To the Graduate Council:

I am submitting herewith a dissertation written by David R. Brown entitled "The Algorithm Designer Project: A Visual Programming Environment for Data Structure Demonstration." I have examined the final copy of this dissertation for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy, with a major in Computer Science.

Bradley Vander Zanden

Bradley Vander Zanden, Major Professor

We have read this dissertation
and recommend its acceptance:

Accepted for the Council:

Associate Vice Chancellor and
Dean of the Graduate School

The Algorithm Designer Project: A Visual Programming Environment for Data Structure Demonstration

A Dissertation

Presented for the

Doctor of Philosophy Degree

The University of Tennessee, Knoxville

David R. Brown

May, 1999

Acknowledgments

I want to thank Dr. Brad Vander Zanden for his patience and support throughout the course of my dissertation. I would also like to thank Dr. Michael Berry, Dr. Bruce MacLennan, and Dr. John Ray for their support and service on my committee.

I sincerely wish to thank my wife Carol for her support and dedication. Her ceaseless efforts on my behalf allowed me to complete this work. I would also like to thank Dr. Allen Smith who inspired me to begin the Ph.D. program and for the journeys we've shared. Thanks to Rick Phillips for going back to school with me and for always coming through when I've needed help. Thanks also to Ron Parr for his support as my supervisor and for all of the instruction that he has given me. Thanks to Nancy Getsi for helping me to get through the Qualifying Exam and for her continuing friendship.

I would also like to thank my brother Danny for taking care of things over the past few years and for understanding me. Thanks to my mother Barbara, my father Aubrey, and my step-mother Judy, for their understanding and support during this time.

This work is dedicated to memories of my grandmother June Brown, my grandfather Paul Maxfield, and my grandfather Hayes Brown.

Abstract

Previous work on pedagogical tools for teaching students algorithms has focused on high-level animations of the algorithms. This dissertation describes a tool that gives instructors the ability to pictorially demonstrate the implementation of algorithms at the data structure level.

The Algorithm Designer Project explores the use of a computer as an electronic whiteboard for instruction of computer science. It improves upon the traditional physical blackboard environment by providing syntactic and semantic support for data structure design and algorithm demonstration. The ultimate goal of this project is to provide an attractive, easy to use, system through which users can demonstrate simple algorithms and data structures, such as those presented in data structures textbooks. The project consists of three components: Data Structure Designer, Algorithm Designer, and Rule Designer. Data Structure Designer allows users to design and customize the appearance of data structures that they intend to use to create visual programs. Concrete examples of these data structures can be placed into Algorithm Designer and directly manipulated to demonstrate algorithms. Visual programs are programs written using pictures instead of, or in conjunction with, text. Rule Designer allows the creation and manipulation of transition rules to define visual program scripts to act upon Algorithm Designer objects. The project was implemented using the Amulet toolkit and runs on Macintosh, Windows, and UNIX platforms.

A key insight discovered during development of the Algorithm Designer Project was that although textbooks employ a wide variety of data structure visualizations, the differences between these visualizations can be grouped into a small number of categories. Two unique interface items were developed during the course of the research: 1) a color mapping widget interface item that provides an easy way for the user to associate a set of colors with a range of values in a data structure visualization and 2) "seeds" and "holes," a mechanism for visually identifying and supporting type-specific semantic behavior for edge-based data structures. Finally, this dissertation describes a novel use of imperative programming constructs within a pictorial rewrite rule-based scripting system and a novel use of these rules for teaching conventional imperative programming.

Contents

1. INTRODUCTION 1

2. DATA STRUCTURE DESIGNER 7

2.1 INTRODUCTION 7
2.2 OVERVIEW 10
2.3 PROPERTY AREA 10
2.4 VISUALIZATION AREA 12
2.5 PROPERTY MAPPING AREA 16
2.6 EXAMPLE VISUALIZATION AREA 20
2.7 CONCLUSIONS 21

3. ALGORITHM DESIGNER 22

3.1 INTRODUCTION 23
3.2 OVERVIEW 24
3.3 DATA STRUCTURE CREATION 25
3.4 CREATING EXAMPLE DATA 26
3.5 SYNTACTIC AND SEMANTIC SUPPORT 26
3.6 ANNOTATION 28
3.7 ALGORITHM DESIGNER EXAMPLE 28
3.8 CONCLUSIONS 32

4. RULE DESIGNER 33

4.1 INTRODUCTION 33
4.2 OVERVIEW 39
4.3 USER INTERFACE 41
4.4 ALGORITHMS 50
4.4.1 PATTERN MATCHER 50
4.4.2 SCRIPT INTERPRETER 54
CREATE STATEMENTS 55
DELETE STATEMENTS 55
ASSIGNMENT STATEMENTS 55

VISUAL PROPERTY MODIFICATION STATEMENTS 56
OBJECT MOVEMENT STATEMENTS 56
SWAP STATEMENT 57
INPUT STATEMENTS 57
OUTPUT STATEMENTS 57
ARITHMETIC EXPRESSIONS IN VALUES 58
CONDITIONAL EXPRESSIONS IN PATTERNS 58
4.5 RULE DESIGNER LIMITATIONS 59
4.6 CONCLUSIONS 60

5. RELATED WORK 61

5.1 OVERVIEW 61
5.2 DRAWING TOOLS 61
5.3 PEDAGOGICAL TOOLS 63
5.4 DATA STRUCTURE VISUALIZATION 65
5.5 VISUAL PROGRAMMING 66
SPECIFICATION-BASED LANGUAGE SYSTEMS 66
PROGRAMMING-BY-EXAMPLE SYSTEMS 67
PICTORIAL REWRITE RULE SYSTEMS 69
TERM OR GRAPH REWRITE RULE SYSTEMS 70

6. EXPERIENCE 71

7. CONCLUSIONS AND FUTURE WORK 74

7.1 CONCLUSIONS 74
7.2 SUMMARY OF CONTRIBUTIONS TO COMPUTER SCIENCE 77
7.3 FUTURE WORK 77

BIBLIOGRAPHY 78

VITA 84

List of Figures

Figure 1 - Search Algorithm Initial State 2
Figure 2 - Search Algorithm Final State 3
Figure 3 - Data Structure Designer Window 11
Figure 4 - Object Properties 12
Figure 5 - Linked List Object Creation 13
Figure 6 - Data Structure Styles 14
Figure 7 - Property Location Modification 16
Figure 8 - Numeric Color Mapping Widget 19
Figure 9 - Boolean Color Mapping Widget 19
Figure 10 - Defining a Sub-range 19
Figure 11 - Algorithm Designer 24
Figure 12 - Prepending to a Linked List 29
Figure 13 - List Traversal Using Pointers 30
Figure 14 - Inserting into a List 31
Figure 15 - Currency Pointer Initialization 35
Figure 16 - Begin Repeat Block 35
Figure 17 - Search Value Found 36
Figure 18 - End of List Condition 37
Figure 19 - Search Value Not in List 37
Figure 20 - Advancing List Currency 38
Figure 21 - End of Repeat Block 38
Figure 22 - Algorithm Designer and Rule Designer Environment 40
Figure 23 - Programming Buttons 41
Figure 24 - Rule Designer Execution Strip 42
Figure 25 - Use of Variables 44
Figure 26 - Keyword Strips 45
Figure 27 - Use of X-mark Object 47
Figure 28 - Tag Options 47
Figure 29 - Condition Box 49
Figure 30 - Visual Script Input 49
Figure 31 - Visual Script Output 50
Figure 32 - High-level Pattern Matching Algorithm 52

Chapter 1

Introduction

Most instructors describe algorithms to students using a blackboard or a whiteboard and chalk or markers. To do this, the instructor typically draws pictures of the appropriate data structures, fills in the data structures with example data, and then manipulates this data according to the rules of the algorithm. The primary use of Rule Designer is to create scripts that aid in describing algorithms and demonstrating the use of various data structures. Additionally, Rule Designer can be used by students to replay a demonstration at a later time in order to examine the workings of the algorithm in more detail.

A physical whiteboard is obviously a simple, low cost, and effective medium for instruction. When used to describe a data structure or an algorithm, an instructor is free to draw any type of shape and to provide annotations anywhere they feel would aid in student comprehension of the material. Color can even be used to emphasize various points about a data structure or algorithm description.

For example, assume that an instructor is attempting to illustrate a sorted linked list search algorithm. They might begin by drawing the initial program state shown below in Figure 1.

The instructor would then talk through the algorithm, showing how the "current" pointer is moved progressively through the nodes of the linked list. At each step in the algorithm, there is a set of patterns that we are looking for. Each pattern requires a different response. For example, if the "current" pointer is pointing to a node whose value is less than the value that we are searching for, then the current pointer is moved to the next node in the list (if we are not at the end of the list).

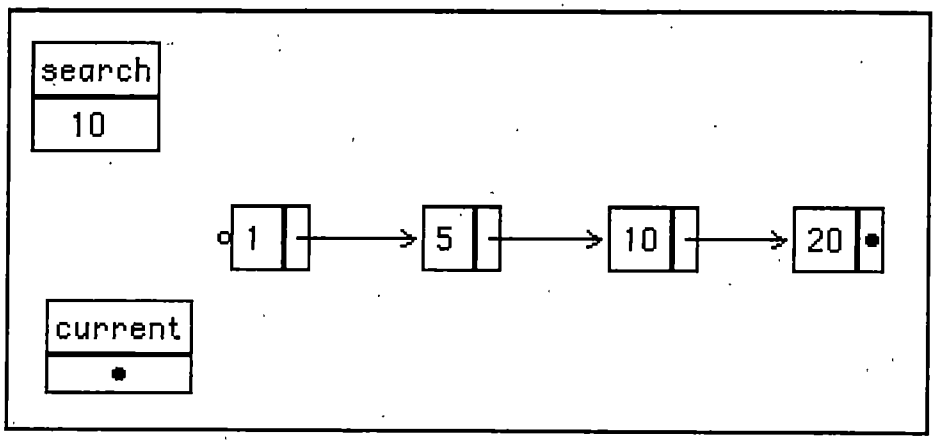


Figure 1 - Search Algorithm Initial State

The instructor may draw side diagrams to illustrate the various patterns and how they would be handled. At each point in the discussion, the instructor is required to erase and redraw objects to throw light upon the important points in the discussion. Eventually the program state shown in Figure 2 is reached and the algorithm ends with the searched for value being located in the list.

The drawback to this low-tech approach is primarily its generality. It can be used to describe anything, but it provides no support for specific types of demonstrations. Demonstrating an algorithm on a physical whiteboard can be quite tedious as each object that is used in the algorithm must be drawn, and then redrawn whenever it is moved.

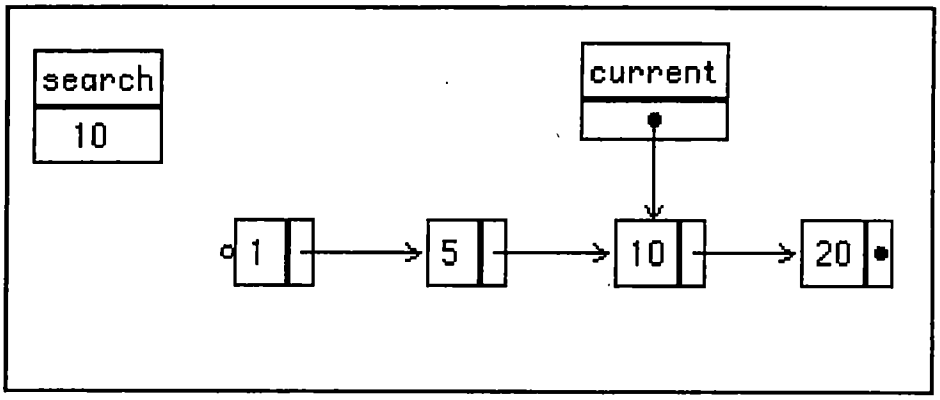


Figure 2 - Search Algorithm Final State

Finally, there is no interactivity or support of any kind for data structure object behavior or algorithm demonstration.

Computers can provide the interactivity and dynamic behavior that is lacking in a physical whiteboard. In addition, the advent of relatively inexpensive projection equipment for computers makes it feasible to economically project a computer's display in a classroom.

There has been previous work on pedagogical tools for teaching students algorithms, but these tools have focused on high-level animations of the algorithms rather than on the data structures required to implement the algorithms. Hence, previous systems have tried to give students a general understanding of the algorithm. In contrast, this dissertation describes a tool that gives instructors the ability to pictorially demonstrate the implementation of algorithms at the data structure level.

There has also been research aimed at providing supplemental computer aided instruction, some using simulation environments. This research, however, is aimed at enhancing the primary learning prop used in most classrooms, namely the blackboard itself. In other

words, this is an attempt to reinvent the most fundamental teaching aid used in the classroom.

This dissertation describes the Algorithm Designer Project. The ultimate goal of this project is to develop the technology required for electronic instruction of computer science. It provides an attractive, easy to use, system through which users can demonstrate simple algorithms and data structures, such as those presented in data structures classes. The visual programs are intended to look as much like the examples provided in data structures textbooks as possible. The system also allows instructors to save examples in data files which can be distributed to students before or after class for their review. It is the first attempt that the author is aware of, to attempt to replace the classroom blackboard with an intelligent, electronic whiteboard.

The Algorithm Designer Project consists of three components: Data Structure Designer, Algorithm Designer, and Rule Designer. Data Structure Designer allows users to design and customize the appearance of data structures that they intend to use to create visual programs. Concrete examples of these data structures (i.e., objects with example values) can be placed into Algorithm Designer and directly manipulated to demonstrate algorithms or to visualize a program state. Rule Designer then allows the creation and manipulation of transition rules to define visual program scripts to act upon Algorithm Designer objects. The Algorithm Designer Project was implemented using the Amulet toolkit [Amu95] and runs on Macintosh, Windows, and UNIX platforms.

Data Structure Designer is a direct manipulation, drag-and-drop graphical editor, similar to a widget-editor that allows users to create attractive, customized data structure objects that very closely resemble the pictures of data structures that appear in textbooks. Data Structure Designer supports arrays, graphs, linked lists, queues, and trees in a variety of styles. Most visual characteristics of the objects can be easily manipulated. A data

structure's properties (i.e., its fields) can be declared and then displayed by associating properties with locations around or within an object visualization. Properties can also be "mapped" such that a given range of property values correspond to a specific fill color or line color for the node or edge of an object. Lastly, example values can be assigned to properties so that the user can see how an actual, instantiated example of the data structure object will look as it is being created.

Algorithm Designer provides a drawing area where data structures created in Data Structure Designer can be interactively drawn, populated with example data, and then used in the demonstration of algorithms. Algorithm Designer provides built-in syntactic and semantic support for data structure design and algorithm demonstration. For example, assignment is accomplished by dragging and dropping a variable onto another variable.

Rule Designer provides a simple visual script creation and execution engine for Algorithm Designer. Using cartoon strips, an instructor can create a series of "pictorial rewrite rules" that define actions to be taken upon Algorithm Designer objects. Pictorial rewrite rules describe a pattern that is matched against objects and the actions that are to be taken upon the instantiated Algorithm Designer objects. Visual strip constructs are also provided to support the creation of loops and conditional execution blocks. The visual program resulting from the set of cartoon strips can then be used to enhance the presentation of algorithm descriptions and aid in data structure design discussions.

While other systems have made use of pictorial rewrite rules, this dissertation describes the first one that the author is aware of, to integrate pictorial rewrite rules with imperative flow-of-control constructs. Typically pictorial rewrite rules are a series of ordered "if condition then action" rules. The rewrite rules are executed by starting with the first rule in an ordered list and continuing until a condition is satisfied. The action is then executed and the search is repeated, starting with the first rule. In contrast, the system described in this dissertation

organizes the execution of pictorial rewrite rules using case statements and loops. The pictorial rewrite rules are like statements in an ordinary program. Pictorial rewrite rules are executed sequentially until a case statement or loop is encountered. During sequential execution, a pictorial rewrite rule's action is executed if its condition is true. Regardless of whether the condition is true or false, control passes to the next pictorial rewrite rule. A case statement selects which rules to execute based on conditional statements. A loop repetitively executes rules until some condition causes control to break out of the loop. Hence a program is a series of pictorial rewrite rules governed by flow-of-control constructs.

The remainder of this dissertation is organized as follows. Chapter 2 describes the Data Structure Designer, which allows data structure creation and customization. These objects may be used in Algorithm Designer, which is described in the Chapter 3. Algorithm Designer provides a whiteboard environment for data structure design and algorithm demonstration. Chapter 4 describes Rule Designer, the pictorial rewrite rule visual scripting environment. Chapter 5 describes previous work related to this research. Experienced gained through informal testing is described in Chapter 6. Conclusions and ideas for future work are summarized in Chapter 7.

Note that the Related Work chapter appears toward the end of the dissertation. Since this dissertation describes a system that spans several different areas of research, the discussion of related work is more meaningful after an in-depth discussion of what was accomplished during the research.

