END-USER SPECIFICATION OF INTERACTIVE DISPLAYS

(Ph.D. Dissertation)

Shamim Mohamed

TR 92-35

September 16, 1993

Department of Computer Science
The University of Arizona
Tucson, Arizona 85721

This research was partially supported by the National Science Foundation under grants IRI-8702784, CDA-8822652 and IRI-9015407.

END-USER SPECIFICATION OF INTERACTIVE DISPLAYS

by

Shamim Mohamed

A Dissertation Submitted to the Faculty of the

DEPARTMENT OF COMPUTER SCIENCE

In Partial Fulfillment of the Requirements

For the Degree of

DOCTOR OF PHILOSOPHY

In the Graduate College

THE UNIVERSITY OF ARIZONA

1 9 9 3

END-USER SPECIFICATION OF INTERACTIVE DISPLAYS

Shamim Mohamed, Ph. D.
The University of Arizona, 1993

Director: Larry L. Peterson

Presenting data graphically can often increase its understandability—well-designed graphics can be more effective than a tabular display of numbers. It is much easier to get an understanding of the relationships and groupings in data by looking at a pictorial representation than at raw numbers. Most visualization systems to date, however, have allowed users to only choose from a small number of pre-defined display methods. This does not allow the easy development of new and innovative display techniques.

These systems also present a static display—users cannot interact with and explore the data. More innovative displays, and the systems that implement them, tend to be extremely specialised, and closely associated with an underlying application. We propose techniques and a system where the user can specify most kinds of displays. It provides facilities to integrate user-input devices into the display, so that users can interact and experiment with the data. This encourages an exploratory approach to data understanding.

Most users of such systems have the sophistication to use advanced techniques, but conventional programming languages are too hard to learn just for occasional use. It is well known that direct manipulation is a powerful technique for novice users; systems that use it are much easier to learn and remember for occasional use. We provide a system that uses these techniques to provide a visualization tool. Extensions to the WYSIWYG (What You See Is What You Get) metaphor are provided to handle its shortcomings, the difficulty of specifying deferred actions and abstract objects. In the data graphics domain, the main drawbacks of WYSIWYG systems are the difficulty of allowing a variable number of data items, and specifying conditional structures.

This system also encourages re-use and sharing of commonly used display idioms. Pre-existing displays can be easily incorporated into new displays, and also modified to suit the users' specific needs. This allows novices and unsophisticated users to modify and effectively use display techniques that advanced users have designed.

STATEMENT BY AUTHOR

This dissertation has been submitted in partial fulfillment of the requirements for an advanced degree at The University of Arizona and is deposited in the University Library to be made available to borrowers under the rules of the Library.

Brief quotations from this dissertation are allowable without special permission, provided that accurate acknowledgement of the source is made. Requests for permission for extended quotation from or reproduction of this manuscript in whole or in part may be granted by the Head of the major department or the Dean of the Graduate College when in his or her judgement the proposed use of the material is in the interests of scholarship. In all other instances, however, permission must be obtained from the author.

SIGNED: ______________________________

ACKNOWLEDGEMENTS

I couldn't have made it this far without the help of many people. I wish I could do more to express my gratitude, but you will have to be satisfied with having your name recorded for posterity!

I'd like to thank my advisor, Scott Hudson, for all the guidance and support, along with the freedom to find my own way, as well as for being a friend. Thanks also to the other faculty members at Arizona, particularly Rick Schlichting for his valuable guidance and irreverent attitude, Steve Mahaney for showing me other disciplines, Mary Bailey and Rick Snodgrass for providing some very useful insights, and John Leonard and Art Winfree for some fresh viewpoints.

I'd also like to thank the other members of the User Interface Group at Arizona—Tyson Henry, Andrey Yeatts, Bob Simms and Gary Newell. But for their camaraderie and support, and the pressure of the weekly meetings (for which thanks go again to the Meeting Czar, Bob) I might have run out of steam.

