list containing `CONNECT_INFO`s. After each RecObj in the Recognizer has been tested, this new list is considered. If there are no `CONNECT_INFO`s in the list, then the startNode cannot be connected to anything. Otherwise, one connection must be chosen. The Recognizer uses the `chooseBestConn(CONNECT_INFO *ci1, CONNECT_INFO *ci2)` method to choose the best `CONNECT_INFO`. (This method is also discussed in section 3.6.2.) Once the best `CONNECT_INFO` is chosen, the `actuallyConnect(CWire*, CONNECT_INFO*)` method uses it to connect the startNode to the Element or Wire referred to by the `CONNECT_INFO`.

`actuallyConnect` uses the information in the `CONNECT_INFO` to create the connections. If the startNode is being connected to another node, then the `connectNode(CWnode*)` method of other node is called, so that it may absorb the startNode (see section 3.3.4). If the startNode is being connected to a Wire, then the Wire must add the startNode

to itself.[8] This is achieved by calling `CWire::addNodeAt(CWnode*, CPoint*)`; the first argument is the startNode, and the second is the location on the Wire to which the startNode should be connected. The Wire will then pass through the location of the startNode; see section 3.3.5. Finally, if the startNode is to be connected to an Element, in the same manner as with Wires, `CElement::addNodeAt(CWnode*, CPoint*)` is called (see section 3.3.6). The Rects returned by these methods and any calls to `CWnode::moveNode(CPoint*)` are combined to produce a single Rect representing the area to be redrawn, which is returned. `adjustWireEnds` adds this Rect to a list of redraw rectangles.

At this point, the startNode of the Wire has been adjusted, and the entire process is repeated for the endNode.

Noteworthy Differences

The `adjustElement` method contains the same ideas, but `CONNECT_INFO`s are collected by a different method. Each Wnode in the Element is added to a list. Paired with the Wnode is a list of `CONNECT_INFO`s. Then each RecObj in the Recognizer is checked, as above. For each Wnode in the list, if the RecObj is close enough, `getConnInfo` is called.[9] Each `CONNECT_INFO` generated is added to the list of `CONNECT_INFO`s of the appropriate Wnode.

It is necessary to build these lists because there may be situations where more than one Wire is drawn as a connection to an Element before the Element itself is drawn. If two connections are close and should connect to two different Wnodes on the Element, a simple distance measure as used for Wires would fail to produce correct connections if both connections are closer to one Wnode on the Element.

After the lists are built, all Wnodes on the Element for which only one `CONNECT_INFO` was generated can be connected. Once a particular Wnode on a RecObj is connected, all `CONNECT_INFO`s for that RecObj are removed from all of the lists. When this is done, the only remaining Wnodes to be connected are those for which more than one `CONNECT_INFO` is still valid. This is left as an ambiguous case, so no connections are made. The user may then draw more lines to connect the Element properly.

More complex is the `adjustOtherObjsToElement` method. This is because while an

[8]Note that if the startNode is connecting to a Wnode, this satisfies the first case. Hence it is known that a connection to a Wire requires a connection to a corner or a segment of the Wire, not a Wnode.

[9]Since `getConnInfo` is overloaded, the first argument in this case is an Element.

Element may only connect to other circuit elements by connecting Wnodes, the other parts of the circuit may connect to any "connectable" paths on the Element. So a different list is built, this time containing Points paired with lists of `CONNECT_INFO`s. As the Recognizer tests each RecObj, any `CONNECT_INFO`s generated are added to the list of Points. The location on the Element which is being connected to (the `otherPoint` field of a `CONNECT_INFO`) is stored in the list. If that location is already in the list, the new `CONNECT_INFO` is added to that Point's accompanying list. Otherwise, the location is added to the list, with a list containing the `CONNECT_INFO`.

When all of the RecObjs have been looked at, the final result is a list of locations on the Element which may be connected to. These locations may include both locations of Wnodes and locations on connectable Paths. Each location has a list of `CONNECT_INFO`s referring to the same location. The locations representing Wnodes on the Element are connected first. The connection described by the first `CONNECT_INFO` in the list for that location is connected with `actuallyConnect`. Then each subsequent `CONNECT_INFO` for that location is connected. To take into account that the first connection may have produced a lead if one was not present before, the subsequent `CONNECT_INFO`s are recomputed in that case.

After this process, the remaining Points represent locations on connectable Paths of the Element. Arbitrarily, the most recently created RecObjs are treated first. As done earlier with connections to Wnodes, the connection described by the first `CONNECT_INFO` is made, then further `CONNECT_INFO`s for the same Point are recomputed if necessary.

Finally, `adjustOtherObjsToWire` cycles through the list of RecObjs, testing each, as in `adjustWireEnds`. If a Wnode on the RecObjs is close enough and `getConnInfo` returns a `CONNECT_INFO`, a connection is possible, and the Wnode is connected. This, in effect, causes more recent objects to connect to the Wire first but if there is more than one RecObj which can connect to the Wire, the order in which connections are made should not be relevant.

Why Do We Need Both Directions?

One might ask why there are methods to adjust the new object, as well as methods to adjust other objects in the circuit to the newest one. Looking at the rules for valid connections as seen in section 3.6.2, it can be seen that only the endpoints of a Wire can be connected to another object. It does not make sense for the Wire to identify corners or locations between corners which other objects should connect to, as the most important information

is part of the other object. Hence, having an adjustment method which will attempt to connect the other object to the Wire provides for the case where a connection might be at or between corners without the Wire needing to maintain information about each location through which the Wire passes. Likewise, if an Element has no predefined Wnodes, it is difficult to determine where other RecObjs should connect to it, as the connection could potentially be to any point on the Element. `adjustOtherObjsToElement` deals with this case.

3.6.4 Other Algorithms

The algorithms used to recognize circles and create wires are temporary. As discussed previously, the problem of recognizing elements was laid aside to allow work on topology to proceed. Yet the algorithms used, especially for wires, are worth mentioning.

To create a Wire from a stroke, the Recognizer must locate the corners in the Stroke. Recall that a Stroke is a series of Points, and is drawn by drawing a line segment between each consecutive pair of points. The Recognizer uses these line segments to locate corners. First, the angle between each line segment and the next is computed, by normalizing the segments to a length of 1, and then taking the dot product between each consecutive pair. Then a simple threshhold is applied; larger angles are considered corners and the Point between the two segments is added to the Wire to become a corner. With the first and last Point in the Stroke added to the Wire, the Wire will be a simpler version of the Stroke, and for Strokes without slowly drawn curves, will have straighter lines.[10]

The circle recognition was written as a simple way to get reasonable recognition of circles, was derived quickly and empirically, and is not sufficient for general use. Circles are recognized in the following manner: a Point at the middle of the bounding box is created. Then the distance of each location in the Stroke from this center Point is computed. If the average distance divided by the minimum distance subtracted from the maximum distance is greater than the threshhold value 2.2, and the distance between the first and last Points in the Stroke is less than twice the difference between the maximum and minimum distance, the Stroke is identified as a circle.