Chapter 2

Data Structure Designer

2.1 Introduction

Traditionally data visualization research has been oriented toward generating attractive data structure visualizations from user code for data examination and debugging purposes. This research attempts to do just the opposite. It is intended to allow users to create attractive data structure visualizations, which are then used to create a visual computer program.

As stated previously, Data Structure Designer is one of three components of the Algorithm Designer Project, the other two components being Algorithm Designer, and Rule Designer. Data Structure Designer allows users to design and customize the appearance of data structures that they intend to use to create visual programs. Concrete examples of these data structures (i.e., objects with example values) can be placed into Algorithm Designer and directly manipulated to demonstrate algorithms or to visualize a program state. The Rule Designer allows the creation and manipulation of transition rules to define visual program statements and program components.

One of the chief obstacles faced in Designer was the wide variety of data structure visualizations that textbooks employ. For example, linked lists can be shown with a pointer coming out of a pointer box, or simply with a pointer extending from the list node. Structures are sometimes shown vertically, and other times horizontally. Color and other visual characteristics are often used to emphasize certain data structure elements or just to make the picture more attractive. Different shapes, such as rectangles or ovals, may be used to represent the data elements. Finally the location of the values associated with a data structure vary widely. Values can appear within, above, below, or beside data structure elements.

The goal of Designer was to allow a user to quickly duplicate this wide variety of styles. In particular, it was felt that users should not have to create data structure visualizations from scratch. After examining a number of data structures textbooks (e.g., [AT87, Sed90]), it became clear that although there are many different visualizations of data structures, these differences can be grouped into a small number of categories:

1. Style

   Most data structures have a few stereotypic styles. For example, lists typically have pointers coming out of the ends of the data element. The pointers may optionally be set off with pointer boxes. Similarly, tree nodes typically have pointers coming out the bottom or coming out the sides.

2. Shape

   Most data elements are represented as either rectangles, circles, or ovals.

3. Orientation

   Most data structures are oriented either vertically or horizontally.

4. Partitioning

   Most data elements display a small number of fields by partitioning the data element or by placing the fields around the data element.

5. Color

   Color is sometimes used to call attention to particular data elements. Specific colors may also appear based upon the value of a data structure field.

Data Structure Designer allows users to customize data structures by selecting attributes from each of these categories using a direct manipulation, drag-and-drop, graphical user interface. Users select a data structure type and are provided a default template by the system. Users then customize the template via palettes and through a color mapping widget that allows ranges of values to be mapped to specific colors.

As the template is being customized, the user can provide example values for the fields of a sample data element. The data structure editor instantiates the template with these values to provide a constantly updated example visualization of how the data structure will appear with actual data.

Currently, the Data Structure Designer supports five types of data structures: arrays, lists, queues, trees, and graphs. As stated previously, it is written using the Amulet toolkit and runs on Macintosh, Windows, and UNIX platforms.

The remainder of this chapter provides an overview of Data Structure Designer, followed by an in-depth description of Designer components and features, and finally, presents conclusions based upon the Data Structure Designer research.

2.2 Overview

The Data Structure Designer consists of four parts (see Figure 3):

1. Visualization Area

   This area is used to define a customized, pictorial data structure.

2. Property Area

   This area is used to specify the names of the data structure's fields, their types, and their example values.

3. Example Area

   This area combines the specification from the visualization area with the example values from the property area to show what an instance of the data structure will look like in the Algorithm Designer.

4. Mapping Area

   This area displays and allows modification of the color mapping information associated with a particular data structure property.

Figure 3 shows the completed design of a linked list data structure. The next four sections will describe the various parts of Data Structure Designer and will use the linked list data structure as a continuing illustration of how a user creates a visualization.

2.3 Property Area

The property area allows the user to define the properties (fields) of a data structure. Since trees, lists, and graphs typically consist of two types of objects, a data element and an edge or pointer, the property area allows properties to be defined for either a data element (called a node) or an edge. To conserve screen space, only the node properties or the edge properties are displayed at any given time. A title above the scrolling area indicates which

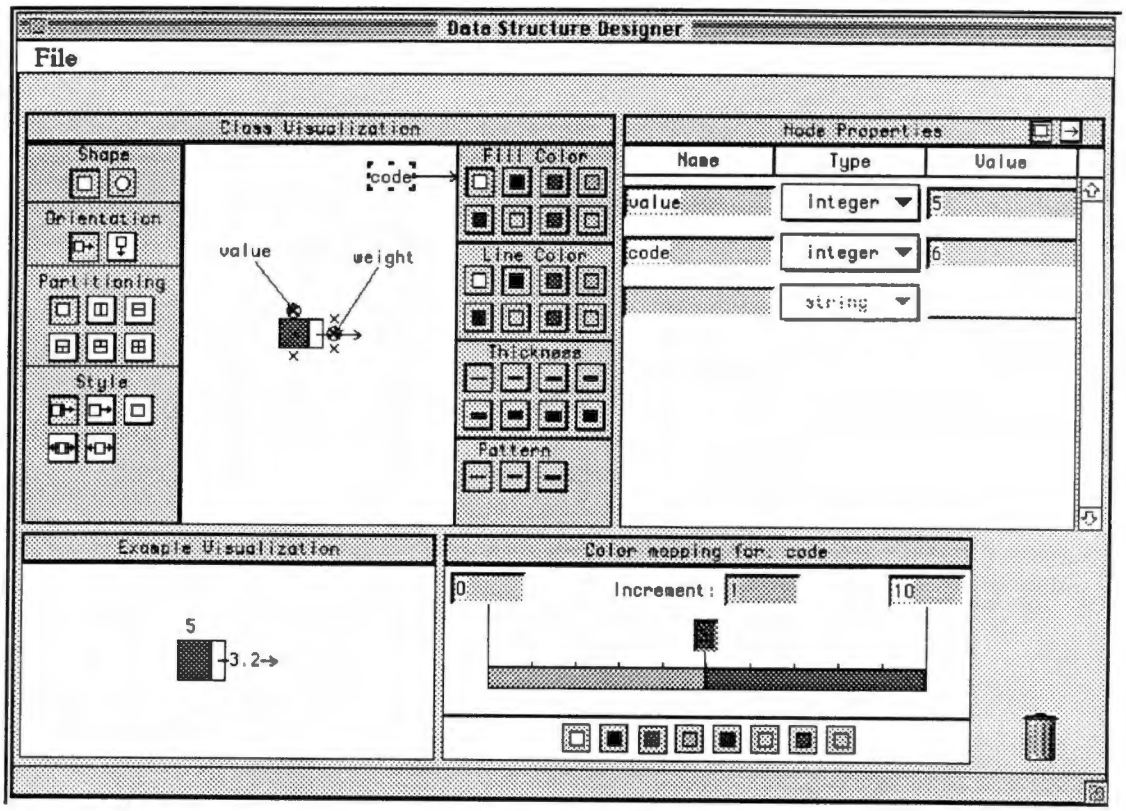


Figure 3 - Data Structure Designer Window

properties are currently displayed. The two buttons at the upper right of the property area (see Figure 4) control which set of properties is currently displayed.

A property consists of a name, a type, and an optional example value. The example value is used in creating the example object visualization that appears in the example area. The example value also serves as the default value for the field when instances of the data structure are created in the Algorithm Designer.

When an object is first created, an empty property list is created for it. A blank property item is used to allow new properties to be created. The user creates a new property by typing a property name on the blank line. Figure 4 shows the result of adding a *value* property to the list of node properties for the current data structure. A new blank line

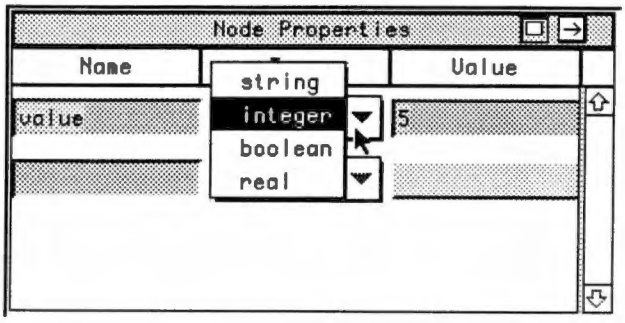


Figure 4 - Object Properties

appears after the *value* line, thus allowing another property to be created. The user can delete a property by dragging the property into the trash can located at the lower right-hand corner of the Data Structure Designer window.

The user can change the name and value fields of existing properties by simply typing new values into the fields. The user can change the property's type by selecting the desired type from a pop-up menu. The data structure designer currently supports four atomic types: string, integer, boolean, and real.

The values of properties can be displayed visually using the techniques described in the next two sections.

2.4 Visualization Area

The visualization area is used to create a graphical picture of a data structure. Figure 5 shows a close-up of this area. In the figure, the user is creating a visualization of a linked list element.

The user begins an editing session by either opening an existing object definition, or by creating a new one by selecting the desired data structure from the File menu. The Data

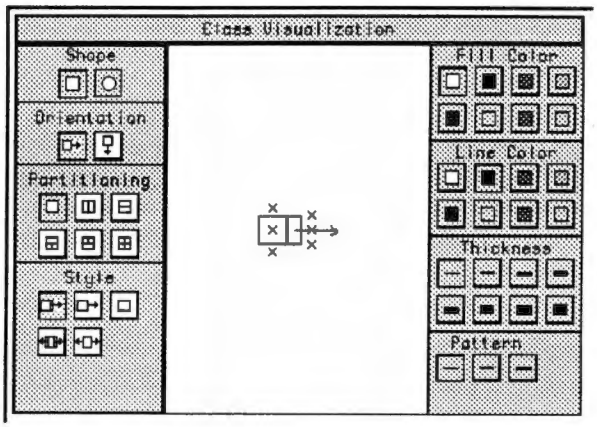


Figure 5 - Linked List Object Creation

Structure Designer places a default image of the data structure in the drawing window. This default image can be altered by selecting options from one of the four palettes on the left side of the drawing window:

1. Shape Palette

   The shape palette controls the shape of the data structure node. Currently rectangles and ovals are supported.

2. Orientation Palette

   The orientation palette controls whether the elements of the data structure are to be oriented horizontally or vertically.

3. Partitioning Palette

   The partitioning palette controls the number of properties of the data structure element that may be displayed within the node at any given time. Anywhere from one to four properties may be simultaneously displayed. The icons on the palette buttons show the positional formats that are available.

4. Style Palette

The style palette controls the overall appearance of a data structure. It is the only palette which changes depending on the type of data structure selected. In the case of a linked list, there are five possible styles, two representing a singly linked list, two representing a doubly linked list, and one representing either a singly or doubly linked list in which the pointers are not displayed. Pointers can be displayed either with or without a pointer box. Figure 6 shows the styles selections available for the various data structures.

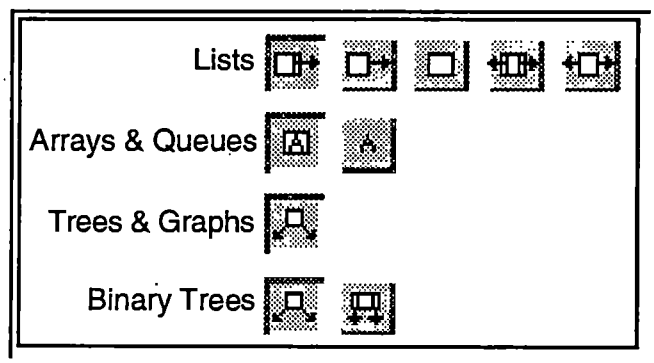


Figure 6 - Data Structure Styles

By selecting options from these four palettes, the user can quickly customize the appearance of the desired data structure. Data structures can be further customized using the color and line palettes on the right side of the drawing window. The portions of the data element that may be customized are: 1) the object representing the data element, and 2) any edge emanating from the data element. We experimented with allowing individual fields of a data element to be colored differently (i.e., an object could have up to four different colors if it had four different displayed fields) but decided in this case to simplify the interface, rather than support what would most likely be a little used feature. Consequently, the color of a data element is determined by its fill color and its line color. An edge may be

colored differently from the node, however. Colors may be chosen so that a color is always displayed, or the color may be determined by the value of a field using the mapping widget described in a later section.

The x's within and surrounding the data structure element are positions where object properties may be attached to visualize the field's value. The partitioning palette controls the number of properties (denoted by x's) that may appear within the node itself. In addition, two other properties may be placed around the node. The positioning of x's around the object is controlled by the orientation of the object. For example, in Figure 3 (p. 11), the x's appear on the top and bottom of the horizontally oriented list node rather than on the left and right. This is because a pointer arrow will normally be coming into the left side of the node, and emanating from the right side, and hence would intersect the displayed values. If the user changes the list element's orientation to vertical, the x's around the object are repositioned so that they appear horizontally rather than vertically.

In addition to displaying the properties of a data element, the properties of an edge or pointer may also be displayed. The three x's shown in a vertical line over the pointer in Figure 3 (p. 11) represent the three positions at which pointer properties may be visualized.

The fields of a data structure can be visualized by dragging a property name from the property area and dropping it on one of the x's inside or around a node or edge in the visualization area. The editor ensures that fields can only be dropped on the appropriate x's (e.g., a field of the data element cannot be dropped on an x belonging to an edge). Figure 3 (p. 11) shows the result of dropping the value field on the x above the list node. Notice that the example visualization area (at the lower left of the window) shows how the field will appear when the list object is ultimately used in the Algorithm Designer. The example value from the property area is used to provide a concrete value for the field.

A property's location can be easily modified by simply dragging and dropping the circle that appears at the end of the association line onto the desired location as shown in Figure 7. Property associations can also be easily deleted by dragging the association title into the trash can at the lower right-hand corner of the window. If the property itself is deleted, all of its associations are deleted as well. For example, if the *value* property shown in Figure 3 (p. 11) was dragged into the trash can, the *value* association shown in the visualization area would also be removed.

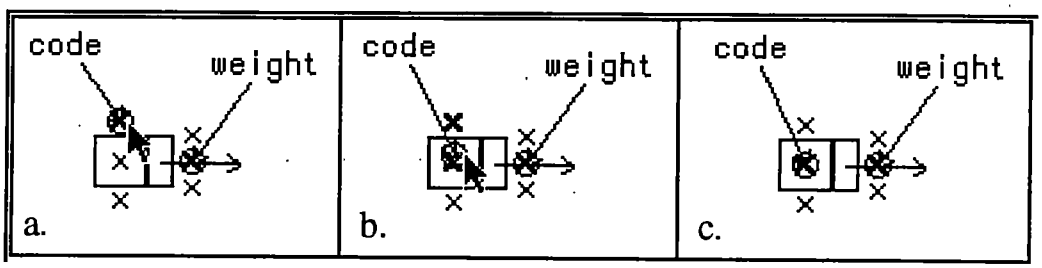


Figure 7 - Property Location Modification

A field's value can also be visualized via the fill color or line color of a data element or edge. The next section describes how such color mappings can be established.

2.5 Property Mapping Area

A property mapping is a way to specify that a range of values for a property should correspond to specific fill colors and line colors in the object visualization. When a mapped property is set to a value within its mapped value range, a corresponding change takes place in the fill or line color. For example, assume a property is mapped to the node fill color as follows: [1, 10] = red, [11, 19] = green, [20, 30] = blue. If the *value* property is set to 15, the fill color of the node will be set to blue.

A property mapping is created by dragging a property name from the property area and dropping it onto either the fill or line color button palette. The mapping constrains visual features of either the node or its edges depending on whether a node or edge property is used to establish the mapping. Figure 3 (p. 11) shows the code property mapped to the node's fill color. When a mapping is created, an association is established in the visualization area showing the field name and an arrow pointing to the mapped fill or line color palette. The mapped field name is selected and a default color mapping is created and displayed using a color mapping widget in the property mapping area located at the lower right of the Data Structure Designer window. Property mappings are deleted by simply dragging the property mapping name from the visualization area into the trash can.

The color mapping widget provides an easy way for the user to associate a set of colors with a range of values. There are two types of color mapping widgets: boolean and numeric. The type of widget displayed in the mapping area depends upon the type of the mapped property. The color mapping widget is made up of the following parts:

1. Start Value

   The start value specifies the first value of the range (numeric widget only).

2. End Value

   The end value specifies the last value of the range (numeric widget only).

3. Increment

   This value specifies the increments in which sub-ranges may be defined (numeric widget only).

4. Color range strip

   The color strip is a rectangular region that is divided into color strips. Each color strip denotes a sub-range of values that should be mapped to the color of the strip.

Color strips are divided by tabs that display the boundary value of the sub-ranges. The color of the tab denotes which sub-range includes the boundary value (i.e., the boundary value is owned by whichever color strip matches the tab's color). Figure 8 denotes the color mapping: [0, 5] = green, (5, 10] = red. The boolean widget works in a similar fashion to the numeric widget except that the color strip is divided into two fixed regions, representing the values true and false. The boolean widget in Figure 9 denotes the mapping: true = red, false = blue.