Huge thanks to all the CS grad students at Arizona—in particular Mike Soo and Ed Menze, for the cycling and bar-hopping; Vic Thomas (who also saved me many grey hairs by helping me in the constant battle against TEX and for being the "he's been in grad-school longer" scapegoat); Curtis Dyreson and his work ethic; Peter Bigot, especially for his TEXpertise; Pat Homer; Nick Kline; Mark Abbott; Siva Kumar; Clint Jeffery; Shivakant Mishra; all the members of the "lunch train" (which I still think is a repugnant name); Herman and Zoë Rao; and of course Bob Mercier and Bala Vasireddi for the *joie de vivre*. Thanks also go to Mike Coffin, for writing the LATEX dissertation style, and to Eric Dyreson for being a guinea pig.

Thanks and eternal gratitude go to D'Anne Thompson—the finest pilot, instructor, and mentor anywhere.

Thanks also to D. B. Phatak and D. M. Dhamdhere of IIT Bombay for their enthusiasm, and to all the IITians: Sohoni, RD, Jog, Mosh, Mad (all three), Meena, Cheeps, Hedge, Patkar, Baliga, Anal, Vivek, Deven, Mayya, Tan, Gundhy, Apte, Vijayakar, Ghare, Sanjoy, Murthy, Licky, 'Droid, Biru, KD, Kapils, Gus, Ghoda, RLC, Chandshah... it's all your fault!

Thanks go to everyone out there releasing free software. The editor I use (gnuemacs), the compilers (gcc and g++), the debugger (gdb), the typesetting program (LATEX, TEX, pstricks and dvips), the previewer (ghostscript/ghostview), the window system (X11), the shell (bash), the graph package (gnuplot), drawing program (idraw), mailer (elm), news-reader (trn), ... all free. In keeping with the spirit of the FSF (Free Software Foundation), all of the work presented in this dissertation (including source code) is freely available, to anyone, for any use under the terms of the Gnu Public License.

Special thanks to Bubu, Michelle and Natasha.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Presenting data graphically can often increase its understandability—well-designed graphics can be more effective than a tabular display of numbers. It is much easier to get an understanding of the relationships and groupings in data by looking at a pictorial representation than at raw numbers. Most visualization systems to date, however, have allowed users to only choose from a small number of pre-defined display methods. This does not allow the easy development of new and innovative display techniques.

These systems also present a static display—users cannot interact with and explore the data. More innovative displays, and the systems that implement them, tend to be extremely specialised, and closely associated with an underlying application. We propose techniques and a system where the user can specify most kinds of displays. It provides facilities to integrate user-input devices into the display, so that users can interact and experiment with the data. This encourages an exploratory approach to data understanding.

Most users of such systems have the sophistication to use advanced techniques, but conventional programming languages are too hard to learn just for occasional use. It is well known that direct manipulation is a powerful technique for novice users; systems that use it are much easier to learn and remember for occasional use. We provide a system that uses these techniques to provide a visualization tool. Extensions to the WYSIWYG (What You See Is What You Get) metaphor are provided to handle its shortcomings, the difficulty of specifying deferred actions and abstract objects. In the data graphics domain, the main drawbacks of WYSIWYG systems are the difficulty of allowing a variable number of data items, and specifying conditional structures.

This system also encourages re-use and sharing of commonly used display idioms. Pre-existing displays can be easily incorporated into new displays, and also modified to suit the users' specific needs. This allows novices and unsophisticated users to modify and effectively use display techniques that advanced users have designed.

Chapter 1
Introduction

Graphical presentation is at the heart of effective data analysis. Graphics reveal data—well-designed data graphics are often the simplest and the most powerful method for analyzing information. To quote Tufte: "Often the most effective way to describe, explore and summarize a set of numbers—even a very large set—is to look at pictures of those numbers." [67]

We often use "see" as a synonym for "understand." Indeed, the visual cortex is one of the largest parts of the human brain and vision is the most important source of information about the world for most of us. Exploiting this innate ability effectively can allow a much faster and more insightful approach to data analysis than a tabular presentation. Looking at data in the form of numbers requires a cognitive effort, which can be avoided with a visual representation; relationships between quantities and trends in a graph can be seen at a glance. It is clichéd but often true that one picture is worth a thousand words.

Computer based display and visualization systems also have the potential to further increase the understandability of data by creating interactive data graphics. Interactive displays and animation bring the data even closer to the physical world, making them easier still to understand. This interactive approach encourages users to explore and experiment with the data. In this work we explore the use of visual programming for the specification of exploratory data displays.