As the full recognition necessary to recognize multiple and more complex elements is

[10]Slowly drawn curves tend to be full of single-pixel and other extremely short line segments, which can have large angle changes between them and still appear fairly linear.

added, these two recognition methods will change. The circle identification can be removed as a special case, and recognition will improve. The recognition of Wires could also be improved at this point; two possible avenues for improvement include making the threshhold vary depending on global conditions or using other recognition methods for determining where straight lines should be.

3.7 A Closer Look at the Example Circuit

Returning to the sample circuit seen in section 2.3, the various steps The Natural Log takes as the user draws can now be followed in detail.

The window waits for a pen down event. When this occurs, `CRecognizer::newStroke()` is called. The window adds Points to the new Stroke until a pen up event, then calls `CRecognizer::strokeDone()`. The user has drawn what is seen in figure 3-2.

Figure 3-2: The first Stroke.

The Recognizer evaluates its `recognizeCircuitMode()` method. The Stroke is first tested as an element. `testElement()` finds that the most recent Stroke, as seen above, is not a circle, and returns `NULL`. Hence `recognizeWire()` is called. The angle between each line segment is computed, and two corners are found. A new Wire is created, with the startNode in the lower left, two corners, and the endNode in the lower right. This Wire is returned. Back in `recognizeCircuitMode()`, the Recognizer adds this Wire to its `m_recObjList` and then proceeds to `adjustWireEnds(CWire*)`. This method goes through all but the most recent RecObj in `m_recObjList`, and since here there is only one element in the list, simply returns an empty list of Rects. The Recognizer calls `adjustOtherObjsToWire(CWire*)` and again, there are no other objects, and an empty list is returned. The two lists are combined, and the wire is redrawn by calling `CWire::setRedrawn(TRUE)`.

In this method, the Wire need do very little as it always maintains the Path representing itself. A pointer to the Rect describing the area needing to be redrawn is returned. This is added to the still-empty list of Rects by the Recognizer, and then

`CPainter::invalidateRectangle(CRect*,int)` is called for each Rect. The Painter calls the appropriate method of the window to invalidate each rectangle. Finally, the Recognizer returns a pointer to the new Wire, and control is returned to the window. See figure 3-3 for the result.

Figure 3-3: The Stroke has been recognized as a Wire, and redrawn.

The user then draws a second stroke, in figure 3-4. All actions performed are the same until `recognizeCircuitMode()`. This time, `testElement()` creates a new Circle to represent the most recent Stroke. This Circle is returned, and the Recognizer determines unsurprisingly that the Stroke making up the circle does not overlap with those in any other object in the circuit. The Circle is added to `m_recObjList` and redrawn by `CElement::setRedrawn(TRUE)`.

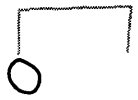

Figure 3-4: The newest Stroke is in black.

In this method, the Element will generate a new set of Paths necessary to represent itself. In this case, only one path is needed. After the Path is created, the Element will add all Wnodes which should be created, but as there are no predefined nodes in a Circle, none are added. A pointer to a Rect is returned. The Recognizer, still in `recognizeCircuitMode()`, calls `adjustElement(CElement*)`. As there are no Wnodes on the Element, nothing happens, and an empty list is returned. In `adjustOtherObjsToElement(CElement*)`, the list of RecObjs is traversed. The first and only RecObj on the list which is not the new Element is the Wire. `getConnInfo(CWire*, CWnode*, CRecObj*)` is called, first with the startNode. The startNode of the Wire cannot connect to any Wnodes on the Element, as there are none. The segment described by the startNode and the first corner (upper left) is passed to `CPath::findConnectionPoint(CPoint*, CPoint*)` so that the Path repre-

senting the Circle may compute the nearest location on the circle where the line segment crosses the circle. The Point returned is placed in a `CONNECT_INFO`. The endNode of the Wire is too far from the Element, so `getConnInfo` is not called. Only one valid connection to the Element is found, so the startNode is connected to the Element by a call to the `CRecognizer:actuallyConnect(CElement*, CONNECT_INFO*)` method.

The connection is actually made by `CElement:addNodeAt(CWnode*, CPoint*)`. A lead is built, with one end at a new Wnode residing at the location described by the Point returned by `findConnectionPoint`. The other end is about five pixels away, along the line described by the connection point and the startNode of the Wire. The startNode is then connected to the Wnode at the far end of the lead by calling the startNode's `CWnode::connectNode(CWnode*)` method. In this method, the startNode assumes the connections of the Wnode at the end of the Wire. There is only one connection in this case. `connectNode` results in a call to `CElement::changeNode(CWnode *from, CWnode *to)`; the first argument is the Wnode that was at the far end of the lead and the second argument is the startNode of the Wire. The Element will pass this method on to the Wire representing the lead by calling `CWire::changeNode(CWnode *from, CWnode *to)`, which will change the lead's pointer (which happens to be its endNode) to point to the startNode.

When the connection is finished, `adjustOtherObjsToElement` redraws and returns. The Recognizer returns the Circle and control is returned to the window to collect more input. The result can be seen in figure 3-5.

Figure 3-5: The second Stroke has been recognized, redrawn as a circle, and connected to the previous Wire.

If now the user draws a third Stroke, as in figure 3-6, this will be interpreted as a Wire. This time, `adjustWireEnds` has two RecObjs to look at. Only the startNode of the Wire is close enough to any other object, the Element. In `getConnInfo`, the Wnodes on the Element are first tested. None are close enough to the startNode, so a connection point is found on the Path representing the Element. After the connection is made, creating a lead, `adjustOtherObjsToWire` is called. The Wnode at the end of the lead just created is

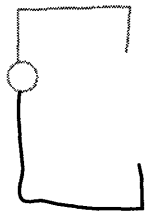

Figure 3-6: The newest Stroke.

already connected to the Wire, so a connection cannot be made. No other object in the circuit is close enough, so no further connections are made.

When the next stroke is drawn, yet another Wire is created (figure 3-7). The endpoints

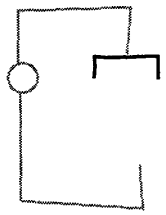

Figure 3-7: The second Wire has been redrawn, and connected to the circle.

of this new Wire are not close to any other circuit element, so `adjustWireEnds` does not make any connections. However, during `adjustOtherObjsToWire`, the first Wire drawn generates a connection from its endNode to the newest Wire. Since the endNode is too far from the endpoints of the newest Wire, it cannot connect to them. Corners are tested next, and these are also too far away. Thus, the Path representing the newest Wire generates the Point at which the connection can be made. This connection is the only `CONNECT_INFO` generated, so the connection is then made. Next, the user draws two more circles, which are each connected in the same manner as the first (to the startNode and endNode of the most recent Wire, respectively). A final Wire is drawn; its startNode and endNode are each connected to a Circle by `adjustWireEnds`, and the bottom Wire is then connected to it through `adjustOtherObjsToWire`. See figure 3-8.