5. Color chips

   The color chips appear on a button palette from which colors for ranges are selected.

New sub-ranges can be defined by dragging and dropping a color chip onto the color range strip as shown in Figure 10. The placement of the inserted sub-range depends upon the mouse location at the time of the drop. If the color chip is dropped on the first half of an existing color strip, the new sub-range is inserted before the color strip, and if dropped on the second half, it is inserted after the color strip. As the chip is being positioned, an arrow appears showing where the new sub-range would be created if the chip were dropped at that location. In either case, the insertion is accomplished by dividing the range of the original color strip in half and rounding to the nearest increment. One half of the strip is assigned to the new color and the other half is assigned to the original color. The new color range can then be adjusted by either entering a new value into the boundary tab which divides the new and old color strips, or by dragging the tab until it reaches the appropriate value.

Colors for existing ranges are changed by selecting the associated color strip and then clicking on the desired color chip. Color ranges are deleted by dragging them into the trash can. A neighboring color range is extended to cover the range vacated by the discarded color strip.

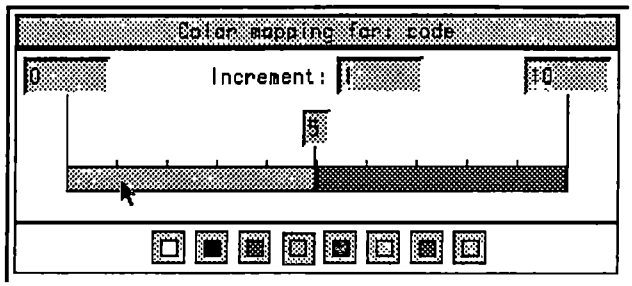


Figure 8 - Numeric Color Mapping Widget

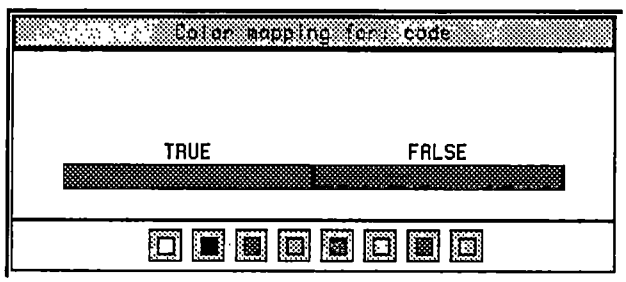


Figure 9 - Boolean Color Mapping Widget

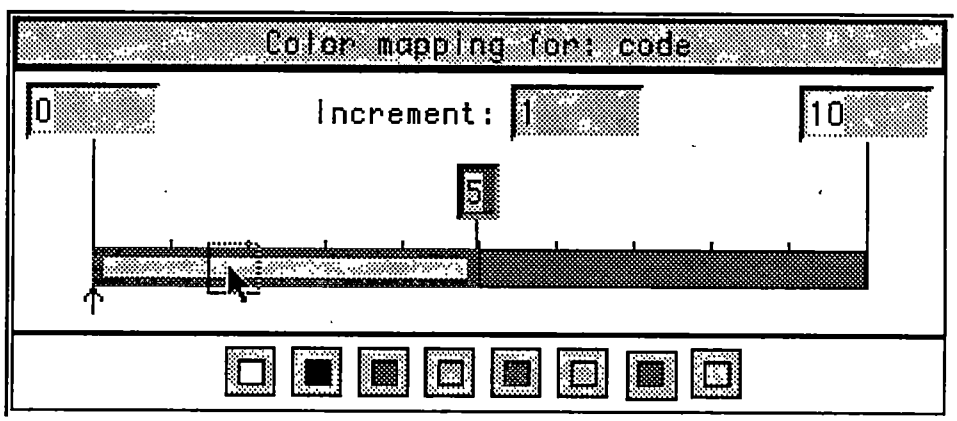


Figure 10 - Defining a Sub-range

The next section describes the Example Visualization Area which ties together the various pieces of a data structure visualization.

2.6 Example Visualization Area

The Example Visualization Area contains an example showing how the data structure object being defined will appear when used in Algorithm Designer. The Example Visualization Area is display-only. All of the interaction that takes place to produce the example visualization is done in the visualization area, property area, and property mapping area. The example visualization is the synthesis of the following components:

1. The data structure definition from the Visualization Area.
2. The example values from the Property Area.
3. The color mappings from the Mapping Area.

In the example shown in Figure 3 (p. 11), the example value of the *value* property is displayed above the linked list node. Notice that, within the visualization area, the *value* property has been associated with the location above the node and the example value of the *value* property has been set equal to 5 in the node properties area. Likewise the weight edge property *value* of 3.2 is displayed centered on the forward-pointing edge of the linked list node.

Also note in Figure 3 (p. 11) that the fill color of the node has been mapped to the *code* property. The color mapping area shows the color mapping: [0, 5] = green, (5, 10] = red. Since the value of the *code* property is currently 6, the fill color of the example object is red.

2.7 Conclusions

Data Structure Designer allows users to quickly create pictures of data structures that resemble those used in textbooks. A key insight found during development of Data Structure Designer was that although textbooks employ a wide variety of data structure visualizations, the differences between these visualizations can be grouped into a small number of categories. These categories can be manipulated using palettes in the Data Structure Designer. Some of the other features of the Data Structure Designer that make it easy to create pictorial data structures include: 1) a drag-and-drop, graphical interface, 2) a color mapping widget that allows the values of fields to be visualized using colors, 3) a drag-and-drop technique for allowing the values of fields to be associated with various locations in a data structure, and 4) a constantly updated example visualization of how the data structure will appear with actual data.

The next chapter describes Algorithm Designer, an electronic whiteboard environment where the data structure objects created by Data Structure Designer can be used to create data structures and to demonstrate algorithms.

Chapter 3

Algorithm Designer

Most instructors describe algorithms and data structures to students using a physical blackboard or a whiteboard and chalk or markers. This chapter describes Algorithm Designer which provides an instructor with the functionality of a whiteboard. This environment improves upon a whiteboard by providing interactivity and dynamic capabilities that whiteboards lack. The Algorithm Designer screen can be projected onto a classroom wall, thus providing a substitute for a whiteboard.

Algorithm Designer provides a drawing area where data structures created in the Data Structure Designer can be interactively drawn, populated with example data, and then used in the demonstration of algorithms. Built-in semantics facilitate common operations. For example, assignment is accomplished by dragging and dropping a variable onto another variable.

The remainder of this chapter describes the Algorithm Designer in detail.

3.1 Introduction

To describe an algorithm, an instructor typically draws pictures upon a physical whiteboard of the appropriate data structures, fills the data structures with example data, and then manipulates this data according to the rules of the algorithm.

A physical whiteboard has a number of features that are helpful to the instructor in this regard:

1. It allows an instructor to draw any type of shape.
2. It allows an instructor to provide annotations wherever desired. For example an instructor can draw arrows to denote the movement of data or write a small piece of code, such as "x < y?", to illustrate a condition or operation.
3. It allows an instructor to use color to emphasize various points in a data structure or algorithm description. For example, an instructor might assign different colors to the visited and unvisited nodes in the depth-first search of a graph.

A physical whiteboard also has its limitations, however:

1. It provides no built-in objects, each must be drawn from scratch.
2. It does not allow an instructor to duplicate an object. If an instructor wants to duplicate an object, it must be drawn again from scratch.
3. It requires the instructor to manually erase and then redraw parts of an object if an operation moves the object, changes the color of the object, or changes any other visual characteristics of the object.
4. It does not allow instructor to easily highlight parts of the data structure (e.g., highlight two array elements that are about to be swapped).

In short, whiteboards provide a good low-end solution due to their availability and low cost, but they do not provide any interactive or dynamic capabilities. Algorithm Designer takes advantage of the interactive capabilities of a computer to remedy these shortcomings.

3.2 Overview

Figure 11 shows the Algorithm Designer window, which looks much like a drawing editor. On the left side are object palettes for creating primitive data types, data structures, and custom data structures. In the middle of the screen is a drawing area. On the right side are palettes for changing the color or line style of either a line or an object. Across the top of the editor are a variety of pull-down menus that provide common drawing editor functionality.

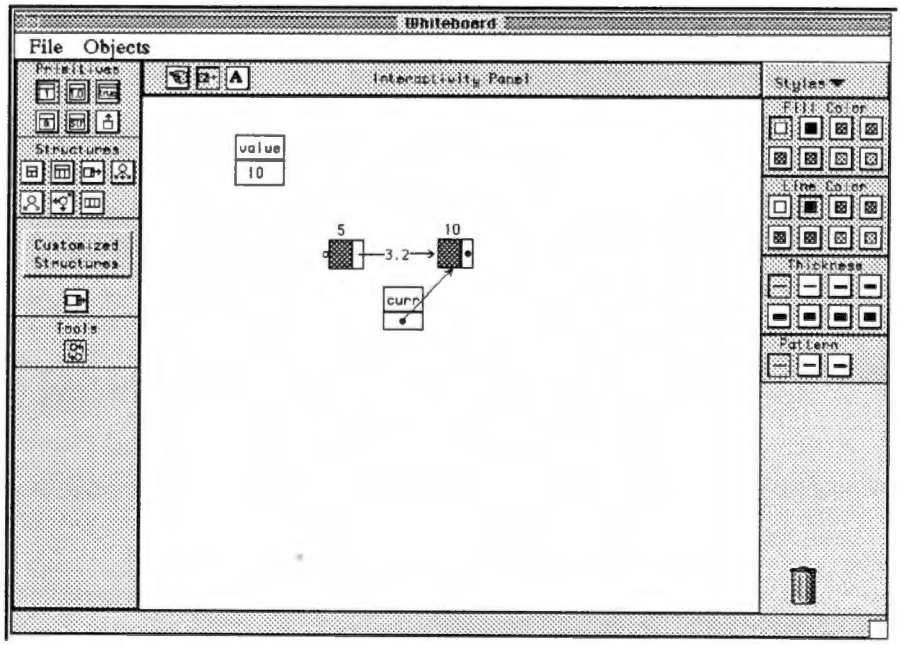


Figure 11 - Algorithm Designer

Algorithm Designer focuses upon direct manipulation and directness [Shn83]. This allows people to quickly learn how to use, and feel comfortable with, the system. Objects are created, selected, moved, and deleted just as they would be in an ordinary drawing editor. Additionally, if a user can see a property, such as the name of a variable or the value of a variable, then the user can edit that property directly. In the case of a name or a primitive type, the value is changed by editing a text string. In the case of a pointer, the value is changed by grasping the pointer's arrow and dragging it to a new object. In the case of a color, the appropriate object is selected and the color is changed by selecting the desired color from a color palette.

The remainder of this chapter describes several of the other features of the Algorithm Designer environment that facilitate algorithm demonstration. These features include the ability to easily create data structures, populate them with example data, provide syntactic and semantic support for data structure operations, and supply annotations for the resulting visualizations.

3.3 Data Structure Creation

Algorithm Designer allows users to easily create variables with common primitive types (integers, strings, booleans, characters, real) and common data structures (records, lists, arrays, graphs, queues, trees, binary trees). Variables and data structures are created by selecting the appropriate object from a data object palette, and then clicking on the Algorithm Designer to create the selected object at the current mouse position. Customized objects can be imported from the Data Structure Designer by pressing the "Customized Structures" button and entering the name of the desired data structure file. The Algorithm Designer environment creates an icon for the new data structure and adds the icon to the object palette (Figure 11). The icon picture is automatically created based on the data

structure's properties, such as its type, its orientation (vertical or horizontal), and its partitions.

3.4 Creating Example Data

The data structures and variables that are placed on the screen can be quickly populated with example data by either 1) directly editing the individual fields in a data structure, or 2) pulling up a property sheet for a data structure and entering values there. The property sheet also allows a list of values to be entered for arrays. These values are mapped to the array elements, with the array size being automatically adjusted to accommodate the number of values in the list.

If a custom data structure from the Data Structure Designer is used, then the display of the data elements will be computed by applying the property mappings established in Data Structure Designer to the example data.

3.5 Syntactic and Semantic Support

The Algorithm Designer environment provides support for a number of common operations, such as assignment, manipulating pointers, "wiring" up edge-based data structures, such as lists, trees, and graphs, and swapping the elements of an array.

Assignment. Constant values can be assigned to a variable or to a field of a data element by directly editing the appropriate value. A value from one object can be assigned to another object by dragging and dropping the appropriate field of the source object onto the appropriate field of the target object. Depending on the sequence of mouse keys and auxiliary keys used, the contents of an entire object, or simply one field in the object can be assigned to another object.

A user can also assign one pointer to another pointer using this drag-and-drop mechanism. For example, pointer y can be assigned to pointer x by dragging pointer y to pointer x and dropping pointer y over x. Both pointers will now point to the object pointed to by y.

Pointers can also be made to point to an object simply by grasping the pointer's arrow, dragging it over the object, and dropping it. When the pointer is over a compatible object, the object is highlighted, thus making it clear to both the instructor and student which object will be pointed to if the arrow is released.

Wiring a Data Structure. Algorithm Designer provides a generic connectivity concept through which pointers and data structure edges can be manipulated using one simple technique. The Algorithm Designer's connectivity mechanism is based on a concept borrowed from electrical circuits, which we refer to as "seeds" and "holes." Seeds are small circles from which edges can be "sprouted." Holes are connection points into which edges can be "planted." This concept, used in conjunction with the ability to drag edges with the mouse, provides a universal mechanism for manipulating simple pointers, and for inserting and removing edges from data structures that utilize edges. For example, a singly-linked list node has a seed on the right of the node from which its next pointer can be sprouted, and a hole to the left of the pointer into which a next pointer from another node can be planted. Edge-based nodes can be easily connected and these connections can be directly modified, or removed completely.

The seeds and holes concept provides a mechanism for visually identifying and supporting data structure, type-specific semantic behavior. Some objects, like linked-lists, have a specific number of incoming and outgoing connections. Other objects, like graphs, may have any number of incoming or outgoing connections. To support these varying capabilities, visual cues are given by seeds and holes associated with objects to provide users with insight into the behavior of the associated data structure node.

The user can "grow" a pointer out of a seed by clicking over it with the mouse and then dragging the mouse to the appropriate object. As the mouse is dragged away from the dot, an arrow appears and tracks the mouse. If the arrow is released over an object with a compatible hole, then the pointer is made to point to the object. This technique is very simple to learn and can be used in a generic way with simple pointers or with edges that emanate from data structures, such as lists, trees, and graphs.

Swapping Data Elements. One of the most common operations that an instructor performs on an array or other data structure is to swap the contents of two elements. The Algorithm Designer environment makes this operation easy by providing a swap operation. The instructor selects the two objects whose content should be swapped and selects the swap option. The Algorithm Designer swaps the contents of the two objects and ensures that the corresponding object property visualizations are updated properly.

3.6 Annotation

Oftentimes, a small bit of free-form text can help explain or at least remind students of an important point in the design of a data structure or in the demonstration of an algorithm. Annotations are also used to call attention to a particular object or area of the screen. The Algorithm Designer environment provides an annotation mode (the "A" button on the horizontal bar above the drawing window) whereby text annotations can be quickly and easily created, moved about, modified, or removed, in the same manner as other data objects.

3.7 Algorithm Designer Example

In this section, we will show how an instructor might demonstrate part of the algorithm for inserting into a sorted, singly-linked list. We begin the demonstration with a list consisting

of a single list node (whose value equals 10), a traversal pointer (*curr*), and a variable that holds the value to be inserted into the list (*value*). We will insert the values 5, 15, and 7 into the list.

The algorithm we are demonstrating states that if a value of the list node to which *curr* points is greater than value, then a new list node is created and inserted into the list before the current node. Since the next number to insert is 5, 5 is less than 10, and the current node is the head of the list, then the value should be prepended into the list. This process is shown in Figure 12.

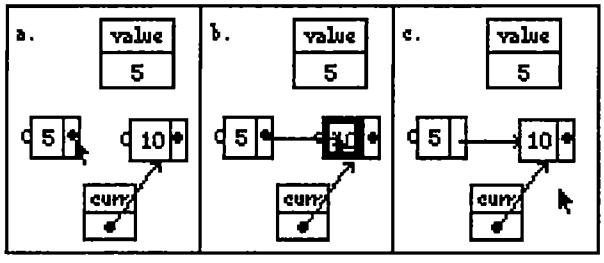


Figure 12 - Prepending to a Linked List

As discussed earlier in this paper, pointers and data structure edges are initialized by pulling an edge out of the seed contained in the body of the pointer object. Figure 12a shows the cursor positioned above the next seed of the newly created node and in Figure 12b an edge has been dragged out of the seed and on top of the head of the list. The head node has highlighted itself to indicate that the pointer may be dropped here. Figure 12c shows the new node successfully prepended into the list.