A *display* is traditionally defined as a graphical representation of quantitative data. We extend this notion to include any objects responsible for controlling the aspects of the diagram by incorporating user input like buttons, sliders etc. We will refer to this combination of input devices and the visual representation of data as an interactive display, or simply as a display.

As an example, Figure 1.1 presents one of the earliest uses of graphics to illustrate data. It was drawn in the mid-19th century by Minard (and recently re-emphasized by Tufte, [67]) and is a representation of the march of Napoleon's army on Moscow in the winter of 1812/13. The progress of the army eastward is shown by a line whose width represents its strength. The advance toward Moscow is colored grey, and the retreat is colored black. A subsidiary chart below the main one represents the temperatures they encountered on their retreat from Moscow, cross-referenced to the position. It is an effective presentation of the data, and clearly highlights some important events that would be hard to see in a tabular representation—for instance, the crossing of the Bérézina river in retreat was particularly bad, with almost half the men dying. The temperature at the time was -20° Réamur (-25°C, -13°F).

A *specification* is an abstract representation of some desired system. For example, a program in a conventional programming language like Pascal can be considered to be a specification for an execution of a virtual machine implemented by the programming language. In the present domain of interest, that of interactive data display (or visualization), a specification is an abstract representation of a display to be created by an interpreter.

Effective visual displays can be very useful, and many systems are available that offer means of visualizing data. Most end-user display specification systems to date have either been visual