A final Stroke is drawn (figure 3-9). In `adjustWireEnds`, the startNode is near an Element. When `getConnInfo` is called with the newest Wire, the startNode, and the Element, the Wnodes of the Element are checked first. Note that the startNode cannot

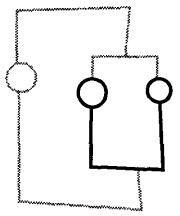

Figure 3-8: The three most recent Strokes have all been connected and redrawn.

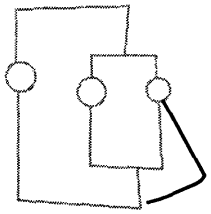

Figure 3-9: The final Stroke to be recognized.

connect to the Wnode on the Element's end of the lead, because that Wnode is full. This forces the startNode to be allowed only to connect to the far end of the lead, which is the desired behavior. Since a connection to a Wnode was possible, connectable Paths on the Element are not considered. If the startNode was farther away from the lead, a connection to the lead would not be possible, and then a connection would be created on the Path of the Element at the appropriate place.

This illustrates the critical difference between Wnodes and the rest of the representation of a circuit object. The Wnode, in effect, attracts connections in order to avoid making excessive new connection points. While many of these would not affect the final topology of the circuit, they make the circuit more complicated and harder to understand.

The `CONNECT_INFO` for the Element is not the only one generated. A connection is also possible to the endNode of the Wire connected to the Element. This proves to be two connections to the same Wnode, but this is not necessarily true in general. Since there is more than one valid `CONNECT_INFO` for the startNode, one must be chosen. The Recognizer uses `chooseBestConn(CONNECT_INFO*, CONNECT_INFO*)` to make this decision. The connection to the Element will be chosen, as connections to Elements are preferred. In this case, it would not matter which is chosen. If there were two connections generated to

the same Wnode, and both connections were to Wires, `chooseBestConn` would return the first connection. This final choice, obviously, is not relevant.

`adjustWireEnds` then moves to the endNode of the newest Wire. This Wnode is only close enough to one RecObj, a Wire. No Wnodes of the Wire are close enough. However, a corner is, and a connection is possible. Since a corner produces a valid connection, the possibility of the newest Wire connecting to a location between corners or Wnodes of the other wire is not considered. This is very similar to the precedence of Wnodes in Elements, except that there is an added level, the corners. Only one `CONNECT_INFO` is produced for the endNode, so that connection is made, producing the final circuit, in figure 3-10.

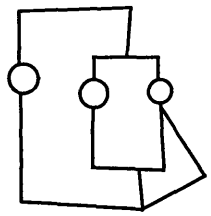

Figure 3-10: The final circuit.

Chapter 4

Future Directions

Work on circuit recognition for The Natural Log is not yet complete. With the addition of a few more pieces of functionality, the current system could be quite useful. What is needed is discussed first, in section 4.1. Circuit recognition also needs to be folded into the rest of The Natural Log, currently being worked on separately. This integration is necessary for such things as allowing the user to write an equation for the value of an element. The interface will also improve, and the allocation of resources between different parts of The Natural Log will change. This is seen in section 4.2.

But what has been learned from the work to date? It can now be seen that several parts of the circuit recognition, and even The Natural Log in general, do not work sufficiently well to provide a final solution. Section 4.3 discusses what does not work well. This knowledge provide some direction on what improvements should be made to the system. As seen in section 4.4, there are many possibilities, ranging from small changes in the representation of data, to more comprehensive changes, which if undertaken, would essentially require a rewrite. At least several good questions have been produced, which are discussed in section 4.4.3. Answering these questions early in development will help produce a more effective system.

4.1 Complete the Implementation

With the addition of several features, the circuit recognition in The Natural Log would be fairly complete and useful. The first addition necessary is the ability to recognize elements other than circles. This is the most difficult and complicated feature of those necessary for

a useful implementation. Other necessary features are comparatively less work.

Of the two additions listed here which are not necessary, code to draw the circuit better is the largest task. With the addition of this code, The Natural Log would draw circuits with right-angle corners and a cleaner layout. This problem could prove to require as much code as the rest of the circuit recognition; see section 4.1.7 and section 4.4.2 for further discussion.

4.1.1 More Elements

One of the first changes necessary is to add recognition and use of at least several different kinds of elements.

Recognizing Elements

As discussed briefly elsewhere (primarily in sections 3.5.3 and 3.6.4), work on recognition of multi-stroke characters has progressed in parallel to the circuit recognition. While incorporating this work will increase the number of elements which are recognized, adequate recognition may only be achieved if elements are drawn in a particular orientation. Since, unlike letters and numbers, there are no restrictions (except for aesthetics) on the orientation of circuit elements. Additional work must be done to allow for this situation.

One workaround which would allow a circuit to be drawn correctly, would be to expect all elements to be drawn first as circles. The user then may draw the element desired *inside* a circle, in the appropriate orientation. The recognition phase would then only have to deal with one or two orientations of any given element. The Recognizer would then be required to determine a good orientation for the element, based on the connections which had previously been made to the circle in which the element was drawn. The angle at which the new Element should be drawn would then be passed to the Element, so that when the Paths representing the Element are created, they can be rotated as necessary.

When recognition develops to the point where circuit elements can be recognized at many angles, it will no longer be necessary to draw inside of circles. In this case, the angle representing the orientation will be produced by Recognizer in the recognition step, instead of afterwards. As before, this angle can be used by the Element to create the proper representation.

Redrawing of Elements

An Element must be able to redraw itself by creating a set of Paths which represent the Element to the user. As seen in section 3.3.6, each Element has a description of what it should look like, which Paths are connectable, and where Wnodes should be added to the Element during the `setRedrawn` method (also discussed briefly in section 3.7).

This sort of information should not be coded directly into the Recognizer. This makes the description of an Element difficult to change, and makes adding a new Element to the system prohibitive. Hence, an input file should be created to contain the description for each known Element. This small addition would then allow new Elements to be generated by the user of The Natural Log. This functionality is an important part of building a useful system for recognizing circuits.

4.1.2 Let the User Draw Nodes

As with a circuit drawn on paper, a circuit drawn with The Natural Log may need the addition of nodes to clarify that two crossing wires should be interpreted as connected; otherwise convention will indicate that the wires are not touching.

The first step towards adding this functionality is to recognize • as a Wnode. If the full recognition, when added, does not adequately recognize these solid circles, a special case would have to be added. Once a Stroke has been identified as a Wnode, several actions need to be taken. First, all circuit objects (Wires and Elements) for which the Wnode as drawn by the user covers a Path or Stroke representing the object must be found. Then, a connection to the new Wnode must be added to each. The Wnode should be visible to the user, even if there are few enough connections to the Wnode that it would not normally be visible.

4.1.3 Connect Nodes in the Middle of Wires

Another similar functionality of nodes is missing. If the user draws a new Wire on top of a Wnode in another Wire, the new Wire should be connected to the old at the location of the Wnode. Presently, this will only happen if the Wnode in the older Wire is either the startNode or the endNode.