Now that a value has been inserted, we must reinitialize *curr* to point to the beginning of the list. We do this by simply dragging *curr* back to the beginning of the list (Figure 13). The instructor also could have maintained a pointer to the head of the list and simply assigned

the head pointer to the *curr* pointer. Figure 13b shows how the list element containing the value 5 highlights itself to show that the *curr* pointer can be made to point to it and Figure 13c shows the repositioning. Note that the pointer object moved itself so that it maintained the same relative position to the target object (i.e., it stays slightly to the left and bottom of the object to which it points).

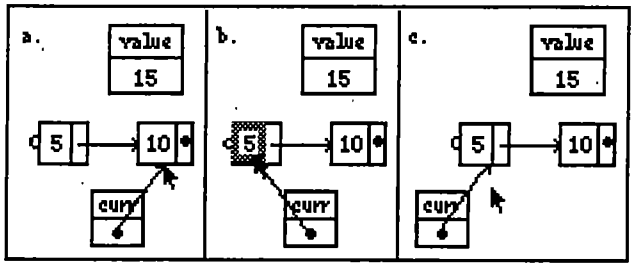


Figure 13 - List Traversal Using Pointers

The instructor is now ready to demonstrate the insertion of the next value, 15. The instructor ends up dragging the *curr* pointer to each of the two elements in the list. At this point the next rule of insertion comes into play: if the traversal pointer reaches the tail of the list, and the number to insert is larger than the tail's value, the new value should be appended to the end of the list. The steps required to create the new list node, initialize it, and then connect it to the tail node, have already been described earlier in the example description.

Finally the instructor inserts 7 into the linked list. The instructor moves the curr pointer through the elements of the list until the current pointer points to list node 10. At this point the instructor explains that the *curr* pointer cannot be used to perform the insertion since it has already moved past the insertion point. To handle this problem, a second pointer called *prev* is introduced that holds the previous value of current. If desired, the instructor could

back up to the beginning and reinitialize *curr* to the head of the list. Then the instructor could move *curr* through the list, always taking care to assign *curr* to *prev* before moving curr to the next element. Figure 14 shows the point at which current has reached node 10, *prev* points to node 5, and the insertion is being made.

Figure 14a shows the next pointer for the new list node for 7 being made to point to list node 10, while Figure 14b shows list node 5's next pointer being made to point to the new list node.

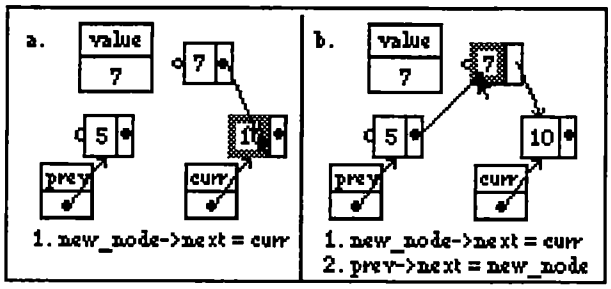


Figure 14 - Inserting into a List

Annotations have been added at the bottom of the example to indicate that these assignments should be accomplished by:

1. assigning *curr* to the next pointer of new node
   (new_node->next = curr), and
2. set list node 5's next pointer to point to new node
   (prev->next = new_node)

Note that we could have used drag-and-drop operations to make the assignments, but pedagogically, it is clearer to change the arrows directly. We use the text annotations to show what the code would actually look like.

These three insertions show the high-level details of the insertion algorithm. The instructor could now proceed to provide more explicit coding details.

3.8 Conclusions

This chapter described Algorithm Designer, an electronic whiteboarding environment. From the first use of the system by an instructor, the environment will feel familiar since it strongly resembles and behaves like a traditional drawing program. It improves upon this environment by providing syntactic and semantic support for data structure design and algorithm demonstration. It provides this support through the use of built-in data structure objects, and a small set of unique features that operate over the entire set of objects in a consistent manner. One of these features is the "seed" and "hole" concept that is used to initialize and assign pointer objects and to connect all types of edge-based structures. Another feature is the use of the swap tool that can be used to exchange two array elements, or in fact, the contents of any two data structure nodes. This reutilization of ideas helps minimize the number of features that must be learned in order to effectively use the system and helps to make the Algorithm Designer a very accessible environment for data structure design and algorithm demonstration.

Algorithm Designer whiteboard snapshots can be saved to data files and distributed to students before or after class. These files can be reloaded into Algorithm Designer to recreate a specific data structure or algorithm state. This provides students with the ability to review and replay examples and demonstrations at their own pace.

The next chapter describes Rule Designer. Rule designer adds functionality to Algorithm Designer by allowing the creation of visual language scripts to manipulate the data structures created in Algorithm Designer's electronic whiteboarding environment.

Chapter 4

Rule Designer

4.1 Introduction

The Rule Designer component is intended to provide a simple visual script creation and execution engine for the Algorithm Designer. Using cartoon strips, an instructor can create a series of "pictorial rewrite rules" that define actions to be taken upon Algorithm Designer objects. Pictorial rewrite rules provide a pattern that is matched against Algorithm Designer objects and a set of actions that are to be taken upon the instantiated Algorithm Designer objects. The visual program resulting from the set of cartoon strips can then be used to enhance the presentation of algorithm descriptions and aid in data structure design discussions.

Visual programming scripts are made up of a series of cartoon strips [Kur90]. The most basic of these strips is an execution strip. Execution strips consist of pictorial rewrite rules that define an action, or a series of actions, that are to be carried out on objects that appear in the Algorithm Designer. Other strips allow an instructor to specify loops in the visual script code, to create case-based conditional execution sequences, and to control execution within these cartoon strip blocks.

The pictorial rewrite rules used in execution strips are made up of a "Pattern Frame," and a "Result Frame." The Pattern Frame is used to specify a set of conditions which will be matched against the current program state, as defined by the Algorithm Designer. The Result Frame is used to define the actions which are to take place upon the Algorithm Designer objects, should a match be found for the pattern.

Data structure and pattern matching objects are created and modified on the Pattern Frame to create object patterns that are matched during visual script execution. Data structure objects are matched against Algorithm Designer objects by comparing their types, property values, and visual characteristics. Additional pattern matching objects are available that enhance the capabilities of the pattern matching engine. These include a pattern that can be used to specify the absence of a particular object, a text-based conditional expression pattern, and a general-purpose object tag pattern that is a short-cut for many common patterns (list head/tail, root, leaf, etc.).

During execution, Pattern Frame objects are instantiated with Algorithm Designer objects, by a pattern matching engine. A visual language interpreter then executes the actions on these objects that were specified by the Result Frame. The Interpreter performs these actions upon the instantiated Algorithm Designer objects.

An example set of cartoon strips that specify the pictorial rewrite rules necessary to perform a search of a sorted linked-list are shown in the following figures.

Figure 15 shows how the current pointer is initialized to point to the head of the linked list to be searched. The pattern shows a linked list node with an attached "Head" tag indicating that this node is the head of the list. It also shows a pointer called "current." Note that in the pattern, current does not point anywhere. This type of pointer pattern will match any pointer, regardless of where it points. In the result frame, the current pointer is initialized to point to the head of the list.

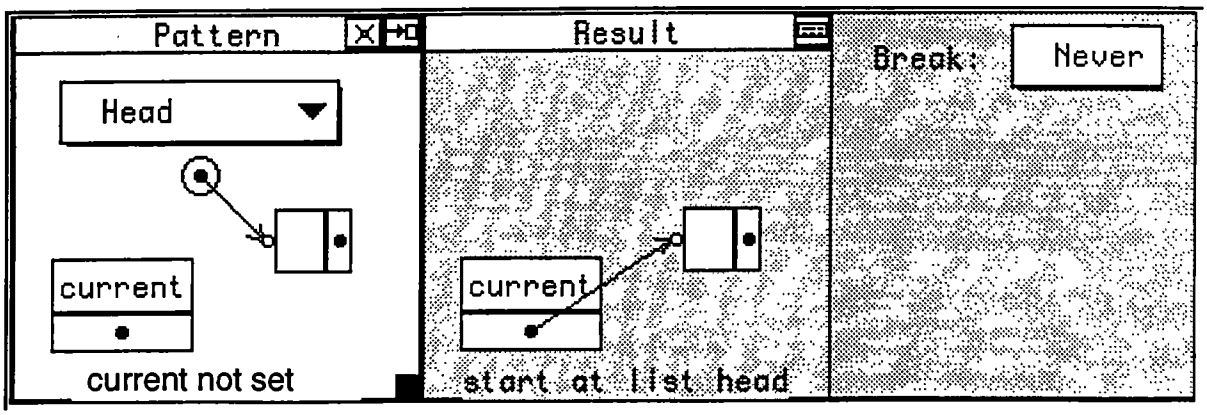


Figure 15 - Currency Pointer Initialization



Figure 16 - Begin Repeat Block

Figure 16 shows the next step of the linked list search algorithm. A loop must be begun to search through the list until the search value is found, or it is determined that the sorted linked list does not contain the value.

There are a variety of situations that may now be encountered as the search progresses through the linked list. The first of these cases is shown in Figure 17 where the search value is found in the list. The condition block is used to show that the value of the search variable must be equal to the value of the current node in the list being searched. Notice that this strip causes a break when the pattern matches, thereby leaving the current pointer pointing to the linked list node having the same value as the search variable.

The next case (shown in Figure 18) handles the condition wherein the current pointer is pointing to the tail of the list, and it is not the searched for value. We know that it is not the search value, since this is handled in the previous pattern (Figure 17). For this particular case, the result frame shows the current pointer disassociated from the linked list. The break condition for this strip is also set such that a break is caused when the pattern matches.

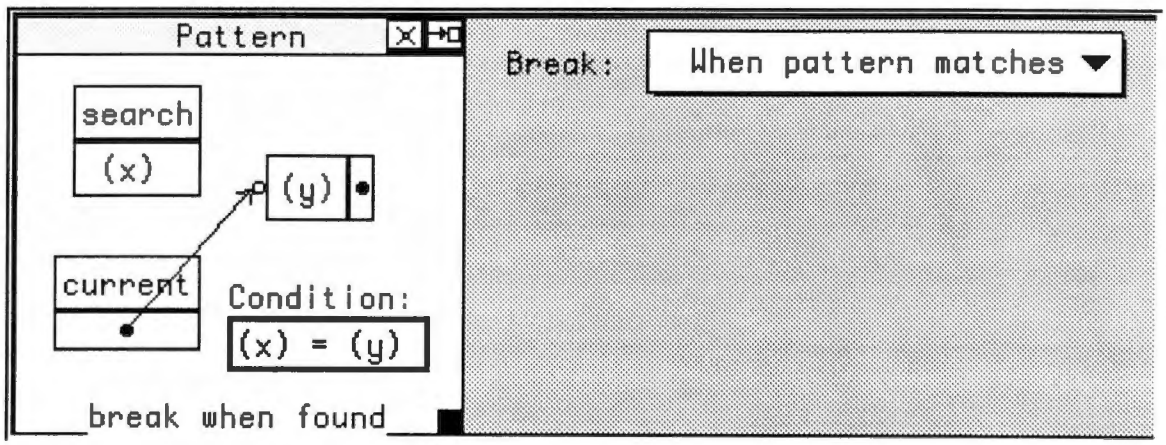


Figure 17 - Search Value Found

The next case to be handled in the sorted linked list search algorithm is shown in Figure 19. Here we see that the current linked list node contains a value that is greater than the search value. Therefore, the search value is not contained in the list. As in the previous case, the result frame causes the current pointer to be disassociated from the linked list and a break is generated.

The last case is where the search value is greater than the linked list node value. This indicates that if the sorted linked list contains the search value, it must be farther into the list. Therefore, we must move the current pointer, thereby traversing the list. Notice the use

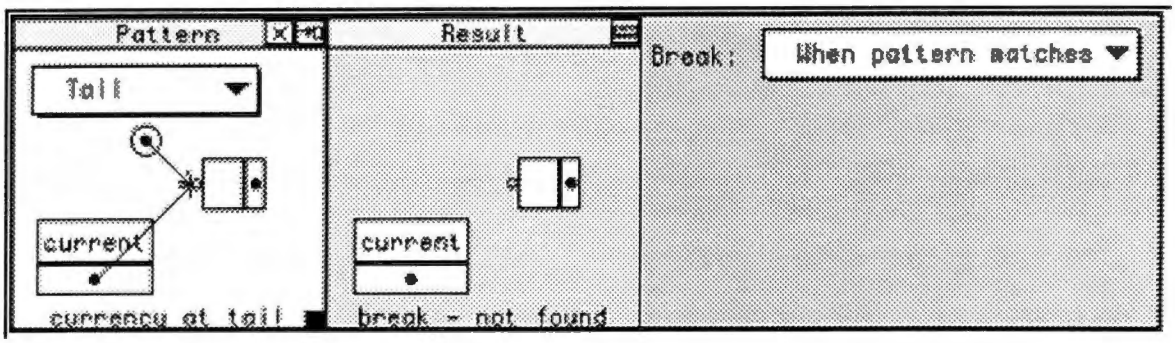


Figure 18 - End of List Condition

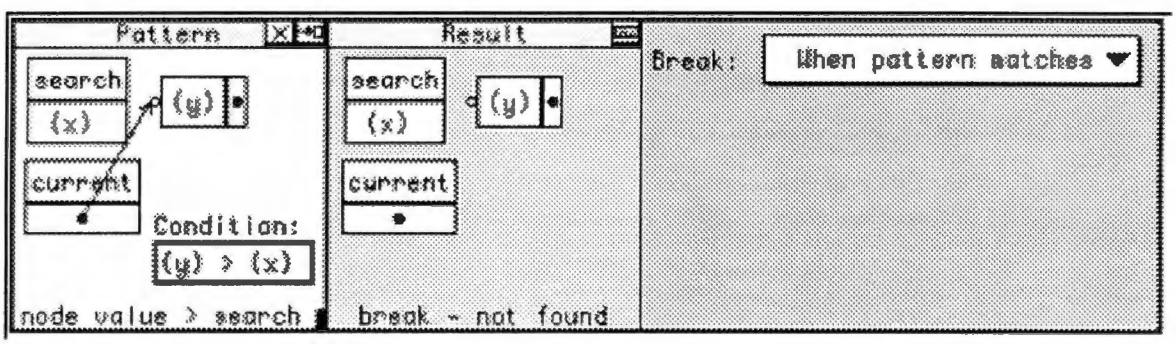


Figure 19 - Search Value Not in List

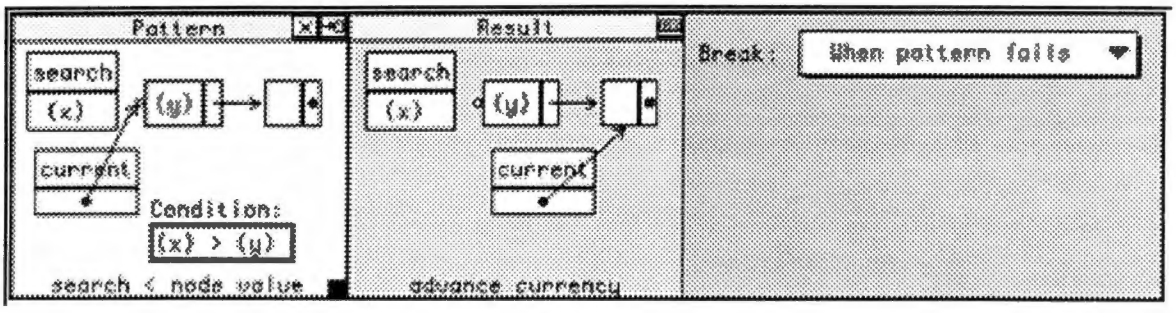


Figure 20 - Advancing List Currency



Figure 21 - End of Repeat Block

of the "Condition" object in the pattern shown in Figure 20. The result shows that the current pointer has been advanced to the next node in the list.

Figure 21 shows the end statement that designates the end of the repeat loop. This completes the Rule Designer visual program for linked list searching.

The rest of this chapter describes the Rule Designer application in detail. An overview of the entire application is presented first, followed by a description of the Rule Designer user interface, the visual code grammar, and the algorithms utilized by pattern matching engine and the visual code interpreter.

4.2 Overview

Rule Designer introduces a picture-based visual language. Data structure objects can be manipulated directly to form visual language statements. Objects created on the Pattern Frame stand for collections of object instances that are expected to be encountered during script execution. "Wildcard" matching capabilities are supported by allowing object names or property values to be left empty. Any values that are not specified are assumed to be unimportant in the pattern matching process. For example, no value is specified for the list node in Figure 15 (p. 35) since this value is of no importance to the desired pattern. Variables may also be specified for the names and values of objects, allowing more extensive patterns to be specified, and to support indirect data assignment specifications.

Objects may be created on pattern and result frames by selecting a corresponding object creation tool from a tool palette and clicking on a frame or by dragging objects from the Algorithm Designer onto a frame. When objects are dragged from the Algorithm Designer, copies are made of the object and it is added to the frame.