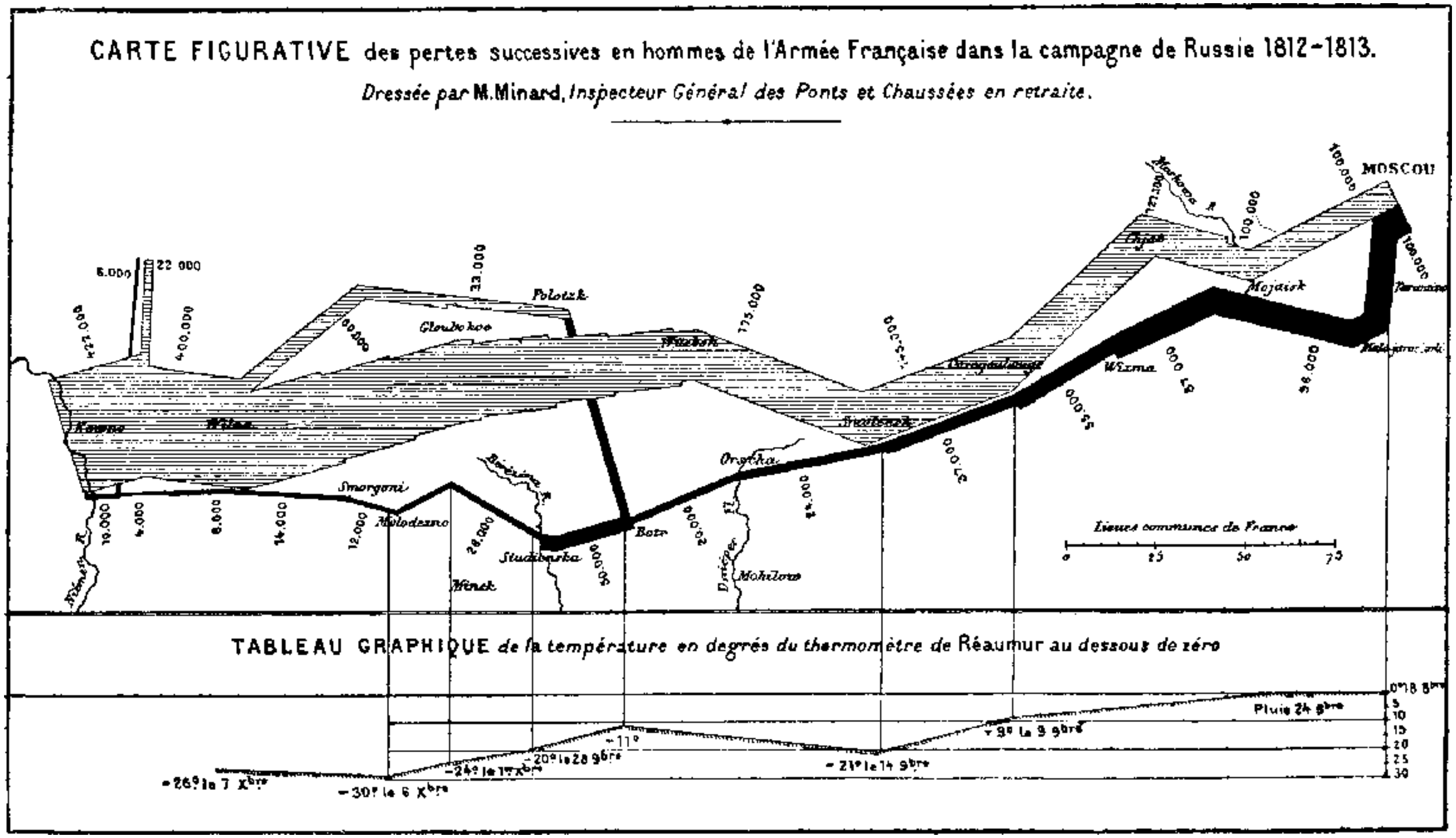

Figure 1.1: Napoleon's march across Russia

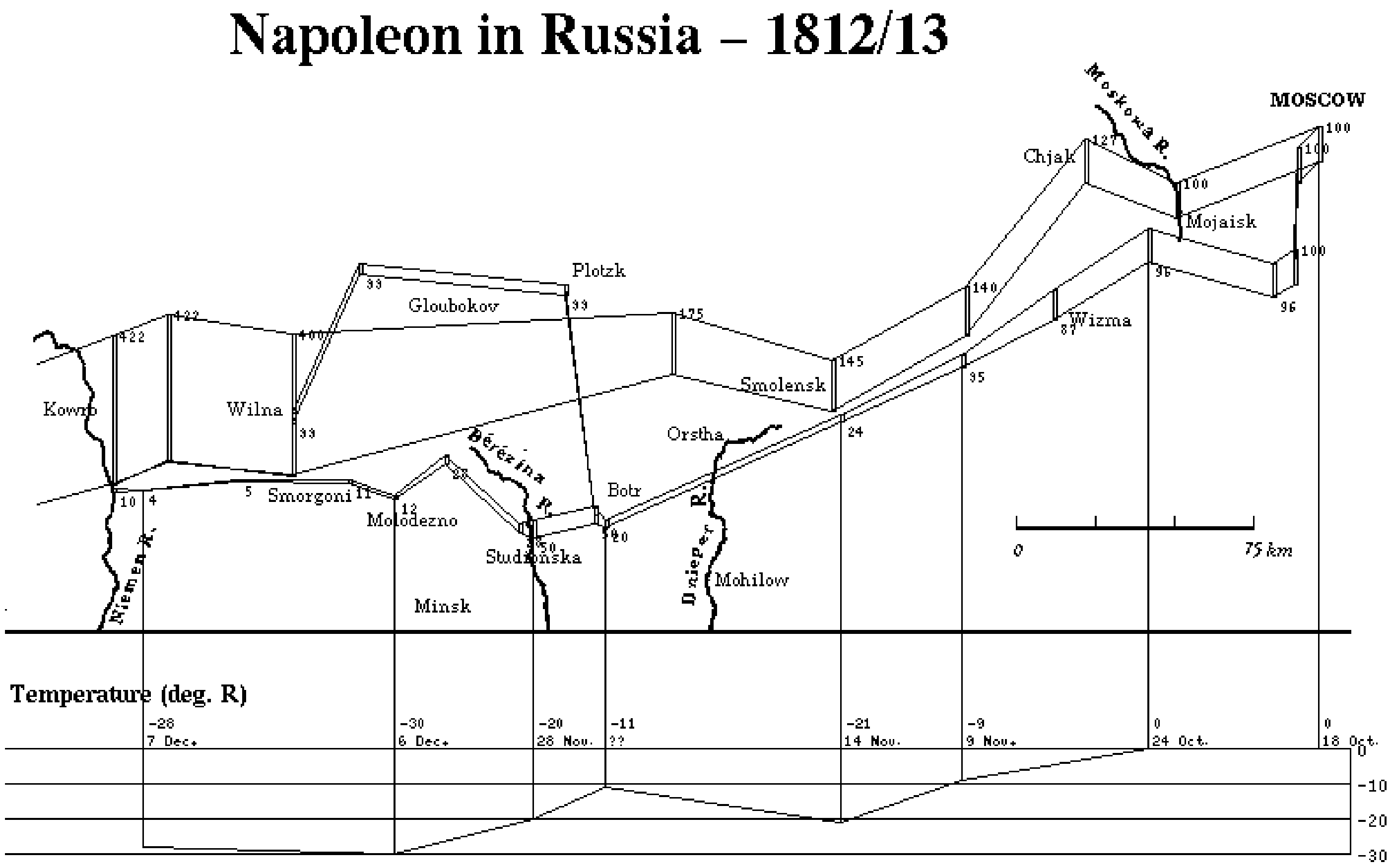

Figure 1.2: A graphic drawn by a Pluto specification

programming systems with small application domains (for example, [62]) or have offered a small number of (fixed) display methods for data [45, 48, 63]. While many of them are quite effective in their individual domains—program visualization/animation [9, 49, 63], program data structure display [45], bitmap displays [48]—the specialized techniques they provide typically cannot be directly used in some other application. We often need displays that go beyond the canned techniques provided by such tools. Expert users (end-users) in scientific domains often have very specific display requirements, ones that are not (or cannot be) provided by these programs. Since the mapping from the application data to the display may be arbitrarily complex, we need to provide them with the ability to implement the displays themselves. Unfortunately, most scientists have no programming expertise, and cannot implement these displays, while the programmers writing display systems do not have the domain-specific information required to design them. We present a system (Pluto) that provides a graphical specification editor to the user. The user creates or modifies a specification using it, and then invokes an interpreter called ppc (either directly or via a Pluto command) to create the display. This system attempts to bridge this gap by putting much more of the power to specify custom displays into the hands of the end-user.

Our approach will allow sophisticated and general visualizations to be constructed and modified without conventional programming. For example, in Fig. 1.2 we present a display created by this system of the same data as the chart presented earlier, drawn by Minard, with a slight difference—instead of using a thick line to draw the path, we use two lines so as not to obscure underlying details of the map. (Since in this display we have approximated line width by vertical displacement, there are anomalies at the sections where the lines are close to vertical.)

In addition to end-users, a general-purpose visualization/interaction tool like the one presented here can also help the experienced programmer. Very often, writing a graphical display program using conventional toolkits and window systems is a tedious process, since a lot of effort has to be made towards book-keeping, layout and placement of objects and other details. The techniques we present allow programmers to concentrate instead on the display and interface, allowing rapid prototyping and easy fine-tuning.