Adding this functionality requires that when the new Wire is drawn, each other Wire is

checked to see if the new Wire crosses at any point. If so, each location at which the Wires cross needs to be checked; if a Wnode is located there, the Wires should be connected. The addition of a method to Wires and Elements which can provide information about crossings (section 4.4.1) will facilitate this. Also, because of the way in which Wires are redrawn, this probably will not be very accurate until connections can be determined before the Wire is redrawn, as discussed in the next section.

4.1.4 Find Connections Before Redrawing

One difficulty in using the circuit recognition in The Natural Log stems from the fact that all circuit objects are redrawn before connections are made. This causes trouble, especially when a Wire containing a curve is drawn. If the user draws a big curve in a new Wire because the middle should connect to another circuit element, like in figure 4-1, the wire

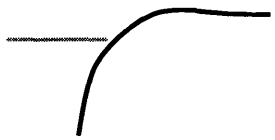

Figure 4-1: The black Stroke will be recognized as a Wire with a single straight line segment. This will cause the Wire to be too far from the grey Wire to connect.

recognition algorithm in the Recognizer (see section 3.6.4) will smooth out the wide corner. This smoothing causes the Wire to be quite some distance from the intended connection point, so no connection is made.

This can prove frustrating to the user, so it would be good to have The Natural Log check for connections before the Wire is redrawn and pulled away from possible connection points. Likewise, straightening the Wire too early can cause connections which were intended to be avoided by a curve in the wire.

While these two problems can be alleviated by lowering the sharpness of a corner necessary to be recognized as a corner, so that the resulting Wire looks more like the Stroke, this does not solve the most important problem: currently, it is not possible for the Recognizer to only redraw the circuit when requested by the user. Each Stroke drawn by the user must be redrawn before the next Stroke can be recognized. If this does not happen, the user will certainly be able to draw a circuit which appears to be connected but that the Recognizer

does not connect properly, because the redrawn representation may not resemble the hand-drawn representation closely enough. If circuit elements can be connected to before they are redrawn, the Recognizer can build up a correct topology as the user draws, without interrupting the user to redraw each Stroke.

Implementation

Since a single Stroke represents a Wire, adding the ability to create connections on a new Wire is not difficult. The Wire can be recognized, and then not redrawn. Then, when the Recognizer looks for possible connections, the Wnodes on the Wire may be checked first, then the corners. If no connections are possible for these two cases, connections may be made to the Stroke drawn by the user. Note that the Points of the Stroke as well as the line segments between the Points should be tested. If a connection is to be made in this case, a Wnode should be added to the redrawn version of the Wire at the appropriate location. When the Wire is finally redrawn, the Wnodes created this way will cause the redrawn Wire to pass through those locations, as was desired by the user. A data structure maintaining the correspondence between certain Points in the Stroke and the locations of corners and Wnodes on the Wire will facilitate the addition of Wnodes due to connections made to the Wire before it is redrawn. This information is also necessary for the case where two connections are made to a particular location on the Wire before it is redrawn; if the Wnode moves for the first connection, the Wnode's original location will be needed to test certain connection rules for the second connection, and the new location will be needed for other rules.

For Elements, creating connections on the version drawn by the user can prove more difficult. When the Element is created, the locations of automatically added Wnodes on the version of the Element drawn by the user must be determined. These locations would be different than those for the clean version generated by the Element itself. For multi-stroke elements where Wnodes should be created at ends of lines, for example, this may prove a difficult calculation, as the order in which the Strokes are drawn may vary, and even though an angle at which the Element is drawn may be provided by the recognizer, it will require very careful searching of all of the Strokes to determine the location of the Wnodes. This will require that either additional information be provided in the description of the Element, or that careful routines are provided which will match each part of a hand-drawn

Element against the Paths of a redrawn version. If the recognition step provides enough information about each Stroke which is a component of an Element, it may be possible to match locations without a large amount of extra information about what the Element looks like. Once the locations of the Wnodes are determined, the areas of the Strokes which correspond to connectable Paths must also be determined. This information is likely to be easily created while the Element is generating correspondences between locations of Wnodes.

The locations of the Wnodes on the Paths created by the Element and the corresponding locations on the Strokes drawn by the user must be maintained until the Element is redrawn. If a connection is to be made to a Wnode at the Element's end of a lead, little need be done, as this Wnode should not be moved. Wnodes at the far end of a lead may be moved to make a connection. If another connection to this Wnode is considered, the starting location should be used to determine if the Wnode is close, but the new location should be used to determine if making the connection would violate any rules for the object connecting to the Wnode. Connections made to connectable Paths should follow the same rules, but as making such a connection creates a lead, this case should be fairly simple.

If it is not required immediately that the Recognizer be able to recognize a circuit only after many Strokes have been drawn, or this capability is provided by delaying any recognition until after the circuit is completed, this functionality is less needed. However, it is still important to improve the interface for the user. Fortunately, this functionality is most important to the user for Wires, which are easier to implement than Elements.

4.1.5 Redraw-On-Command

If The Natural Log could recognize and redraw a circuit only when instructed to by the user, the drawing process would not have to be interrupted to redraw each Stroke drawn by the user. This redrawing can change the appearance of the circuit a great deal.

One easy way to implement this is to simply collect each Stroke drawn into a list of unrecognized Strokes, until the user requests that the circuit be recognized. Then, the Recognizer can look at each Stroke in turn, as if it had just been drawn, until all the Strokes are processed.

This could also be implemented fairly directly after code is written which allows Wires and Elements to be connected to before they are redrawn, as discussed in section 4.1.4.

In this case, recognition would again occur with each Stroke, but no changes would be presented to the user until he requests it. At that point, all Wires and Elements need only be redrawn to make the changes visible. The advantage of this approach is that recognition is performed incrementally, which allows The Natural Log to change the interpretation invisibly. Additionally, idle time could be spent checking the circuit to determine if certain connections should be different, improving overall performance. See section 4.4.2 for one possible example of such a process. If the circuit is not redrawn until the user desires, he does not have to observe changes made in this manner.

4.1.6 The Output of Circuit Recognition

Without the ability to analyze circuits, The Natural Log would be little improvement over paper. Several ways in which The Natural Log could evaluate and analyze circuits are presented in section 2.1. One means of circuit analysis involves writing out a representation of the circuit and using a preexisting analysis program to return results.

One disadvantage of including output at this stage of development is that after the output file is written, the user will have to edit the file in order to put the actual values of the Elements in. The Natural Log will have to be able to associate output from equation recognition with particular circuit elements before the output files can be written fully automatically (see section 4.2.2). In the meantime, it would not be too difficult for The Natural Log to request the element values from the user before writing the output, so that the file need not be edited by hand.

Spice

Spice is a common, powerful simulation tool which can produce data for the behavior of circuits containing non-linear elements, such as transistors.[1]

The advantage of using Spice for circuit analysis is that a translation from the representation of the circuit used by The Natural Log to that needed for input to Spice is fairly straightforward. The input file (called a "Spice deck") allows any number of nodes. Each need only be given a unique name starting with the letter n. All elements are described by first naming the element, then listing the nodes to which each terminal is connected, then

[1] In fact, the name "Spice" stands for "Simulation Program with Integrated Circuit Emphasis."

adding the value or values associated with the element. For linear, two-terminal elements like resistors, a line might look like this:

```
r1 n1 n2 10k
```

For more complex elements like transistors, more nodes and parameters are necessary.