Once a pattern has been specified, the user may then specify the actions that are to take place upon the objects in the pattern to create the desired results. The actions are performed visually, for example, assigning one object to another is accomplished by simply dragging and dropping the assignment source object onto the target object. Visual user actions are translated and recorded into text statements which are stored for subsequent execution during visual script interpretation.

The set of visual user actions available on the Result frame is the same set of actions available on the Algorithm Designer. Therefore, a script writer may experiment with objects on the Algorithm Designer, and once a specific sequence of actions are arrived at, these actions may be repeated on the result frame. The sequence is then recorded and translated into text statements that are interpreted during visual script execution.

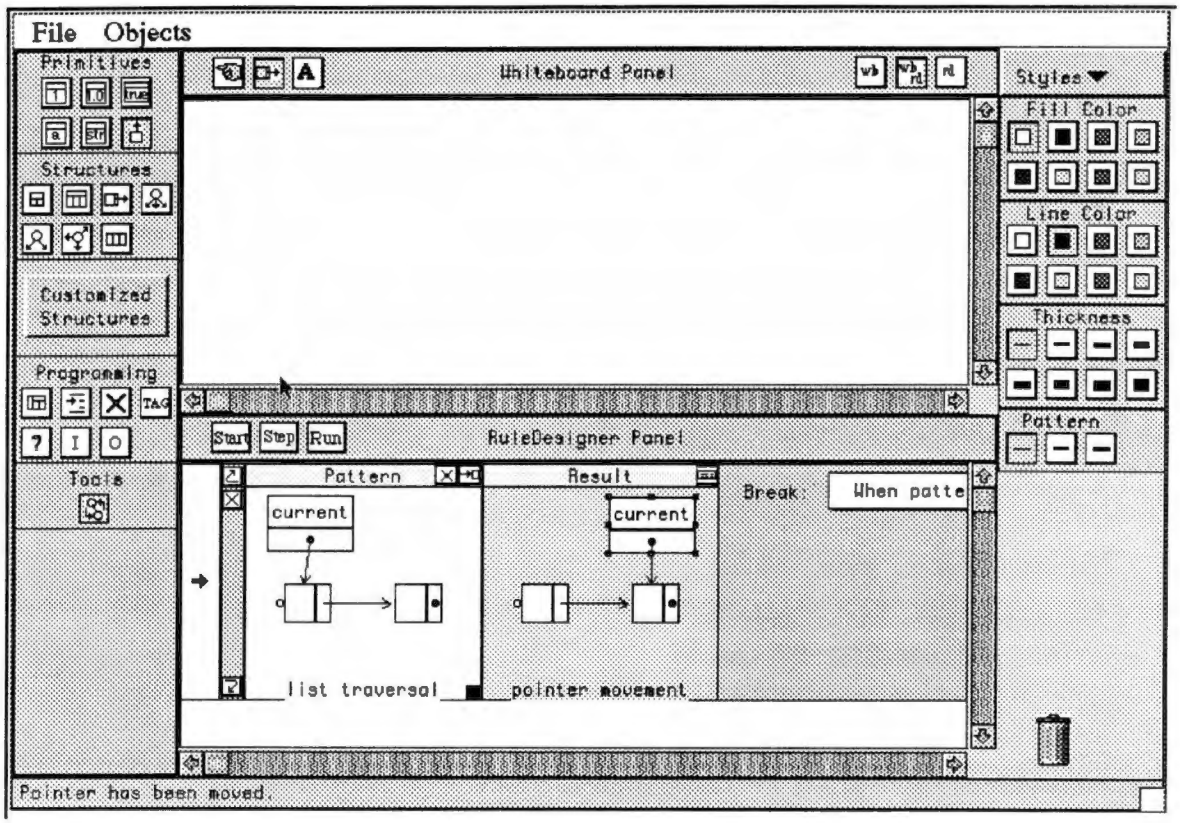


Figure 22 - Algorithm Designer and Rule Designer Environment

Figure 22 shows the entire Algorithm Designer and Rule Designer environment. Notice the buttons at the upper right of the Algorithm Designer Panel title bar. These buttons allow the user to display the Algorithm Designer only, the split-screen Algorithm Designer and Rule Designer environment shown, or the Rule Designer panel only. The buttons at the left of the Rule Designer title bar control visual script execution. These scripts can be single-stepped, or run until completion.

The Programming palette is that appears below the Customized Structures button at the left of the window is used exclusively by Rule Designer.

Rule Designer, like Algorithm Designer, allows snapshots to be saved in data files and distributed to students before or after class. These files can the be reloaded into Rule Designer to recreate a specific whiteboard state and to reload visual script frames.

4.3 User Interface

This section briefly describes the Rule Designer user interface. In Figure 23, the programming buttons are shown. These buttons create and support visual program creation within Rule Designer.



This button allows users to create Rule Designer Execution Strips. These strips contain pictorial rewrite rules that define the actions that take place upon Algorithm Designer objects during visual script execution. The initial format of this strip is shown in Figure 24.

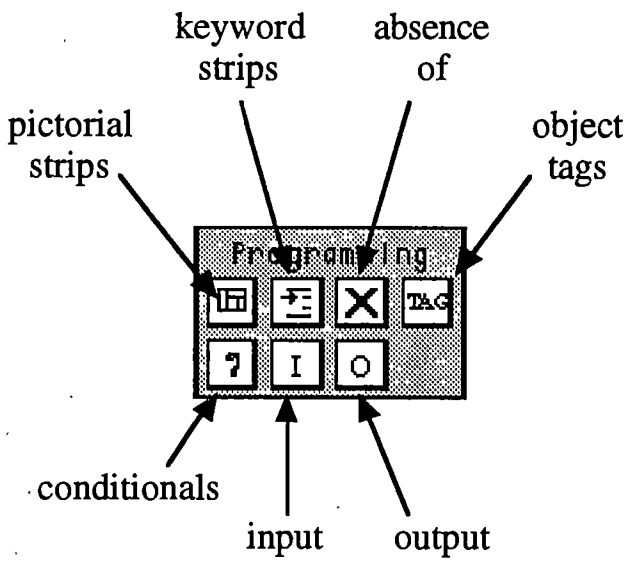


Figure 23 - Programming Buttons

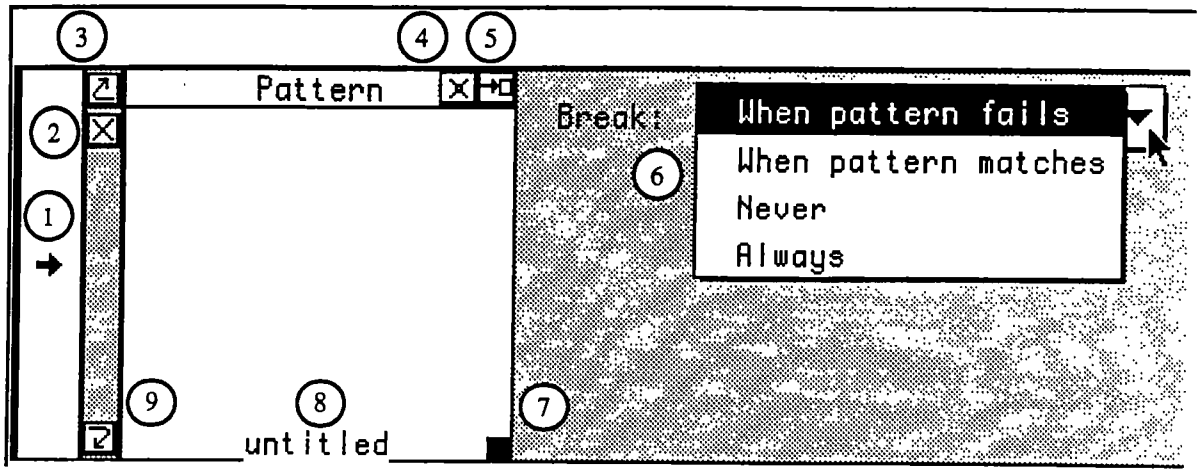


Figure 24 - Rule Designer Execution Strip

The interactive parts of the strip are labeled by the numbered circles and are described below:

1. The arrow is a currency indicator. When running a script, it points to the next strip to be executed.
2. Clicking on the X-mark deletes the strip.
3. Clicking on the upward pointing arrow when creating a new strip causes it to be inserted prior to this strip. Strips can be moved within the visual program by dragging them above or below other program strips. The upward pointing arrow highlights when another strip is dragged over the upper half of strip. This indicates that dropping the dragged strip would cause it to be moved in front of the strip.
4. This icon minimizes or maximizes the size of the strip. Clicking on it while the strip is normally sized causes it to be reduced to a small size that only shows the user-defined title to save room on the screen.

5. Clicking here copies the current contents of the Pattern frame onto the Result frame. The Result frame is only displayed once this copy occurs. The user may then define the actions that are to take place upon the objects defined in the Pattern.
6. This popup menu allows the user to define when a break will occur for the current strip. A break causes control to be returned from inside the currently defined scope. For example, if a strip is inside of a Repeat loop, a break would exit the loop. The four break options are: on pattern failure, on pattern success, always, and never. These define when a break takes place. A break normally occurs when a pattern fails. We can force a break to occur on pattern success so that once the result is executed a break is executed. We also have the option of always generating a break whether the pattern succeeds or fails, or of never generating a break, again regardless of the success or failure of a pattern match.
7. The black box at the bottom right of the Pattern frame is the resize box. It resizes both the Pattern and Result frames.
8. This is a user-defined Frame title.
9. The downward pointing arrow functions exactly like #3 except it allows insertion and movement of strips after the current strip.

When objects are created on the Pattern frame, any attributes that are left unspecified are assumed to be irrelevant to the pattern. For example, if the pattern omits the name for an integer, then the name property is assumed to be a wildcard that matches any name, and hence is ignored by the pattern matcher.

Variables can also be specified within a pattern and utilized to enhance pattern matching capabilities and to provide a symbolic framework for result actions. Variables are specified by placing a variable name within parentheses. An example of their use is shown in Figure 25. Once a variable successfully matches an Algorithm Designer object, the symbolic

variable is bound to a value. This bound value will persist through pattern matching and visual script interpretation.

In this example, the Pattern matches when we find three variables on the Algorithm Designer, one named **current**, having value x; one named **next** having value y; and some other variable with value x (the same value as **current**). In the Result frame, we see that the user has requested that the value of the unnamed variable be set to y (the value from **next**).

Also note the button at the upper right of the Result frame. Clicking on this button will display the visual program statements that have been created by the users actions on the Result frame. The grammar for this language is described later in this section.

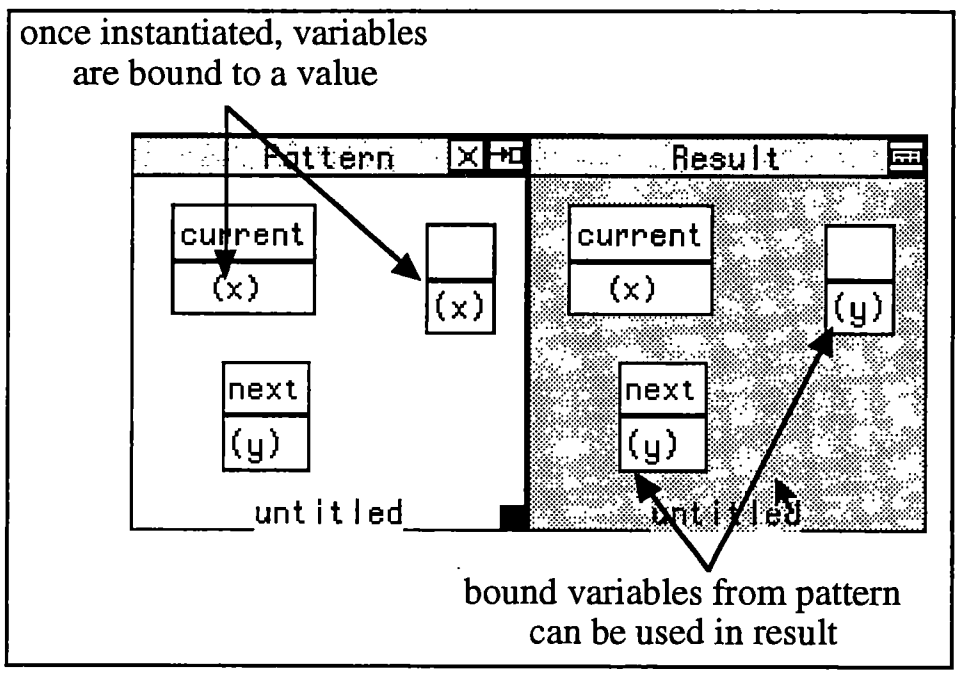


Figure 25 - Use of Variables



This button creates keyword strips as shown in Figure 26.

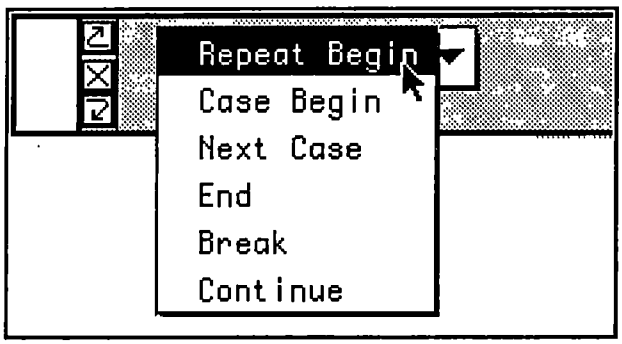


Figure 26 - Keyword Strips

Keyword strips allow users to create conditional blocks and loops. **Repeat Begin** starts a loop block, and is ended simply with **End**. Case blocks begin with **Case Begin** followed by the first case. Subsequent cases begin with **Next Case**. The case block is ended with **End**. **Break** exits out of the currently enclosing block. **Continue** is used in repeat loops to end processing of the current iteration and continue with the next iteration.

Looping conditions are specified using either a Break keyword strip, or break conditions specified on the strips themselves. When a Break keyword strip is encountered inside a Repeat block during visual script execution, the loop is exited and execution continues with the strip following the Repeat block. Another way to break out of loops is using the break conditions on the strip. For example, if the selected break condition is when the pattern fails, then when the pattern fails during script execution, a break is generated with the same results as if a Break keyword strip had been encountered.

Case statements have the following general format:

```
Case Begin
     <Case 1 strips>
Next Case
     <Case 2 strips>
Next Case
     <Case 3 strips>
End
```

Break keyword strips and strip break conditions operate within Case blocks in the same manner that they operate within Repeat blocks. Breaks always exit the most recently created block scope.

The test condition for each case is the first pattern that appears within the case. If this pattern matches objects on the Algorithm Designer, then the rest of the strips within the case are attempted. If any of these strips has a break statement that is triggered, then control will be passed to the first rule after the case statement. If no break statement is triggered, then control will be passed to the first rule after the case statement once all the strips within that case block are attempted. Note that this behavior is different from the behavior of the C switch statement in which control will drop into the next case block unless a case block is terminated by a break statement. If the first pattern for a case does not match objects on the Algorithm Designer, then execution continues with the first pattern in the next case. If there are no more cases, then the Case block is exited.

 This button creates an X-mark on a Pattern frame. These marks are used to denote the absence of either an object, or a pointer. They are primarily used to denote null pointers. They are created on a Pattern frame, then dragged and dropped onto either "stand-alone" pointers (pointers that do not point to a particular object) or "stand-alone" objects

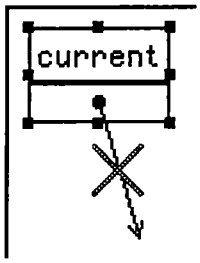


Figure 27 - Use of X-mark Object

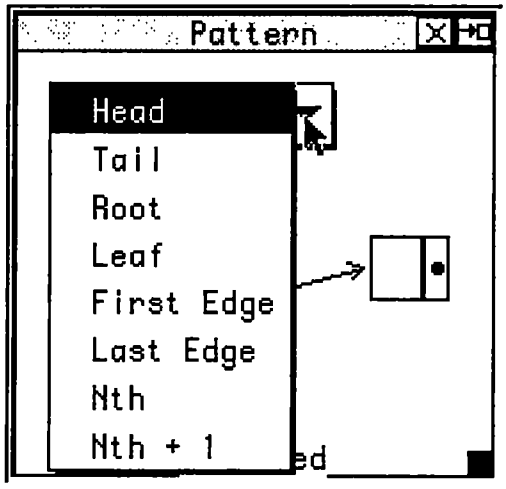


Figure 28 - Tag Options

(objects with no other objects connected to it via pointers). Figure 27 shows an X-mark used to indicated that the current pointer must be null.



This button creates tag objects on the Pattern frame. Tag objects allow the user to specify information about edges or edge-based structures. Figure 28 shows the tags that can be specified.