We now present some of the techniques that have been used for specifications and the requirements for an effective notation that end-users can use. We also look at the way these are handled in this system, Pluto.

1.1 Text based languages

To date, the most common method of specifying a computation to a computer-based system has been with text. Textual languages, like those used for conventional programming languages, have many advantages: they are compact and very expressive; there is a large amount of prior knowledge about their use and their power; their semantics are well-understood, and they have very efficient translation/execution models.

However, they also have a number of drawbacks, especially where end-users and novices are concerned. They can be compact to the point of being cryptic. For expert users, this is not a drawback, as they carry the semantic context required to use such a system; indeed, for expert users, the total number of input tokens required to complete a task often determines the effectiveness of the notation [12]. However this kind of interaction places a heavy burden of memorization on novice users. These languages also tend to have a highly structured syntax that requires the memorization of many details. Often, instead of having a semantic basis, these details and the syntax are an artifact of the language and the translation process. As an example, most conventional programming languages offer expressions that involve operators with different precedences; this concept is often hard for beginners to grasp. As a result, there is a steep learning curve, so extensive training is required before users can be productive with the language [40, 33].

Textual languages also have another drawback where specifications for visual systems are concerned. While their linear form of a sequence of statements is well suited to conventional sequential programs and offers a good abstraction for the flow of control, it is not suited to typical visual domains. Visual systems usually have no need for sequential steps and are better thought of in terms of a set of objects that have to be placed on a display without any artificial sequencing of operations.