Every piece of information needed to write a Spice deck is already present in the representation of the circuit, except for the element values. This, combined with the fact that Spice can handle very complex circuits, makes it an attractive engine for computation.

Transfer Function

Often, the sort of analysis desired for linear circuits may be as simple as graphing the frequency response, or a step response, or simplifying the transfer function. Hence, another useful output of The Natural Log's circuit recognition might be a transfer function. This transfer function could be handed to a mathematics package such as Maple, Mathematica, or Matlab. The results produced can be returned to The Natural Log, and displayed for the user.

To produce a transfer function, the impedance will have to added to the specification of an Element. Also, the locations of the pairs of Wnodes representing the input and output must be determined. If the circuit is drawn in a left-to-right fashion, a reasonable guess may be produced by locating the leftmost and rightmost pairs of unconnected Wire ends. Otherwise, The Natural Log will either need to ask the user when evaluation is requested, or a special character or gesture can be created so that the user may convey the information by drawing on the circuit.

4.1.7 Draw Circuits Better

As mentioned earlier, adding logic which will beautify the resulting circuit will be difficult. However, there are many different ways in which this functionality can be improved.

The Wire recognition method could be changed to pull corners to places which produce right angles in the Wire. Wires or Elements which pass over the bounding box of previously drawn Elements can be moved (at least one addition mentioned in section 4.4.1 will help with this). Elements can be drawn a fixed size, or only at certain rotations, i.e., at multiples of 45°. Wires which are parallel but not along the same line could be connected by a short perpendicular line, instead of changing the angle of one Wire. See figure 4-2 for an example.

Figure 4-2: At left, two Wires. In the middle is a possible result produced by The Natural Log. The right-hand picture would be a better result. Note that the top Wire has been shortened.

There are potentially many other rules for drawing a circuit nicely. Some of them will be difficult to implement, such as requiring that Wires have corners only at certain angles. This is because extra information about the representation of a Wire is needed while the Recognizer is making decisions about valid connections. Also, making a connection may require many parts of the circuit to move, including objects which are not directly involved in the connection. The structure of connections in Wnodes (a pointer to the object being connected to, and the other end of the line segment the Wnode is on) is designed to help, but that is not enough information.

4.2 Combine with Other Parts of The Natural Log

Significant progress has been made on The Natural Log since the undertaking of circuit recognition. To facilitate early development, the circuit recognition development was done separately from equation recognition. Integrating the two is an important step which must be undertaken before certain aspects of the project may be accomplished. One benefit of integration directly related to circuit recognition is that recognition of circuit elements will improve. The project as a whole will also need to develop ways to allow the user to freely use both equations and circuits in the same session, and then more modes of operation can be added. As there are many unanswered questions in this area in addition to those in the recognition of circuits and equations, the project will continue to evolve.

4.2.1 Adapt Recognizer

The largest differences between how the circuit recognition currently works and what will need to be changed lie in the Recognizer itself.

Modes

As discussed in section 3.1, the Recognizer will receive a Stroke from the window. This Stroke may be intended to be part of a circuit, or an equation, or something else entirely. One scheme for choosing how to interpret a Stroke involves recognizing it in a particular mode. If the last several Strokes were part of a circuit, it is reasonably likely the next will be also.

Obviously, the Recognizer needs to be able to change modes. The simplest way is to have the user click a button. But this is not a very natural way to draw in a notebook. A better choice would be to allow the Recognizer to evaluate each Stroke to determine if the mode should be changed. For example, a Stroke registering as extremely likely to be a q, for example, might cause the Recognizer to switch out of circuit mode, because there are not any elements which match well.

The use of modes, if it is sufficiently easy to change modes, would allow the Recognizer to interpret a horizontal straight line as a Wire, if Elements were drawn immediately before and after. Otherwise, it might be a minus sign. Modes can provide constraints on the use of a character, which will be important in producing a final interpretation.

Threads

The Natural Log is evolving to use a more complex, partially asynchronous processing of Strokes, taking advantage of multi-threading capabilities of MFC. As the Strokes are produced by the user, they are matched against models for all parts of The Natural Log: numbers, mathematical symbols, circuit elements, etc. A listing of possible matches is generated, so that a thread for equations and a thread for circuits can both evaluate the possibilities for the true identity of the stroke in their separate contexts. Thus each mode may look at the data, and each will publish its own interpretation of the Stroke.

The introduction of threads allows many different modes to test a given Stroke. This introduces greater flexibility, but at a cost: now an dispatcher is needed, to decide which mode each Stroke belongs to, based on how well it fits in each. History can be also used. After the decision is made, the modes to which the Stroke is not assigned may need to reinterpret surrounding Strokes, especially since more Strokes may have been drawn and interpreted before the decision was made.

Hopefully this task can be made simpler by introducing ideas such as the notion that parts of a given equation are likely to be laid out on a line. They will not be spread about the drawing area. Likewise, circuit elements should be relatively close, and connect together well.

In order to accommodate threads, several changes must be made to the Recognizer. Each mode as described earlier would become a separate thread. Hence, most of the Recognizer would be moved into a circuit thread. Then the thread must be given the ability to read Strokes. Most of the changes will be in the production of data. Instead of just producing RecObjs, the circuit thread may need to produce the entire interpretation of the circuit. Some metric of how well the new Stroke fits into the circuit will also be needed. The exact information produced will need to be decided upon during the integration process.

Another unknown is how redrawing will be handled. Options include: the circuit thread will produce information about what areas need to be redrawn, as it does now; the circuit thread will produce only RecObjs, and they are redrawn by another thread; and the circuit thread will retain control of the output area given it, and will redraw very much the way it does so now. The first two of these particular options depend on the existence of another thread which takes over the functionality of the Painter. The latter option does not need a painter thread, but some control must be exercised over which threads may draw where. It is quite likely that more than one thread would request to draw in the same place. Even if segmentation of Strokes between threads is perfect, an equation could be drawn on top of a circuit element.

The difficulties in coordinating asynchronous interpretation of the same Stroke and in redrawing will require much work. However, most of these decisions will not require much reworking of the circuit recognition, as the rules will not change, only the way in which the results of the rules are returned.

4.2.2 Matching Numbers with Elements

After Strokes are divided between threads, they can be interpreted as part of a circuit or an equation. The next step is to combine the interpretations of separate threads, if appropriate. If an equation is written next to an inductor, is the equation supposed to specify the value of the inductor? Or is the equation a proposed transfer function of the circuit? The Natural Log should do very different evaluations of the circuit depending on which interpretation is

correct.

If the Strokes in an equation or circuit are sufficiently grouped, it may be possible to treat them as a new kind of atom. Then, another thread could combine these groups. Rules for this might be determined empirically. It may prove that a good solution to this is very similar to a solution for grouping parts of an equation. This is a difficult problem to solve, and will be the subject of much further research.