Tags are created on pattern frames and associated with objects by drawing out a line from the tag seed to the object to be tagged. The **Head** tag is used to indicate the beginning of a

linked list as **Tail** indicates the end of list. The **Root** and **Leaf** tags are used to indicate the root nodes and leaf nodes of trees and graphs. **First Edge** and **Last Edge** tags can be associated with edges and match the first or last edges of any edge based structure. The suitability of such a match is based upon the relative position of the edge and the orientation of the structure to which the edge belongs. For example, in a horizontally oriented structure edges are numbered top-to-bottom, whereas a vertically oriented structure has edge numbered from left-to-right. The **Nth** and **Nth + 1** tags are used in tree or graph data structures that have nodes containing more the two edges. The **Nth** and **Nth + 1** tags are used in conjunction with one another. The **Nth** tagged edge is first instantiated to an edge, then the **Nth + 1** tagged edge is matched against the edge which follows the edge matching the **Nth** tag. This allows scripts to be created that sequence through the edges emanating from a node.

An example of the use of tags appears in Figure 15 (p. 35) where the Head tag is used to denote the head of a linked list. In this example, a currency pointer is being established to point to the beginning of a list before a list traversal begins.



This button creates a conditional object on the Pattern frame. Simple conditions such as (x) > (y) can be specified in these text objects. The operands for the conditions are variables specified within the pattern objects. See the description of simple frames for a discussion of object variables.

Figure 29 shows the condition box that appears in Figure 20 (p.38) of the linked list search example described above. In Figure 20 (p. 38), the value of (x) is instantiated to the value of the search variable and the value of (y) is instantiated to the value of the current linked list node. The condition specified in Figure 29 is therefore that of a search value greater than the value of the current list node. In the algorithm presented above, this indicates that

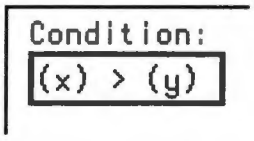


Figure 29 - Condition Box

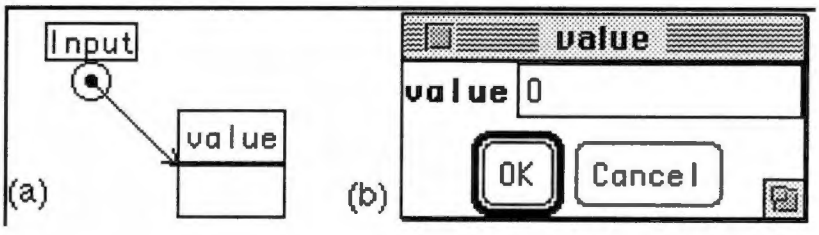


Figure 30 - Visual Script Input

the lists search must continue on to the next node in the list, since the value, if it appears in the list, is past the current node.



This button creates an Input object on the Result frame. When connected to an object, it generates a statement that when executed by the script, causes a dialog box to appear, prompting the user to enter data for its associated object.

Figure 30 part a shows an Input object associated with an object. The Input object is associated with an object by simply dragging a line out of its seed and dropping the line onto an object. Once associated, an input object may not be moved since the association action generated a visual script statement. Figure 30 part b shows the dialog box that results when an input statement is encountered during visual script execution. The values that the user inputs for the object properties replaces those properties for the input object.



This button creates an output object on the Result frame. When connected to an object, it generates a statement that when executed by the script, causes a dialog box to appear, displaying the current data for its associated object.

Figure 31 part a shows an Output object associated with an object. The Output object is associated with an object in the same manner as Input objects and tags, by simply dragging a line out of its seed and dropping it onto an object. Output object associations generate a visual script statement. Figure 31 part b shows the dialog box that results when this output statement is encountered during visual script execution. The values of output object properties are displayed to the user.

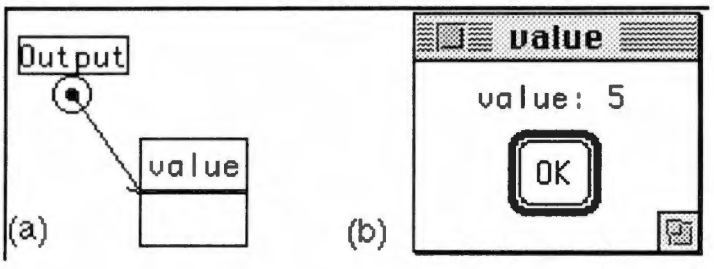


Figure 31 - Visual Script Output

4.4 Algorithms

The heart of the Rule Designer program is its pattern matching engine and it Script Interpreter. This section describes the algorithms used to implement these two components.

4.4.1 Pattern Matcher

The purpose of the Pattern Matcher is to attempt to instantiate the pattern defined on a Pattern Frame, with actual object instances on the Algorithm Designer. When successful object matches are found, an entry is made into a symbol table, and cross-references are

made between the pattern object and the Algorithm Designer object instance. These symbol table entries consist of object references to the pattern and match objects. Entries are also created for any variables defined for matched object properties. These entries contain the variable name and its instantiated value.

Figure 32 shows a high-level view of the pattern matching algorithm. The pattern matching process begins by creating two lists: a match list and a pattern list. The match list consists of Algorithm Designer objects ordered to facilitate pattern matching. To this end, the edges appear first in the list and are followed by the node and variable objects.

Once the match list is created, the pattern list is created. The pattern list consists of the Pattern Frame objects that are used as templates to match Algorithm Designer object instances. The edges appear first in this with any edges marked with an n+1 tag appearing last. The edges are followed by the node and variable objects, and finally by conditional expression objects.

While the list is being created, the stand-alone node objects marked with an x are checked to see if they exist anywhere on the Algorithm Designer. This marking is used to create a pattern that means the absence of the associated object. If a matching object is found, there is no need to continue with pattern matching.

A matching Algorithm Designer object instance must be found for each object in the pattern list for pattern matching to succeed. Pattern matching is driven by the pattern list. This list is matched in order from head to tail. Matching takes place against the match list. Each pattern object keeps track of where it last matched an object in the match list in case backtracking is necessary. The backtracking scheme is described later in this section.

```
Frame_Match
   create match_list from Algorithm Designer objects
   create pattern_list from Pattern Frame objects
   backtracking_possible = true
   match_list.Start()
   pattern_list.Start()
   while (backtracking_possible)
      p = pattern_list.Get()
      if (p = NULL)  // end of pattern list--matching has succeeded
         return true
      if (p.matched)
         match_list.Start()
         pattern_list.Next()
         continue
      m = match_list.Get_Next_Unmatched_Element()
      if (m != NULL) // not end of the match list
         if Type_Match(p, m)
            match_list.Start()
            pattern_list.Next()
      else  // end of the match list
         backtracking_possible = Backtracking(pattern_list, match_list)
   endwhile
   return false // backtracking has failed so matching has failed
end Frame_Match

Type_Match(pattern p, match m)
   on failure of any match - return no match
   match types of pattern p and match object m
   when object type of p is an edge
      match from and to objects of p and m
      match symbolic tags, edge properties and visual characteristics
   when object type is a simple variable
      match name if applicable
      match edge properties and visual characteristics
   when object type is a data structure
      match names if applicable
      match symbolic tags, edge properties and visual characteristics
end Type_Match

Backtracking(list pattern_list, list match_list)
   pattern_list.Prev()  // move to previous pattern
   p = pattern_list.Get()
   if (p = NULL)  // if we were dealing with the first pattern in the list
      return false // we cannot backtrack any further so return false

   m = p.Get_Current_Match()
   p.Uninstantiate_Match()
   symbol_table.Reset(before_match(p, m))
   match_list.Set_Cursor_After(m)
return true
```

Figure 32 - High-level Pattern Matching Algorithm

When doing name or property matching, an empty value in the pattern is treated as a wildcard value, meaning it will match any value during pattern matching. Variables are instantiated with the appropriate property value in a matched object.

Edges appear first in each list and serve to force the structure of a pattern to be matched before the nodes which are interconnected within the structure. An edge is matched by first attempting to instantiate the objects connected by the edge. Once object instantiation is completed, the edge is checked to see if it matches the characteristic denoted by any attached symbolic tags. For example, if the "First Edge" symbolic tag was connected to an edge, it would be checked to see if it was in fact the first edge emanating from its owning object's instance. Edge ordering is based on an edge's location relative to the owning object and horizontal or vertical orientation of the owner. Edges are ordered left-to-right or top-to-bottom. Following tag checking, edge matching continues by checking edge properties and visual characteristics.

Structure nodes are matched on name, associated symbolic tags, node properties, and visual style. Simple variables also match on name, properties, and style.

The Pattern Matcher must backtrack when a pattern match fails to ensure that all possible combinations have been attempted in the match of patterns to Algorithm Designer objects. As stated previously, pattern matching is driven by the pattern list. Matching is attempted upon each object in the pattern list, when a failure occurs, the previous match is uninstantiated, and a search begins for a different match. If the first pattern fails, then the entire match fails.

Each matched pattern object keeps track of the location of its matching Algorithm Designer object in the match list. If it is called upon to find another match, it continues walking the match list forward, beginning with the next object in the list following its current match.

In order to backtrack, the symbols in the symbol table that were based upon the pattern and its matching object (object id correspondences and any variable instantiations), must be removed. References stored directly within patterns and match objects that provide a cross-reference must also be cleared.

After implementing this algorithm, it was realized that original problem is actually a bipartite matching problem, a special case of a maximum matching problem and that the augmenting path method could have been used to create a much more efficient pattern matching engine [HU83]. This change was not made for three reasons: 1) the current algorithm works fine, 2) performance was not a problem for a small number of match items, and 3) since the applications are intended for research, not as commercial applications, there will never be a large number of match items.

4.4.2 Script Interpreter

When an execution strip is encountered during visual program execution, the Pattern Frame of the strip is checked by the Pattern Matcher to see if a set of matching Algorithm Designer object instantiations can be found. Once pattern matching completes, the script interpreter then uses the pattern object instantiations from the pattern matcher and the symbol table, to execute the previously generated visual program script. Visual program scripts are associated with the Result Frame on which the visual operations were performed.

The Script Interpreter uses a top-down, predictive parser to interpret visual language statements. The grammar of this language is described below.

Create Statements

The create statement has two forms, one for node creation, and one for edge creation. Nodes and edges are created on the Result Frame by selecting object buttons, and then clicking in the frame to create them. This is the same way that objects are created on the Algorithm Designer and on the Pattern Frame. The two forms of the create statement are shown below:

```
create OBJECT_TYPE <id> at <x>,<y>
create EDGE_TYPE <id> from obj <id> to obj <id>
```

Delete Statements

The delete statement again has two forms, for nodes and for edges. Nodes are deleted by either dragging them into the trash can or by selecting them and pressing the <delete> key. Edges are deleted by selecting and pressing <delete>.

```
delete obj <id>
delete edge <id>
```

Assignment Statements

Assignment statements are of three basic types. The first type handles simple assignments of literal values to Result Frame objects. The second type of assignment involves variables that have been instantiated in the pattern matching symbol table. Once the user defines a name or property as a variable in the Pattern Frame, this variable may then be used in assignment statement. The first two types of assignment statements are generated when the user types a new value into an object property. The last assignment statement format is used when one Result Frame object is assigned to another. The user requests this type of assignment by dragging and dropping one object onto another.

```
set VARIABLE_REFERENCE = "<literal >"
set VARIABLE_REFERENCE = <variable>
set VARIABLE_REFERENCE = VARIABLE_REFERENCE
```

A variable reference will be in one of the following formats:

```
obj <id>
property <name> of obj <id>
element <index> of obj <id>
property <name> of element <index> of obj <id>
```

These formats allow assignment source and targets to be complete objects, specific properties of an object, an element of an array, or a specific property of an array element.

The name of an object is changed by simply typing a new name into the space provided for the name within the object itself.

```
name of obj <id> = "<name>"
```

Visual Property Modification Statements

The visual properties of objects are edited directly by the user by selecting the object and then clicking on the property button that corresponds to the change to be made. For example to change the line color of an object to red, the user would click on the red icon in the "Line Color" palette. Here are the statements generated by changes to an objects visual characteristics:

```
line-color of obj <id> = <color>
fill-color of obj <id> = <color>
line-size of obj <id> = <width>
line-style of obj <id> = <style>
```

Object Movement Statements

Objects can be moved in the Result Frame. These movements are recorded and when played back during visual code interpretation, cause the instantiated Algorithm Designer

objects to move. The incremental positional change made to a Algorithm Designer object is the calculated difference between the starting and ending position of the pattern object.

```
move obj <id> to x, y
```

Edges are moved by dragging their end-points from one object to another. This visual action causes the one of the following statements to be generated, depending on the type of the target object of the edge:

```
move EDGE_TYPE <id> to obj <id>
move EDGE_TYPE <id> to element <index> of obj <id>
```

Swap Statement

Clicking on the "Swap" tool in the "Tools" palette swaps the values between the two objects that are currently selected. This tool works with arbitrary objects of the same type, and with array elements. The two formats of this statement are shown below:

```
swap obj <id> and obj <id> -- swap object values
swap element <index1> and element <index2> of obj <id>
```

Input Statements

The user can request that object property values be prompted for during visual script execution by associating an input object tag with an object in the Result Frame. The user first creates the input tag, then attaches it to the desired object. The resulting statement is show below:

```
input VARIABLE_REFERENCE
```

Output Statements

The user can also request that object property values be displayed via a dialog during visual script execution. This is accomplished by associating an output object tag with an object in

the Result Frame. As with input, the user first creates the output tag, then attaches it to the desired object. The output statement created by this visual action is:

```
output VARIABLE_REFERENCE
```

Arithmetic Expressions in Values

Property values can be set to arithmetic expressions. These expressions are enclosed within parentheses and may contain integers, real numbers, or variables that appear in the symbol table. The following operators are supported: +, -, *, /. The multiplication and division operators have precedence over addition and subtraction. The associativity for all operators is left-to-right. Parentheses may be nested. The following example shows a typical arithmetic expression:

```
((x + 5) * y)
```

Figure 33 shows the grammar for arithmetic expressions:

```
Statement -> (Expression)

Expression -> Expression + Expression
      | Expression - Expression
      | Expression * Expression
      | Expression / Expression
      | Expression % Expression
      | (Expression)
      | Operand

Operand -> variable | constant
```

Figure 33 - Arithmetic Expression Grammar

Conditional Expressions in Patterns

The Pattern Frame allows users to enter a simple textual conditional expression that is matched along with the visual pattern components. Conditional expressions are created by

first selecting the conditional tool from the "Programming" palette, and then clicking on a Pattern Frame. The comparison operators that are supported are: =, !=, >, <, >=, <=. The format for this expression is as follows:

```
<variable> <conditional-operator> <variable or number>
```

The variable must be one that is defined with a property field. See Figure 19 (p. 37) for an example using property variables. Boolean *and*, *or*, and *not* operators are not currently supported.

4.5 Rule Designer Limitations

The Rule Designer is intended as a visual scripting system, not a complete programming language, although it could be developed into one. The primary thrust of this project was to provide support for the visual whiteboarding environment for instructors, and in this it succeeds admirably.

Rule Designer does not support function definition or function calls and therefore, also does not support recursion. This is the primary reason that Rule Designer cannot be used as a general purpose programming environment. Rule Designer has always been envisioned in the context of an interactive Algorithm Designer system and the functionality of the visual language was designed accordingly. The language is robust enough to aid in the graphical design of data structures and in visual algorithm demonstration, and therefore succeeds in its purpose in spite of these limitations.

Another current limitation of Rule Designer is that once visual code is created, it cannot be edited either visually or as text statements. For this reason, objects appearing on the Result Frame cannot be used in visual code, and then later deleted, since the deletion would cause the reference before the delete to fail the next time the Result Frame script is executed. For

example, if during script creation you assign a value to an object on the Result Frame, then delete the object, the object will not be there when the script actually runs and the assignment is attempted. Editing of statements in the initial release of Rule Designer basically involves replacement of the visual statements by recreating the Result Frame.

One other limitation of the initial version of Rule Designer is that the user cannot directly edit hidden properties. Hidden properties are those that are not currently visualized. The user can edit these properties by opening a property editing dialog, and entering desired property modifications, however.

4.6 Conclusions

In summary, Rule Designer is a visual scripting language designed to assist in data structure visualization and algorithm demonstration. Cartoon strips are created by instructors embodying pictorial rewrite rules that act upon data structure objects in the Algorithm Designer electronic whiteboarding environment. Objects in Algorithm Designer are instantiated against script objects during visual script interpretation. The visual scripting language also includes extensions to support conditional execution and looping.

Chapter 5

Related Work

5.1 Overview

Previous research efforts that relate to the work described in this document may be divided into four main categories:

1. Tools for drawing data structures on a whiteboard
2. Pedagogical tools
3. Tools for visualizing data structures
4. Tools for visually specifying a computer program

The remainder of this chapter examines each of these areas and describes the work that has taken place as it relates to the Algorithm Designer Project.

5.2 Drawing Tools

Although it is unlikely that an instructor would use a drawing editor to explain data structures and algorithms, we will begin our discussion of related work here, since Data Structure Designer and Algorithm Designer share many of a drawing editor's capabilities.