To summarise, it is likely that end-users are the most qualified to describe the displays desired, and we need a method that they can use. They usually have no inherent interest in the computer system as such, but want to concentrate on the task at hand. The specification methods should thus be easy to learn without the need to absorb many new concepts; this precludes the use of a conventional textual language. To describe a graphical layout, a *visual language*—a specification system that uses graphical elements arranged on a two-dimensional space—is much better than a textual one: the language corresponds much more closely to the objects being described. This is borne out by the fact that for specifying user interfaces, textual languages are being superseded by visual systems. These factors combine to make a visual language the best choice for the input to the system.

1.2 WYSIWYG Systems—Advantages and Limitations

Visual specifications have many advantages where end-users are concerned. Well-designed visual systems are easily learned and can be used effectively by non-computer-literate users. The success of the direct manipulation interfaces—like the "desktop" first introduced with the Xerox Star [60] (and popularized by the Apple Macintosh)—is a good example of this.

WYSIWYG (What You See Is What You Get) systems are one example of direct manipulation interfaces. A direct manipulation system is one that allows users to work directly on graphical representations of objects on the screen. Actions are typically specified with the mouse on an image of the object rather than with a command language. A WYSIWYG system extends this concept to displaying, and allowing the user to change the final result of the specification.

One reason direct manipulation interfaces [31, 57] are so easy for novices to use is that representations of all objects of interest, the objects that can be manipulated, are always visible. Operations on these objects are carried out by operations with the mouse on their graphical representation. These commands rely on the user's model of the physical world, instead of syntax that has to be learned—for instance, a file on the system can be moved to a different folder by picking it up with the mouse and dragging it over into the folder.

Complex operations can also be described incrementally. The effect of each such change is displayed, so intermediate states of the objects are visible. Furthermore, in a well-designed direct manipulation interface, these incremental actions should be reversible so that any mis-step taken by the user can be immediately rectified [31, 56].

When experienced users interact with a system, they do not always think in terms of commands being described that are then carried out by the system; instead, they think in terms of performing actions directly on the objects of interest. For example, an experienced user using the Unix shell to compile all C language files in a particular area types a command like `cc -c *.c` without consciously having to remember that to indicate the C compiler to the shell, `cc` should be typed, or that to indicate all C files the shell expression `*.c` should be used; these processes become integrated into the user's mental model of interaction with the computer. In a well-designed direct manipulation interface, novices can also have a similar ability: the actions seem to occur at the same level that the objects are at. They do not have to break down the task into different complex sequences of actions but can rely on their mental model of the physical world to suggest ways of doing things.

WYSIWYG systems exploit all these advantages of direct manipulation interfaces, especially visibility. They are quite successful in some applications, like interactive drawing programs and word processors.

However, direct manipulation interfaces do have some drawbacks. There are some objects that cannot have a direct visual representation, like objects that have not yet been created. Another problem is the specification of sets of objects that are chosen according to some rule, instead of being directly identified. Considering the Macintosh desktop example again, there is no easy way to specify that all documents whose names start with the letter 'n' are to be removed. It is also hard to specify a deferred action, one that will act on an object at some point in the future. In contrast, the Unix shell user can use the same naming scheme—a text string—both for objects that are currently available and for objects that will exist when the deferred action is performed. Rules for describing sets are available with abstractions like wildcards.

Most WYSIWYG systems also have certain other shortcomings. Often these systems do not in fact show all the components (and the relationships between them) that affect the behaviour of visible objects. A specification system needs to present a complete picture of the behaviour, not just the appearance of the final product. For example, many WYSIWYG drawing tools have the facility to group objects together into a composite that then behaves like one object; however, on the drawing canvas, the appearance of these objects does not usually change to reflect this. (Many popular drawing programs do indicate this, usually by putting some sort of border around the grouped items; however, this is not a WYSIWYG construct, since these grouping indicators do not appear in the final product.)

WYSIWYG systems also do not represent abstractions very well. For example, some drawing programs allow the user to replicate objects so that if any object is changed, all instances of it reflect the change. This is a very useful feature to have. For instance, a change to a subsystem of a technical plan or blueprint does not have to be repeated for each place it is used in the drawing, but just making one change will automatically update all copies. Not only is this convenient in that it reduces the drudgery, it also cuts down on the likelihood of errors being introduced into the plan. However, in most WYSIWYG systems these objects appear to be independent; there is no visual representation of the relationship.