4.3 What Does Not Work

Certain aspects of the circuit recognition do not work well. Most problems can be fixed; others would probably be left for a rewrite, should one happen.

4.3.1 Error Handling

Error handling in The Natural Log is insufficient. Very little has been done to decide how to deal with error cases. At present, the best mechanism for dealing with errors is to print a message to a console window and then proceed as best possible, which is inadequate.

One problem with error handling is that a functional programming style has been used in C++, a language not best fitted for this. An argument can be made that methods should be changed to return error codes, rather than the values produced by the method. Many methods would need to be able to produce several different error codes, and the calling code would become much more lengthy, as tests for the error conditions must be added. One possible alternative, provided by C++, is the use of exceptions. The use of exceptions would achieve the same goals, while easily allowing certain errors to be recoverable and others fatal; fatal errors would simply not be caught when the exception is thrown.

To use either method of error handling would require changing the code used in every part of The Natural Log. However, this is necessary if The Natural Log is to continue to work smoothly as new functionality is added. Also, while it is easy to add some error handling to each thread, a method for dealing with an error returned by any particular thread must be designed. Certain errors may be ignored, while others may not. Errors generated by one thread may provide information which needs to be passed on to other threads. No mechanism for these cases is in place at the present.

4.3.2 Too Much Code

A lot of code in the four methods which make up `getConnInfo`, discussed in section 3.6.3, is repeated in each. There are sometimes subtle variations between methods. This, coupled with the length of the methods, makes the code very difficult to understand. Breaking up the methods further might help, but the differences between the four cases will remain. This situation also makes the methods hard to maintain, and adding functionality requires a full understanding of each method.

A great improvement can be made by combining the connection methods into one method. It should be fairly easy to add a set of if statements at each difference in logic. Then, not only will code no longer be duplicated with minor changes, which is especially hard to maintain, but the differences between the different cases will be much more clearly laid out.

The four adjustment methods, seen in section 3.5.3, are similarly duplicated. These methods share fewer similarities; `adjustElement` and `adjustOtherObjsToElement` have a very similar process involved, which is quite different from that of `adjustWireEnds` and `adjustOtherObjsToWire`, which are themselves even more similar. However, these methods are also quite long and difficult to interpret; modification depends on a very complete understanding of what is already present. The adjustment methods are not as easily broken up as the connection methods, but each pair might be combined intelligently.

The implementation of a general ruleset for adjustment and connections would be further improvement (see section 4.4.2), because then both the adjustment methods and the connections methods would be very general. In this way, only the rules need to be modified to change or add behavior, and the code would be much cleaner.

4.3.3 Connections Maintained by Wnodes

Wnodes are very important to the circuit. Almost all information about what objects are connected and where resides in Wnodes. Most circuit elements can only find out what they are connected to by obtaining the listing of objects to which the Wnodes are connected. Unfortunately, the way in which Wnodes deal with connections needs improvement.

One quick improvement would be to add methods which allow the Recognizer or objects in the circuit to get all information on connections from a Wnode. Currently either a list

of objects to which the Wnode is connected, or a list of places to which the Wnode is connected, is available. These two pieces of information should be provided together as well.

More importantly, the choice of data stored in a connection proves to have been inadequate. As explained in section 3.3.4, a connection consists of a pointer to the object to which the Wnode is connected, and a Point representing the location of the "other end" of a line segment, since the Wnode is assumed to be one endpoint of this line segment. This was designed to make it easier for the Recognizer to determine ideal locations for Wnodes, as it can easily be determined if moving a Wnode will cause the line segments which it terminates to be horizontal or vertical. A Wnode which is part of a vertical line segment might be constrained to move only vertically, and likewise for horizontal line segments.

Since The Natural Log does not yet produce circuit diagrams drawn in a particular manner, as in section 4.1.7, this functionality of the Wnodes is not used. An important negative aspect of the functionality is that it creates very fragile data. Consider a simple case of a Wire with two end Wnodes and one between the endpoints. If the Wnode in the middle needs to be moved, the Wire is required to do a large amount of work in its `nodeMoved` method. Each Wnode on the Wire which is connected to the location of the middle Wnode must be located. Then, connections to that location must be removed, and new ones added, to reflect the change. The detail work required to maintain the proper connections of Wnodes on a Wire to other parts of the Wire and vice versa makes it very easy to create bugs, leaving poorly-built Wires in the circuit, which shortly leads to serious problems. Also, any small change in Wires or Wnodes can cause many different failures.

Elements place more requirements on Wnodes. Wnodes on the Element itself can be marked as full, to prevent new connections, but they cannot easily be marked as unmovable. Hence Elements have to rely on the Recognizer to make a distinction between these Wnodes and others, so that they will not be requested to move. The introduction of Paths which are arcs also complicates matters. The "other end" of a connection assumes that the Wnode is on a line segment, which is not the case when it is on an arc. It might seem easiest to make connections to either end of the arc, and add another special case to the Recognizer, but if the Path is a circle, another problem appears. There may not be more than one connection to a given location, but a circle is an arc with the beginning and end in the same place.

The difficulties described above demonstrate that the connection mechanism in Wnodes

is not robust enough. It is also not easily generalized, as shown by the problems arising with Elements. These problems will be alleviated if a better representation of connections is designed and implemented. See sections 4.4.1 and 4.4.3 for some additional discussion.

4.3.4 Adjustment Algorithms

The adjustment algorithms, despite being long and complex (as noted in section 4.3.2), are not complete. Each is missing tests for certain cases, or more cases are considered ambiguous than should be. See section 3.6.3 for a description of these four methods.

`adjustWireEnds` will attempt to adjust only the ends of a Wire. This is not adequate, as a Wire may be drawn on top of a preexisting Wnode. If this happens, the Wire should be connected to the Wnode (see section 4.1.3). In order to achieve this, any locations where the Wire crosses another Wire or Element must be examined, to determine if there is a Wnode at that location which should be connected to the Wire.

`adjustOtherObjsToWire` immediately makes a connection to the Wire as soon as one is found to be possible. This means that the most recently drawn RecObjs will be connected first, since the list of RecObjs is traversed from newest to oldest. All possible connections should be collected instead. This would allow the Recognizer to make the best connection, which may be based on criteria other than distance. This is very similar to `adjustOtherObjsToElement`. Then, other connections could be made if still possible.

`adjustElement` faces additional difficulties. If the user draws two parallel Wires, then an Element with two Wnodes which should connect to both Wires, the Recognizer must take care to connect one Wire to each Wnode. Possibly both Wires are closer to one Wnode, so factors other than distance must be taken into account. First, for each Wnode on the Element, all possible connections are collected into a list. Next, connections need to be chosen so that the combination of all connections made is optimal, since a Wnode on a given object may produce a connection for more than one Wnode on the Element. One possible method for choosing an optimal set of connections is to choose the set for which the sum of distances traveled by Wnodes in making the connections is minimized. This is a multi-variable minimization problem. During minimization, care must be taken to prevent a combination from being created which will break rules, such as causing two Wires to cross. A less optimal solution is currently in use; all Wnodes for which only one connection was produced are connected. `CONNECT_INFO`s for Wnodes on other objects which have been

connected are then removed from the lists for the other Wnodes on the Element. This is done until only Wnodes of the Element with more than one possible connection remain. It is possible that these Wnodes may be reasonably connected by choosing the connection with the minimum distance, but it is safer to leave these Wnodes unconnected, so that the user may then connect them by drawing more precisely. Fewer connections would be labeled as ambiguous if a optimization such a minimization of total distance is used.