Computer drawing programs alleviate many of the difficulties encountered in using a physical whiteboard for data structure and algorithm instruction. Objects can be drawn, moved about, saved for later use, and easily modified without necessitating a complete redrawing of the object. Complete data structure designs can also be prepared in advance and loaded when needed.

To a certain extent, the use of computer drawing programs in teaching data structure design and algorithm demonstration partakes of the main shortcoming of physical whiteboards - generality. Although objects can be created and saved, the objects are not truly dynamic and do not support behaviors that would be typical for objects of their specific type. For example, if a drawing was intended to show two connected linked list nodes, moving one of the nodes would most likely necessitate the reconnection of the arrowed-line connecting them. The line would not reflect the semantics of a linked list edge, it is simply a line and would be either grouped with one or the other of the nodes, or would be ungrouped. Either case would require additional user interaction to put the linked list back in order.

Drawing packages such as the original SketchPad [Sut63], or modern drawing programs such as CorelDraw [Cor99] and AppleWorks [App99] are designed as generic drawing tools. They can neither effectively model an object's visual appearance nor the semantics of the object's operations. Consequently, the user is rarely if ever, deceived into thinking that they are working with the actual objects that are displayed, since they must continue to work within the confines of the drawing package.

In short, computer drawing programs lack any type of syntactic or semantic support which would simplify data structure design and aid in the instruction of algorithm demonstration. This support is provided by the Algorithm Designer Project described in this document.

5.3 Pedagogical Tools

The research on computer aided instruction and learning basically focuses on multimedia applications, simulations, web-based instructional pages, and collaborative conferencing [Bal85 , Hod89, Mor86, Bar93, Sch93, Pea93, Rub93, Ste83. Can98, Kia98, Wat98, Ded98]. These multimedia applications focus sequences of video and audio clips whose order of presentation depends on a script or on actions performed by a user. The simulations can either be web-based or shown in class. The student or the instructor can manipulate the parameters in order to perform experiments. However, the simulations are typically labor-intensive to prepare and are not meant to be the central prop used in the classroom. Web-based instructional pages are meant to be used outside the classroom and do not involve interaction with an instructor. The collaboration environments typically involve some sort of video conferencing and the electronic whiteboard application could be used in such an environment to replace a blackboard. These applications tend to be supplemental to the learning process. Algorithm Designer differs from this previous research in that it is an attempt to replace the main prop used in traditional classrooms, the blackboard itself.

Algorithm Designer provides an electronic whiteboard for data structure and program state visualization and manipulation. Algorithm animation systems such as Balsa [BS84] and Tango [Sta90] provide mechanisms for visualizing the behavior of existing programs. They do not provide the interactive data structure design and algorithm development capabilities that are the focus of the Algorithm Designer environment.

The research on electronic whiteboards has focused on providing support for meetings, usually distributed meetings [Ste87, Elr92, Min91, Tan91, Coo87, Man88, Ish92, Str94, Wol92, Mor98, Ped93]. The whiteboards typically are some type of projection device and can be written on with a pen-like stylus. The electronic ink can be broadcast to other sites,

printed, or saved. Some rudimentary semantics are built in that group the ink into objects, such as shapes or handwriting, and allow the objects to be moved, deleted, cut, pasted, etc. In other words, they are providing much of the functionality of a drawing editor, but via a whiteboard rather than a mouse-based computer display.

The electronic whiteboard research conducted by Moran, van Melle, and Chiu [MVC98] is closely related to the Algorithm Designer. During their research a set of tools was created for specifying and laying out objects that could be used in a meeting. The specification tools allow a person to define a property list for an object, define action rules that indicate how the objects should respond to user actions, and to specify simple spreadsheet-like constraints that compute the values of properties based upon the values of other properties. This functionality provides semantic support for objects on a whiteboard. However, their focus is semantic support for meetings whereas the focus of this research is semantic support for data structure instruction. Their tools are more generic than Algorithm Designer in that a user can create arbitrary objects. However, the user also has to be able to program the semantic behavior. In contrast this research provides a set of pre-defined objects with pre-defined semantics. The appearance of the objects can be easily edited by setting attribute values but the user cannot define new graphical appearances for the objects. The rule designer scripting language also provides a rudimentary facility for specifying interactions between the objects (i.e., algorithms that act upon the objects).

CAT (Collaborative Active Textbooks) [BN96] is an algorithm animation system that operates in conjunction with the World-Wide Web. It is related to Algorithm Designer in that it also provides an electronic whiteboard environment. With CAT, the progress of the same algorithm can be viewed from multiple machines. Unlike Algorithm Designer, CAT algorithm animations are coded in a textual programming language, not in a visual rule-based environment.

In "Teaching Binary Tree Algorithms through Visual Programming," Michail [Mic96] has shown how visual programming concepts such as those used throughout Algorithm Designer, can be used to teach binary tree algorithms specifically, and also algorithmic concepts in general. This study asserts that the manipulation of algorithmic objects allows a student to concentrate on the higher-level aspects of the algorithm, hence increasing overall comprehension.

5.4 Data Structure Visualization

A good deal of research has gone into providing attractive displays of data for debugging purposes. The Algorithm Designer Project takes a different approach by first allowing the design and customization of textbook-like data structure representations and then providing an environment in which they can be manipulated, like the data structure objects they represent. The three components of the Algorithm Designer Project represent three different areas in which related work has taken place.

The Data Structure Designer component provides a data structure specification and customization environment. Amethyst [Aut88], Incense [Mye83], AlgorithmExplorer[McW96], and CERNO [Hos96] provide graphical displays for Pascal, Mesa, C, and object-oriented Prolog data structures respectively. Each of these systems allows the user to write code that customizes graphical data structure displays, but none of them provides a graphical editor like the Data Structure Designer for customizing these displays.

Program visualization systems graphically display some aspect of program execution, but do not allow visual program specification. Amethyst is typical of this type of system. It displays program data structures graphically as a program executes. The system is intended to help programmers understand and debug their programs.

The Algorithm Designer Project includes the visualization of program data structures, but also provides the ability to visually create and manipulate data structures and create program scripts with them as well.

5.5 Visual Programming

The Rule Designer component provides a rule-based visual language script construction environment. A number of approaches have been tried for visual program creation, including dataflow methods, programming-by-example methods, pictorial or graphical rewriting methods, and term or graph rewriting methods. The Rule Designer environment is an example of the pictorial rewriting method.

Specification-based Language Systems

Specification-based (or dataflow) visual languages allow users to draw programs graphically by placing and connecting individual program statements together to form a computer program. These visual languages are really just graphical representations of a textual program.

ProGraph [SC95] is typical of the specification-based visual languages. ProGraph provides program statement tools that can be placed on a drawing surface. The individual statements are then connected to form a data flow diagram. When completed, the diagram describes a program specification.

ProGraph differs from Rule Designer in the fundamental means through which a program is represented and manipulated graphically. Graphical objects in ProGraph merely stand for textual program statements. In Rule Designer, objects representing textbook-style high-level data structures like arrays, lists, trees, and graphs are manipulated to create program scripts.

Graphical Thinglab [Bor81, Bor86] is another representative dataflow system that provides an interface to the Thinglab system and is intended to visually construct constraints for simulations. It uses low-level operators, like plus and minus, and can only specify rather simple numeric constraints. In contrast Rule Designer focuses on higher-level data structures and operators, is procedural, and was developed for instructional purposes.

Programming-by-Example Systems

A number of programming-by-example projects allow programs to be visually specified by manipulating pictures of common data structures. ALEX allows users to manipulate pictorial images of arrays to specify matrix algorithms [Alx87]. DEAL provides a number of data type icons, such as an array icon, and allows programs to be specified by a set of action rules that modify a before-picture of the icon into an after-picture [Erw91].

These systems differ from Rule Designer in that custom data structures cannot be utilized and in that they use variables rather than concrete data to populate data structures. ALEX uses imperative programming constructs while DEAL uses functional programming. Rule Designer uses a rule-based approach using before and after frames. Both systems also differ from our environment in that they were designed to actually generate programs, whereas Rule Designer is intended to provide a scripting environment to support algorithm description and demonstration.

Another programming-by-example language is Forms3, which allows programs to be created using spreadsheet-like programming[BA94]. The programmer provides actual data as the program is being created and the formulas the programmer creates are immediately executed, thus providing immediate feedback about the results of the computation. Rule Designer differs from Forms3 in that it is an imperative rather than a functional language, and in that it is based on pictorial rewrite rules rather than formulas.

Rule Designer cartoon strip *result* frames are created by recording imperative actions taken upon a pictorial *before* pattern. A similar macro recording technique [Mau89] is used in Maulsby's MetaMouse system [Met89], Cypher's EAGER system [Cyp91], and Myer's text formatting system [Mye91].

Maulsby's system tries to guess what user action will be performed next in a drawing editor. EAGER tries to guess what the user will do next in a HyperCard environment. Myer's system tries to guess the formatting macros that should be applied to certain types of text (e.g., section headers). All three of these systems try to infer a set of actions, a macro, based on patterns they see in a user's behavior.

These systems differ from Rule Designer in a number of ways. First, they try to infer the pattern whereas Rule Designer forces the user to provide the pattern. Second, they try to infer the macros whereas Rule Designer forces the user to explicitly specify the actions. Third, they are targeted toward desktop applications while Rule Designer is targeted toward computer science instruction. Fourth, they do not provide an explicit graphical representation whereas Rule Designer does. Fifth, they create single pattern/macro rules whereas Rule Designer pattern/macro rules are like statements that are used to define a larger program.

Kurlander's Editable Histories [Kur92] is like Rule Designer in that it saves a history of a user's graphical editing operations in a cartoon strip and allows the operations in the cartoon strips to be grouped into macros. Kurlander's work differs from Rule Designer in that his macros are meant for a drawing editor whereas Rule Designer's are meant for instructional purposes. Like MetaMouse, Eager, and Cypher, Kurlander's macros are a single pattern/macro rule whereas Rule Designer pattern/macro rules are like statements that are used to define a larger program.

Pictorial Rewrite Rule Systems

Cocoa [CS95], ChemTrains [BL93], Vampire [MG92], and AgentSheets [RS95] are examples of pictorial rewrite rule-based systems.

Cocoa incorporates the use of pictorial rewrite rules to provide a children's toy simulation environment for children. Cocoa programs consist of rules that are graphical patterns that are matched against the current drawing environment, and result frames which incorporate graphical changes to the matched pattern. For example, to move a train down a straight track, the pattern could show the train locomotive with a short section of open track in front of it. The result frame could show the locomotive moved onto the previously empty track. Pictorial rewrite rules in Cocoa are used to make visual changes to graphical images. In Algorithm Designer, they are used to manipulate the program state through changes to program data structure and variables.

ChemTrains is a visual rule-based visual language system. Like Rule Designer, patterns and results are matched and executed semantically, as opposed to use of geometric mappings based on relative object positions. ChemTrains is designed as a general purpose rule-based system and lacks the data structure-specific syntactic and semantic support of Algorithm Designer.

Vampire allows some textual program code in conjunction with its pictorial rewrite rules. Rule Designer also uses text to allow the entry of simple conditional expressions in rule patterns. The systems differ in their aim. Vampire supports the development of iconic programming environments which use graphical elements as semantic building blocks upon which visual languages are constructed (like the visual program scripts of Rule Designer). Algorithm Designer aims to support data structure instruction with tools specifically designed for this purpose.

AgentSheets is another tool for creating iconic programming environments. It uses a metaphor of "communicating agents sharing a structured space." It is again targeted toward the development of general purpose visual programs, not instructional programs. Since it is a general purpose system, it lacks the rich, domain-specific data structures supported by Rule Designer.

Term or Graph Rewrite Rule Systems

Superficially, term and graph rewriting rules appear to be similar to Rule Designer's pictorial rewriting rules but that they really are quite different. The superficial resemblance is in that all three systems have a *before* pattern that gets matched and an *after* pattern that replaces the before pattern [Lel88, Der90, Klo92, Wad71]. However, term rewriting involves matching textual patterns and then replacing textual terms with other textual terms. Graph rewriting involves matching subgraphs and then replacing one subgraph with another subgraph. The pictorial rewriting of Rule Designer involves matching pictures and then performing a series of imperative operations on the matched picture to achieve a result picture. One important distinction between pictorial rewrite rules and term or graph rewriting rules is that the patterns are different. Term or graph rewrite patterns are textual or based upon graphs as opposed to being pictorial representations of a particular state.

Another distinction between term/graph rewrite systems and our pictorial rewrite system is that in term or graph rewrite systems, the *before* and *after* (i.e., *result*) patterns are both specified. An algorithm in the system then automatically figures out how to transform the *before* pattern into the *result* pattern. Rule Designer uses an imperative approach. A before pattern is first specified. The *result* pattern is created by performing a set of imperative actions on the before pattern. Rule Designer does not attempt to figure out how to generate the result pattern from the before pattern. Instead, it simply records the user's actions like a macro facility.

Chapter 6

Experience

An informal test of the Algorithm Designer software was done in front of a class of second-year Computer Science majors at Pellissippi State Technical Community College in Knoxville, TN. The test consisted of a demonstration of basic list processing algorithms (initialization, search, traversal, insertion) using first a physical whiteboard, and then using the Algorithm Designer system. The purpose of the demonstration was to gather feedback concerning the technology that had been developed. It was not intended to be, and was not treated as a formal evaluation or study. The students were simply asked to list the advantages and disadvantages of each approach.

The primary advantage listed by the students was the visual appeal of Algorithm Designer. They said that the algorithm visualization made it more interesting to learn and required less talk and therefore held their attention more readily. The Algorithm Designer data structure visualizations were crisp and clear and could compensate for the poor handwriting or drawing skills of the instructor. Also, since large examples could be produced in much less time, it was easier for them to keep their attention focused on the concepts being demonstrated not on the gestures required to erase and redraw the data structures on the physical whiteboard.

Another advantage of Algorithm Designer described by the students was that it allows for more efficient use of time, both for the instructor and for students. Examples can be created in less time, and they can also be created in advance and saved for later review. While the

physical whiteboard most likely only contains the last step of the algorithm (since previous steps are modified), the Algorithm Designer demonstration still contains all of the steps. Instructors can save these demonstrations to data files that can be distributed to students. The students can then review the example at their leisure, walking through the each step of the algorithm watching its effect, as many times as needed.

The primary disadvantage of Algorithm Designer listed on the student's responses was that it requires good projection equipment. This can make or break the usefulness of the system as can any technical problems that manifest with the equipment. Of course, the instructor could always fall back on the use of the physical whiteboard should these problems occur during a demonstration.

Another disadvantage described in student responses arises from the inherent differences between the physical whiteboard and Algorithm Designer. It was pointed out that in some ways, the physical whiteboard is more flexible than Algorithm Designer. Graphical annotations, lines, or other informational notations, can be made anywhere, at any time. In addition, there is a learning curve for instructors to learn to use any computerized system, whereas every instructor already knows how to use a whiteboard. Finally, it was noted that students are quite accustomed to seeing demonstrations on physical whiteboards, and that any change in media requires them to change the way they are accustomed to watching and interacting with an instructor.

The results obtained from the students were overwhelmingly positive and implied that Algorithm Designer can most definitely reinforce, if not replace, conventional means of algorithm and data structure demonstration. Another conclusion drawn from the responses was that the visual demonstration of data structure algorithms using Algorithm Designer allows concepts to be effectively presented in a language-neutral fashion. In other words, students can focus on pictures of what is taking place, like they would in a textbook, in

order to gain an understanding of an algorithm. The mental juggling involved in translating these pictures into text code can be delayed, since even large examples can still be easily presented visually. This helps decrease the overall complexity of the demonstration and can make it accessible to students who might otherwise become lost in low-level details.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

This dissertation describes the research done as a result of the Algorithm Designer Project. This project consisted of three components: Data Structure Designer, Algorithm Designer, and Rule Designer. The primary goal of the project was to develop the technology required for electronic instruction of computer science.

Data Structure Designer is a direct manipulation, drag-and drop graphical editor that allows users to create attractive, customized data structure objects that closely resemble the pictures of data structures that appear in textbooks. A key insight discovered during development of Data Structure Designer was that although textbooks employ a wide variety of data structure visualizations, the differences between these visualizations can be grouped into a small number of categories. Data Structure Designer provides an interface through which instructors can quickly and easily design custom data structures for use in data structure design or algorithm demonstrations by manipulating these visual characteristics.

In addition, a unique interface element was developed as part of Data Structure Designer. The color mapping widget provides an easy way for the user to associate a set of colors with a range of values in a data structure visualization. Data structure node and edge properties (or fields) can be mapped using this widget such that a given range of values will correspond to a specific fill color or line color.

Algorithm Designer explores the use of a computer as an electronic whiteboard. Algorithm Designer improves upon the traditional physical blackboard environment by providing syntactic and semantic support for data structure design and algorithm demonstration. It provides this support through the use of built-in data structure objects, and a small set of unique features that operate over the entire set of objects in a consistent manner.