In the domain of data display, we have a similar need for abstraction—for example, we need to be able to specify displays with a variable number of elements. We would like to be able to use the same display for different sizes of data sets, and not have to change the specification each time. We would also like to be able to use conditional elements, objects that depend on some property of the data. This represents a case of a deferred action, one that will be executed when the data are actually displayed.

We need to provide a system that exploits the advantages of WYSIWYG systems, but offers alternative strategies to handle their drawbacks.

1.3 Requirements for a flexible system

We have seen two methods of specification, textual and visual. Both have advantages and disadvantages. In this section, we look at some of the desirable properties for a flexible system, one that is easy to use and extend, in more detail.

Usability. The system needs to be usable by end-users. As has been related above, direct manipulation systems make it easy for end-users to get started with minimal instruction. We need to use visibility, the strong point of this approach, in any system targeted towards end-users. In the application domain of data-visualization, the strong correspondence between the visual language and the objects being described make direct manipulation a natural choice.

Power and Expressibility. Ease of use should not, however, be achieved at the expense of expressive power. If only a small number of displays can be described by the system, it will be of use to only a few users. In particular, we need to be able to allow for data sets of variable size—the system should be capable of handling objects that are repeated as many times as necessary. The system also needs to be able to handle conditional elements—the users should be able to describe a display that depends either on the data or on user inputs. It should be able to integrate user input into the display, so that an exploratory approach to understanding the data can be used.

Scalability. We also need the notation to be scalable. Many graphical systems suffer from not being able to describe large and complex specifications. As compared to textual languages, visual languages use up a large amount of screen area, which can cause undue emphasis to be placed on minor details of bookkeeping etc. There are two major approaches to handling this problem:

Scrolling views—the system provides a magnified view of part of the specification, optionally together with an overview of the entire program. The overview may be integrated with the magnified view or distinct from it. The magnified area can be moved around under user control, usually with a slider arrangement. Examples of this technique include scrolling windows, where a moveable window displays part of the program in detail while another shows the overview, and integrated displays like fisheye views or the perspective wall (described in Chapter 2). This kind of display is very effective in showing part of a large specification, and allows the user to concentrate on just one part of the spec. However, the area to be drawn in detail is usually selected based on the physical layout of the specification, which may be an artifact; the view provided may not correspond with the organization of the program, or with the user's mental model of it.

Hierarchical organization—the entire program is organized hierarchically by the user based on functionality. The system can then display in detail part of the hierarchy, while showing an overview of the program in a different view. This method has the advantage that the magnified views are selected based on distinctions the user has specified, rather than depending on physical layout. In most cases, this is the better method, since the detail views are chosen based on the functional units of the spcification as designed by the user.

Re-use. The system needs to allow the re-use of commonly used components. Users can be broadly divided into two categories, tinkerers and workers [40]. The workers do not have any interest in learning about the system, but just want to be able to use it. The tinkerers explore the system and can effectively tailor and modify it, adding useful techniques. The effectiveness of any system can be vastly improved if the tinkerers develop new idioms of usage and then share them with the workers. The specification system should be well separated from the application data for effective re-use [3].

A hierarchical organization is well-suited to this task. The division of the specification into pieces based on function provides a natural place to include other functional units. Objects in the hierarchy can be stored in, or used from, a shared library.

1.4 Pluto

We have presented two major specification paradigms and also the major concerns for a flexible system. We now look at how we address these concerns in this system, Pluto.

Pluto presents techniques and a notation for users to specify displays easily and pictorially. It provides support for abstractions that overcome the drawbacks of WYSIWYG systems, while retaining their strong points. The objects in the specification are abstract representations of the objects that will appear on the display. The positions of the abstract representations on the specification do not force the layout of the final display, which allows the description to be organized with regard to its function. The abstractions also support the description of objects that appear conditionally in the display, as well as objects that need to be repeated depending on the amount of data.

Display specifications in this system have a hierarchical organization, so that minor details can be encapsulated and hidden from immediate view. The advantages of top-down structure in conventional programs are well-known; the same advantages can be gained from a hierarchical visual language. The re-use of pre-defined specifications is supported by allowing nodes in the hierarchy to come from a library. This allows more advanced users to help novices and makes the system more usable.