`adjustOtherObjsToElement` cycles through each RecObj, and stores a `CONNECT_INFO` for each Wnode which can connect to the Element. Since other objects may connect to either Wnodes on the Element or connectable Paths, a list of `CONNECT_INFO`s is made for each *location* on the Element, not just each Wnode, as in `adjustElement`. Again, certain Wnodes may be able to connect to multiple locations on the Element, and the best combination of these connections must be found. An added complication is that connections on connectable Paths may have been generated very close together. Ordinarily, when the connecting objects are drawn after the Element, the first such connection would create a lead, and the others would connect to the lead. Similarly, close connections should not create many close leads. In this situation, the best of a set of near connections must be chosen. Then the other connections should be connected to the new lead, if possible. Otherwise, they can be connected to the Element to make another lead. It is not clear how to choose which connection should be made first; presently `adjustOtherObjsToElement` will arbitrarily use the first connection found, which would correspond to the most recently drawn circuit object of the set.

The best plan for sorting the connections might be to connect all `CONNECT_INFO`s which connect to Wnodes on the Element first, using the same optimization as in `adjustElement`. Then, remaining locations on the Element are on connectable paths. Another minimization problem arises; the total distance traveled by Wnodes to connect to the Element should be minimized, while also ensuring that no two leads are too close together. At present, the Recognizer attempts to group connections which are a certain distance from each other; however, this is not a sufficient method, as a number of connections is a row could be collapsed into one, where two or more separate leads would be desired. Fortunately, this situation will not arise often, as it would require that the user draw many connections to the Element before drawing the Element itself.

4.3.5 Wires on top of Others

If the first or last segment of a Wire is drawn directly on top of a segment of another Wire, some problems may arise. This is because the two segments are parallel, and on the same line. If only the first or last segment matches exactly, the startNode or endNode will be moved to the same location as the corner or Wnode at the other end of the segment. A more serious problem arises when more segments than this one match. Wnodes and corners should be removed from the end of the new Wire until the new Wire diverges from the older Wire. This is almost impossible to achieve in the current system, as this situation might not be discovered until a set of `CONNECT_INFO`s has been generated for the original location of the Wnode. These `CONNECT_INFO`s must be discarded, the startNode or endNode moved, intervening corners removed, and the traversal of all RecObjs in the Recognizer must be restarted. There is no mechanism available in the Recognizer to signal the need for such action. This is an ideal case for the addition of exceptions (section 4.3.1).

4.4 Improvements

Improvements to circuit recognition in The Natural Log can be made on many levels. First, smaller improvements are listed. These do not cause any significant change in the control flow and algorithms of the Recognizer, but will make code simpler, or more robust, or improve performance. In section 4.4.2, some proposals which could require a rewrite are discussed. Finally, some unanswered questions remain; any new work on the circuit recognition should start by choosing answers to these questions.

4.4.1 Small Ways

`findSegmentsCrossing`

This method of `CPath`, depended upon greatly, even if indirectly, by the Recognizer to compute locations for connections, should be rewritten. It computes the slopes of the two lines, described by endpoints of two segments, taking into account the special case of infinite slope, then uses these slopes and the locations of the segments to compute the intersection point. Slopes are generally a poor way to carry out this sort of computation.

One alternate method would involve writing an equation with a scale factor as the

unknown, instead of the crossing point. The scale factor represents the distance along one of the line segments from the starting point to the crossing point. A value of 1 is at the ending point of the segment, 0 is at the starting point. Once this scale factor is computed, the crossing point of the two lines can easily be determined from the scale factor and the endpoints of the segment.

Automatically Create Leads

When Elements are redrawn, Wnodes which should always be on the Element are added (section 3.3.6). Some Wnodes of an Element have been added automatically and do not have leads, and some are on the far ends of leads. Connecting to an Element would be much simpler if leads are created at the same time as the Wnodes on the Element; then there would never be a case where the Wnodes created automatically during redrawing are available for connections.

Methods for Finding Object Crossings

Methods should be added to `CWire` and `CElement` which will return a list of locations where the representation of another object crosses the Wire or Element in question. These methods are mentioned in sections 4.1.3 and 4.1.7.

Multiple Connections to One Place for Wnodes

As seen earlier (in sections 4.3.3 and 3.3.4), a Wnode may only have one connection to a given location. This requirement should be implicitly enforced by the rules of drawing a circuit, since any connection of a Wnode implies a Wire between the two locations (zero impedance, in an ideal circuit), so there should not be two different objects in the same place, unless they too are connected.

However, it has proven useful to be able to have more than one connection to the same location. It should not be used in general, but there are cases where it might be used (i.e., on circles, as mentioned in section 4.3.3). It would also make it easier to write fault-tolerant code, because the current implementation is free to either replace a connection or ignore the new one in situations where a connection is added to a location already connected; it would be better if the connections were allows so that information is not lost.

List Classes

Section 3.4.2 lists the subclasses of MFC list classes used by circuit recognition. As discussed, `CMyPtrList` was created to add an assignment operator. However, the assignment operator is rarely used, and declarations are no less cumbersome. Given the relative difficulty in using `CObjList` compared to `CPtrList`,[2] the subclass should have been declared to always use `CPtrList`. This would allow assignment to be added, while simultaneously reducing clutter in the code: `CMyList<CPoint*>` would then be sufficient, replacing `CMyPtrList<CPtrList,CPoint*>`.

4.4.2 Bigger Ways

Use a Ruleset

Discussions of the adjustment methods and `getConnInfo` refer to the need to combine the different methods involved in each, so that they will be easier to understand (section 4.3.2, less in section 4.3.4). In order to allow both of these sets of methods to be flexible enough to accommodate any new rules for connections, the best plan would be to write these methods very generally, and then provide a ruleset for them.

If the rules are to be added without significantly changing the code, they should be resident in a file which The Natural Log can read. A format for the rules needs to be created, and code must be written to apply these rules to the circuit when it is drawn. However, once this is done, a generic `adjustObjToObj` method and a generic version of `getConnInfo` will exist. Written properly, this system would allow additions and changes to the ruleset without requiring that someone learn each detail of the adjustment methods and connection methods first, then edit them, as is the case now. This latter process would likely produce poorly working code.

Another advantage of the use of a ruleset is that the rule format and the code might then be adapted for other uses. Any other drawings using a similar sort of paradigm could work this way, such as an architect's sketch of a building.

[2]The two options provided by MFC for a base class in the template.