One of the unique interface elements created for Algorithm Designer is the "seed" and "hole" concept. This concept is used to initialize and assign pointer objects and to connect all types of edge-based structures. The seeds and holes concept provides a mechanism for visually identifying and supporting data structure, type-specific semantic behavior.

Rule Designer provides a simple visual script creation and execution engine for Algorithm Designer. Using cartoon strips, an instructor can create a series of "pictorial rewrite rules" that define actions to be taken upon Algorithm Designer objects. Rule Designer is novel in that it uses imperative flow-of-control constructs to control which pictorial rewrite rules to execute. In previous pictorial rewrite rule systems, a list of pictorial rewrite rules was scanned. The first rule on the list whose condition evaluated to true was fired. The list of pictorial rewrite rules was then iteratively rescanned from the top. The program terminated execution when no rules could be fired. The only way to control which pictorial rewrite rules were executed was to place one pictorial rewrite rule above another in the list. In contrast, Rule Designer treats pictorial rewrite rules like statements in a program. Case statements can be used to selectively execute pictorial rewrite rules and loops can be used to

repetitively execute pictorial rewrite rules. Sequential pictorial rewrite rules are executed like sequential statements in an imperative program--Rule Designer tries to fire each rule in the sequence. Hence Rule Designer gives much finer grained control over the execution of pictorial rewrite rules than previous systems.

By providing integrated support for imperative programming constructs within a pictorial rewrite rule-based scripting system, Rule Designer demonstrates a novel use of these rules for teaching conventional imperative programming. Visual programs written with this system can then be used to enhance the presentation of algorithm descriptions and to aid in data structure design discussions.

In addition to its use in a traditional classroom setting, Algorithm Designer has potential uses as part of a distance learning program. Students in classrooms could view a projected demonstration that is being performed remotely by an instructor. Students could also walk through pre-programmed examples along with an instructor that is either present with the students, or one that is broadcasting back to a classroom from a remote site. Recorded voice-overs could also be prepared to allow students to walk through a scripted example demonstration.

Finally, both Algorithm Designer and Rule Designer allow instructors to save examples in data files which can be distributed to students before or after class for their review. These files can be reloaded into the applications allowing students to review an instructors presentation at their own pace. Example data files could be easily distributed to students via a local area network, or via FTP (File Transfer Protocol) or the World Wide Web.

7.2 Summary of Contributions to Computer Science

To summarize, the primary contributions of this dissertation to the field of Computer Science are as follows:

- Codification of the ways in which data structures are presented
- Development of pedagogical tools for CS Instructors
- Integration of pictorial rewrite rules with imperative constructs

7.3 Future Work

Currently, Algorithm Designer supports arrays, graphs, linked lists, queues, and trees in a variety of styles. Other data structures such as stacks and circular priority queues could be added to make the system more versatile.

The Rule Designer scripting language could also be made more robust. The scripting language could be extended to support function definitions and calls, and to support boolean operators within conditional expressions.

Finally, the pictorial rewrite rules of the Rule Designer scripting language could be used to generate C++-like code in order to help students see how visual operations could be translated into textual code.

Bibliography

Bibliography

[Amu95] Myers, Brad A., McDaniel Rich, Ferrency Alan, Mickish Andy, Klimovitsky Alex, and McGovern Amy. The Amulet Reference Manuals, Carnegie Mellon University Computer Science Department, Technical Report No. CMU-CS-95-166, June 1995. Also available as Technical Report No. CMU-HCII-95-102.

[AT87] Augenstein, Moshe and Tenenbaum, Aaron. Data Structures and PL/I Programming, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1987.

[Sed90] Sedgewick, Robert. Algorithms in C, Addison-Wesley Publishing Company, Inc., 1990.

[Shn83] Shneiderman, Ben. "Direct Manipulation: A Step Beyond Programming Languages," IEEE Computer, August 1983, Vol. 16 No. 8, 57-69.

[HU83] Hopcroft, A. V. and Ullman, J. D. Data Structures and Algorithms, Addison-Wesley Publishing Company, Inc., 1983.

[Sut63] Sutherland, Ivan E. "Sketchpad: A Man-Machine Graphical Communication System," In 1963 AFIPS Spring Joint Computer Conference Vol. 23, 329-346.

[Cor99] CorelDraw by Corel. Web page at: http://www.corel.com/products/graphicsandpublishing/draw8/main.html

[App99] AppleWorks by Apple Computer, Inc. Web page at: http://www.apple.com/appleworks/

[Bal85] Balkovich, Ed, Lerman, Steven and Parmelee, Richard. "Computing in Higher Education: The Athena Experience," In IEEE Computer, November 1985, Vol. 18, No. 11, 112-125.

[Cha90] Champine, George A., and Geer, Daniel E. "Project Athena as a Distributed Computer System," In IEEE Computer, September 1990, Vol. 23, No. 9, 40-51.

[Hod89] Hodges, Matthew E., Sasnett, Russell M., and Ackerman, Mark S. "A Construction Set for Multimedia Applications," In IEEE Software, January 1989, Vol. 6, No. 1, 37-43.

[Mor86] Morris, Satyanarayanan, Conner, Howard, Rosenthal, and Donelson. "Andrew: A Distributed Personal Computing Environment," In Communications of the ACM, March 1986, Vol. 29, No. 3, 184-201.

[Bar93] Barron, Brigid and Kantor, Ronald J. "Tools to Enhance Math Education: The Jasper Series," In Communications of the ACM, May 1983, Vol. 36, No. 5, 52-54.

[Sch93] Schank, Roger C. "Learning via Multimedia Computers," In Communications of the ACM, May 1983, Vol. 36, No. 5, 54-56.

[Pea93] Pea, Roy D. "The Collaborative Visualization Project," In Communications of the ACM, May 1983, Vol. 36, No. 5, 60-63.

[Rub93] Rubin, Andee. "Video Laboratories: Tools for Scientific Investigation," In Communications of the ACM, May 1983, Vol. 36, No. 5, 64-65.

[Ste83] Stevens, Albert, Roberts, Bruce, and Stead, Larry. "The Use of A Sophisticated Graphics Interface in Computer-Assisted Instruction," In IEEE Computer Graphics and Applications, March/April 1983, Vol. 3, No. 2, 25-31.

[Can98] Candiotti, Alan and Clarke, Neil. "Combining Universal Access with Faculty Development and Academic Facilities," In Communications of the ACM, January 1998, Vol. 41, No. 1, 36-41.

[Kia98] Kiaer, Lynn, Mutchler, David, and Froyd, Jeffrey. "Laptop Computers in an Integrated First-Year Curriculum," In Communications of the ACM, January 1998, Vol. 41, No. 1, 45-49.

[Wat98] Watters, Carolyn, Conley, Marshall, and Alexander, Cynthia. "The Digital Agora: Using Technology for Learning in the Social Sciences," In Communications of the ACM, January 1998 Vol. 41, No. 1, 50-57.

[Ded98] Deden, Ann. "Computers and Systemic Change in Higher Education," In "Communications of the ACM, January 1998, Vol. 41, No. 1, 58-63.
[BS84]Brown, M. H. and Sedgewick, R. "A System for Algorithm Animation." Computer Graphics, July 1984, Vol. 18 No. 3, 177-186.

[Sta90] Stasko, J.T. "Tango: A Framework and System for Algorithm Animation." Computer, September 1990, Vol. 23, No. 9, 27-39.

[Ste87] Stefik, Foster, Bobrow, Kahn, Lanning, and Suchman. "Beyond the chalkboard: Computer support for collaboration and problem solving in meetings," In Communications of the ACM, January 1987, Vol. 30, No. 1, 32-47.

[Elr92] Elrod, Bruce, Gold, Goldberg, Halasz, Janssen, Lee, McCall, Pedersen, Pier, Tang, and Welch. "LiveBoard: A Large Interactive Display Supporting Group Meetings, Presentaions and Remote Collaboration," from 1992 Proceedings of SIGCHI, In Human Factors in Computing Systems, p. 599-607.

[Min91] Minneman, S. L., and Bly, S. A. "Managing a Trois: A Study of a Multi-User Drawing Tool in Distributed Design Work," from 1991 Proceedings of SIGCHI, In Human Factors in Computing Systems, 217-224.

[Tan91] Tang, John C. and Minneman, Scott L. "VideoWhiteboard: Video Shadows to Support Remote Collaboration," from 1991 Proceedings of SIGCHI, In Human Factors in Computing Systems, 315-322.

[Coo87] Cook, Ellis, Graf, Rein, and Smith. "Project Nick: Meetings Augmentation and Analysis," In ACM Transactions on Office Information Systems, April 1987, Vol. 5, No. 2, 132-146.

[Man88] Mantei, M. "Capturing the Capture Lab Concepts: A Case Study in the Design of Computer Supported Meeting Environments," In 1988 Proceedings of the Conference on Computer-Supported Cooperative Work, 257-270.

[Ish92] Ishii, Hiroshi, and Kobayashi, Minoru. "ClearBoard: A Seamless Medium for Shared Drawing and Conversation with Eye Contact," from 1992 Proceedings of SIGCHI, In Human Factors in Computing Systems, 525-532.

[Str94] Streitz, Geissler, Haake, and Hol. "Dolphin: Integrated Meeting Support Across Local and Remote Desktop Environments and Liveboards," In 1994 Proceedings of the Conference on Computer-Supported Cooperative Work, 345-358.

[Wol92] Wolf, C., Rhyne, J., and Briggs, L. "Communication and Information Retrieval with a Pen-based Meeting Support Tool," In 1992 Proceedings of the Conference on Computer-Supported Cooperative Work, 322-329.

[Mor98] Moran, Thomas P., van Melle, William, Chiu, Patrick. "Tailorable Domain Objects as Meeting Tools for an Electronic Whiteboard," In 1998 Proceedings of the Conference on Computer-Supported Cooperative Work, 295-304.

[Ped93] Pedersen, McCall, Moran, and Halasz. "Tivoli: An Electronic Whiteboard for Informal Workgroup Meetings," from 1993 Proceedings of SIGCHI, In Human Factors in Computing Systems, 391-398.

[MVC98] Moran, Thomas P., van Melle, William, and Chiu, Patrick. "Spatial Interpretation of Domain Objects Integrated into a Freeform Electronic Whiteboard," In Proceedings of 1998 ACM SIGGRAPH Symposium on User Interface Software and Technology, 175-184.

[BN96] Brown, Marc H. and Najork, Marc A. "Collaborative Active Textbooks: A Web-Based Algorithm Animation System for an Electronic Classroom," In 1996 IEEE Symposium on Visual Languages, 266-275.

[Mic96] Michail, Amir. "Teaching Binary Tree Algorithms through Visual Programming," In 1996 IEEE Symposium on Visual Languages, 38-45.

[Aut88] Myers, Brad A., Chandhok, Ravinder, and Sareen, Atul. Automatic Data Visualization for Novice Pascal Programmers, In Proceedings of 1988 IEEE Workshop on Visual Languages (Pittsburg, PA, Oct. 10-12), IEEE Computer Society Press, 192-198.

[Mye83] Myers, Brad A. "Incense: A System for Display Data Structures," In Proceedings of 1983 SIGGRAPH Conference (Detroit, MI, July), ACM Computer Graphics Vol. 17 No. 3, 115-125.

[McW96] McWhirter, Jefferey D. "AlgorithmExplorer: A Student Centered Algorithm Animation System," In Proceedings of 1996 IEEE Workshop on Visual Languages (Boulder, CO, Sept. 3-6), IEEE Computer Society Press, 174-181.

[Hos96] Hosking, John G. "Visualization of Object Oriented Program Execution," In Proceedings of 1996 IEEE Workshop on Visual Languages (Boulder, CO, Sept. 3 6), IEEE Computer Society Press, 190-191.

[SC95] Steinman, Scott B., and Carver, Kevin G.Visual Programming with Prograph CPX, Manning Publications Company, Greenwich, CN, 1995.

[Bor81] Borning, Alan. "The Programming Language Aspects of ThingLab; a Constraint-Oriented Simulation Laboratory," In ACM Transactions on Programming Languages and Systems, October 1981, Vol. 3, No. 4, 353-387.

[Bor86] Borning, Alan. "Defining Constraints Graphically," from 1986 Proceedings of SIGCHI, In Human Factors in Computing Systems, 137-143.

[Alx87] Kozen, Dexter, Teitelbaum, Tim, Chen, Wilfred, Field, John, Pugh, William, and Vander Zanden, Brad. "ALEX - an Alexical Programming Language," In Proceedings of 1987 Workshop on Visual Languages (Linkoping, Sweden, Aug. 19-21), 315-329.

[Erw91] Erwig, Martin. "DEAL - A Language for Depicting Algorithms," In Proceedings of 1991 IEEE Workshop on Visual Languages (St. Louis, MO, Oct. 4-7), IEEE Computer Society Press, 184-185.

[BA94] Burnett, M. and Ambler, A. Interactive Visual Data Abstraction in a Declarative Visual Programming Language. Journal of Visual Languages and Computing, March 1994, 29-60.

[Mau89] Maulsby, David L. and Witten, Ian H. "Inducing Procedures in a Direct-Manipulation Environment," from 1989 Proceedings of SIGCHI, In Human Factors in Computing Systems, 57-62.

[Met89] Maulsby, David L., Witten, Ian H., and Kittlitz, Kenneth A. "Metamouse: Specifying Graphical Procedures by Example," In 1989 Proceedings SIGGRAPH, Vol. 23, No. 3, 127-136.

[Cyp91] Cypher, Allen. "EAGER: Programming Repetitive Tasks by Example," from 1991 Proceedings of SIGCHI, In Human Factors in Computing Systems, 33-39.

[Mye91] Myers, Brad A. "Text Formatting by Demonstration," from 1991 Proceedings of SIGCHI, In Human Factors in Computing Systems, 251-256.

[Kur92] Kurlander, David and Feiner, Steven. "A History-Based Macro By Example System," from 1992 Proceedings of UIST, In ACM SIGGRAPH Symposium on User Interface Software and Technology, 99-106.

[CS95] Cypher, Allen and Smith, David C. "KidSim: End User Programming of Simulations". In Proceedings of CHI, 1995 (Denver, May 7 - 11). ACM, New York, 1995, 27-34.

[BL93] Bell, Brigham and Lewis, Clayton. "ChemTrains: A Language for Creating Behaving Pictures," In 1993 IEEE Symposium on Visual Languages. IEEE Computer Society, Bergen, Norway, August 1993, 188-195.

[MG92] McIntyre, David E. and Glinert, Ephraim P. "Visual Tools for Generating Iconic Programming Environments." In Proceedings of IEEE Computer Society Workshop on Visual Languages, 1992, 162-168.

[RS95] Repenning, A. and T. Sumner. "Agentsheets: A Medium for Creating Domain-Oriented Visual Languages." IEEE Computer, January 1995, Vol. 28 No. 1, 17-25.

[Lel88] Leler, W. Constraint Programming Languages: Their Specification and Generation, Addison-Wesley Publishing Company, Inc., 1988.

[Der90] Dershowitz, N. and Jouannaud, J. P. Handbook of Theoretical Computer Science. Elsevier, Amsterdam, 1990.

[Klo92] Klop, J. W. Handbook of Logic in Computer Science, Vol. 2. Oxford University Press, New York, NY, 1992.

[Wad71] Wadsworth, C. P. "Semantics and pragmatics of the lambda-calculus," PhD dissertation, University of Oxford, Oxford, England, 1971.

VITA

David R. Brown was born in Lee County, Virginia on January, 29, 1961. He grew up in Elizabethton and Bristol, Tennessee where he graduated from Tennessee High School in 1979.

David received his Bachelor of Science degree in Computer Science from East Tennessee State University (ETSU) in Johnson City, TN in 1983. He then worked for North American Phillips Consumer Electronics Corporation in Greeneville, TN and Knoxville, TN until September, 1984 when he accepted a position with Martin-Marietta Energy Systems Corporation in Oak Ridge, TN. David worked for Martin-Marietta (and later for Lockheed-Martin) in Oak Ridge until January 1997. During this time he developed IBM mainframe manufacturing applications at the Y-12 manufacturing facility, DEC VAX, Macintosh and Windows applications at Oak Ridge National Laboratory, and HTML/CGI Oracle applications for the K-25 plant.

David continued his education and in 1989, he completed his Master of Science degree in Computer Science at the University of Tennessee. His thesis topic was "Computer Virus Construction, Proliferation, and Control." He began work on his Doctor of Philosophy degree in Computer Science in 1993, and this dissertation represents the conclusion of his Ph.D. program.

David left Lockheed-Martin in 1997 and worked as a Teaching Assistant and later as a Research Assistant at the University of Tennessee until August of 1998 when he accepted a teaching position at Pellissippi State Technical Community College (PSTCC). He is now a full-time Computer Science Technology Instructor at PSTCC.