Re-Couple Representation and Topology

One of the largest contributions to the difficulties described in this chapter is the fact that early work attempted to decouple the representations of Elements and Wires from the computations determining how they should be connected together. However, too much information about the representation of circuit objects is needed by the connection calculations for the two to be decoupled. Since it is extremely difficult to separate the representation and the topology, attempts to do so interfere with the topology computation. Another effect is that the topology computations are sufficiently insulated from the representation that good decisions about the locations of connections can be hard to make, especially when attempts are made to draw circuits neatly (section 4.1.7). Very little separation is needed for this problem to arise.

At present, Elements and Wires do not have very much useful information about what they are connected to. Wnodes, being the significant part of the data on topology, do not have information about what the Wires or Elements look like. Changing the system to allow Wires and Elements to maintain all the necessary information about connections as well as representation, thereby re-coupling the two sets of information, will make much of the Recognizer's work easier.

De-Localize Connection Decisions

Another significant problem with the circuit recognition in The Natural Log is that all connection decisions are made locally. As seen in section 3.6.2, a connection can be made if the distance between the two locations being connected is small enough, and if other criteria are met, e.g., the connection would not cause line segments to change angle by too much. All connections are based solely on the two objects under consideration for a connection. While modifying the adjustment methods as described in section 4.3.4 would factor in other neighboring objects in certain connection decisions, this is still a local decision.

Circuits could be drawn more cleanly with global information about the locations and connections of circuit elements. A broader view of surrounding elements could restrict some choices of connections. The Natural Log cannot produce a better version of the circuit as envisioned in section 2.1 without considering all parts of the circuit.

If Wires and Elements are changed to contain direct information about what they are

connected to, as discussed in the previous section, the Recognizer will be able to work with data which is less local in character. This will allow connection decisions to be made with more consideration of the surrounding circuit.

Another way in which the overall topology of the circuit can be used is to add means by which the Recognizer can search for a topology which fits certain constraints better than the current topology. This could be done while The Natural Log is idle, waiting for more input from the user. One way in which this could be done is by loosening the connections, and then testing different combinations of connections, possibly using an algorithm similar to simulated annealing. To be able to do this, connections may need to be more reversible, which would cause another change in the way in which Elements, Wires, and Wnodes store connections.

Draw Elements on Wires

One common drawing mechanism used in circuits is to draw an element on top of a wire. Perhaps the person drawing the circuit forgot to add the element earlier, or discovered that a capacitor was needed, or simply waited until later to extend the diagram. Whatever the reason for this behavior, it would be useful for The Natural Log to be able to handle such a change in the circuit.

In order to allow this, the Recognizer must be able to identify Elements which are drawn on top of Wires. However, the definition of "on top" is ambiguous. If the Wire passes through the center of the Element, the user probably intended the Element to be added to the middle of the Wire. If the Wire passes through a corner of the Element, it may be due to sloppy drawing or an intended connection. Metrics for which of these cases is the appropriate interpretation must be determined, and then added to the Recognizer. A simple threshhold may not be enough; which parts of the Element the Wire passes through could be an important metric.

Training

In the beginning of the Natural Log project, when there is a limited set of elements, creating the models and the description of what they look like and how they may be connected can be done by a separate data collection program and careful specification by the programmer. Later, if the system is to be extensible, there must be provisions for a user to enter a new

element and the extra information necessary. One possibility is for The Natural Log to enter a training mode, where the user draws several examples of the element. Then, a guess for what the clean representation is can be displayed. With extra effort, The Natural Log might allow the user to edit this representation. Then, the user can be prompted for which locations should have Wnodes added automatically, and which portions of the Element should be connectable Paths. This, and any additional information collected, can be added to a data file. This may be the global data file as proposed in section 4.1.1, or an additional user-specific file. In this manner, the circuit recognition of The Natural Log can include new types of elements as needed.

4.4.3 Questions to Ask

Here are some questions which those working on future implementations of circuit recognition, or rewrites of the current one, should ask. These are issues which have come up during this project, and good answers have not necessarily been found yet.

Close to Parallel

Should the two lines in figure 4-3 be connected?

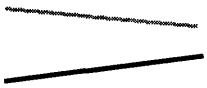

Figure 4-3: It is unclear whether the two lines shown here should be connected, and where.

If the answer is yes, then the circuit recognition must allow for this case. It could be decided that it is always safer to not connect these Wires; the user can always draw another line to connect them.

If these Wires are to be connected, simply increasing the threshhold for the distance from endpoints to the crossing point is not an adequate solution, as this would cause too many connections; the two Wires in figure 4-4 probably will have an Element drawn between them, and connecting them would be undesirable.

Rules for this case will need to be added to the Recognizer, if it is to be supported. When these lines are permitted to connect, it would also be good to make sure that they

Figure 4-4: Increasing the distance threshhold will connect these two Wires, although it is apparent they should not be connected.

are connected with a small line between them, in the manner of figure 4-2.

Wnode Connections

The mechanism by which Wnodes maintain connections to other circuit elements has been the subject of much discussion (especially in section 4.3.3, 3.3.4, and a small amount in section 4.4.1). The way in which these connections should be maintained should be decided upon very early; keeping insufficient information cripples the Recognizer.

The information present in Wnodes was designed to give the Recognizer information about good locations for a Wnode. How can this information be kept without making connections cumbersome? It does not make sense to expect the Wnode to know where the other end of a line is. If an alternate representation of connections is proposed, will it be a noticeable improvement?

Moving Wnodes on Elements

As mentioned in section 4.3.3, Wnodes on Elements should not be allowed to move, except in cases where the representation of the Element changes and the Element itself moves the Wnode. A question to answer is how Wnodes should be changed to enforce this. One solution is to always create leads, as suggested in section 4.4.1. Others are available, including adding a data member to `CWnode`, describing whether the Wnode may move or not. This might even be extended to describe constraints on Wnode movement, or ideal directions of movement, to improve the way in which the circuit is drawn.

Other Ways to Determine Topology

The entire circuit recognition portion of The Natural Log is centered around one mechanism for determining the topology of the circuit being drawn. This mechanism assumes that certain features of the Strokes drawn by the user, most notably end points and sharp changes in direction, betray the intended connection locations. One alternate method might instead look for regions of the circuit enclosed by wires and elements. This would be a good

approach for recognition of images of circuit diagrams, as there is no stroke information present. If other methods of finding connections are developed, several might be employed at once. Agreement between methods would then be a strong case for a given interpretation.

Chapter 5

Conclusion

The implementation of circuit recognition discussed in this document, while not fully complete, is a good start on the problem. At this point, the circuit recognition for The Natural Log can be extended reasonably easily as proposed in Chapter 4 until it is fairly useful. Chapter 3 provides enough information about the workings of the system to introduce a new person to the project. Even if the circuit recognition already implemented is rewritten, the lessons learned in creating the original will provide information on directions to take during the rewrite.

The work presented here also gives advice about some pitfalls to avoid, should a similar system be implemented. The questions posed in the final chapter will be important for continuing work on circuit recognition; a good set of answers to these question will be part of a good final result.