Two views of the specification are provided, a full detail view and an overview. In addition, the realized display can also be brought up (even on incomplete specifications) providing, in effect, another view.

The system also allows interactors or widgets (graphical objects that encapsulate appearance and behaviour [43]) to be included in displays. The input from these interactors can be used to modify the display dynamically. The interactive displays that can be created encourage an exploratory approach to the presentation and understanding of the data.

Traditionally, visual programming systems—ones accepting a visual language as input—have tried to emulate conventional programming languages in the utilities offered to the user (for example, see [52]). Since conventional programs have complex control structures, they are hard to model naturally in pictures in a way that naïve users can understand. In the data display domain, though, the objects being specified are graphical, so there is a natural correspondence between the specification language and the objects being described. This makes it easier for end-users to design and understand.

Furthermore, the control structures in the system we describe are simpler, as they don't have to be completely generalized. The "repeat" and "case" constructs are the only control structures involved, and they map directly onto graphical objects.

One of the major stumbling blocks for students just learning programming is the syntax involved. Often, these syntactic details are in fact artifacts of the language translation and do not encode any semantic information. This syntax leads to a steep learning curve, and an extensive training period before the users can undertake useful tasks. This difficulty is avoided using a visual system with a direct manipulation interface, so that the users can start working with much less instruction.

1.5 Contributions

This work presents a new approach to the exploratory data-display task. Traditionally, non-programmers have been forced to select from a fixed set of pre-defined display methods, or else learn to program in conventional programming languages to implement new data display techniques or to modify pre-existing ones. The main contributions of this work may be summarized as follows:

- it puts a new capability in the hands of a wider group of users. End-users will not be forced to rely on pre-existing display techniques, or to learn programming to implement displays.

- it presents a new approach to visual programming by restricting the domain to data visualization. The correspondence between the visual language and the display being specified overcomes some of the disadvantages that general visual programming systems have.

- it provides a notation that supports a set of abstractions for expressing concepts like repetition and conditionals. The notation provides a level of abstraction needed to implement more general types of displays than WYSIWYG systems.

We present a complete system that embodies the concepts of these contributions. The graphical specification editor exploits the advantages of direct manipulation, allowing occasional users and non-programmers to learn to use the system quickly.

It is hoped that with the capability of data display design and implementation begin placed at the disposal of a wider group of users, new and innovative display techniques can be developed.

1.6 Dissertation Outline

In the rest of this dissertation, we discuss the data visualization problem in more detail. Chapter 2 presents a look at some related work in the field. In Chapter 3 we discuss the display framework and provide an overview of the system.

Chapter 4 presents the basic elements that displays are built up from. It also describes the notation for describing relationships between them, one-way constraints. Chapter 5 presents advanced features of the notation that implement the control structures in the system. It also describes the other part of the data visualization task, data selection. While the display techniques will work with most data selections paradigms, we present a simple selection tool to test the display specification system.

Chapter 6 describes the higher level operators that put together pieces of the display to form larger displays. These *composition operators* do not provide any additional power over the objects described in earlier chapters; instead, they provide an overview of the specification and a slightly different way of describing displays.

In Chapter 7 we present examples of displays created with Pluto, and the specifications that created them. We show that most types of displays can be created quite easily with Pluto.

Chapter 8 describes the experiences of users with this system. Trials were carried out with users of varying amounts of computer expertise, from novices to experienced window-systems programmers.

Chapter 9 presents some concluding remarks about the system. We discuss the strengths and the weaknesses of the notation, and also the implementation status.

Some features of the current implementation of the system that were not covered in the main body of the dissertation and are presented in Appendix A. These include details of the interaction semantics and some details of the interactors provided.

Chapter 2
Related Work

In the design of this system, we have built upon the contributions of many others. Data display techniques are a rich body of work, with some of it dating from almost 200 years ago. The field of visual programming is also extensive, and we have used those experiences in the design of our system. In particular, the visual specifications of user interfaces are closely related to this work; some of these systems have also incorporated features for user-tailorable data displays. Although none of this work has explicitly addressed the issues of visual programming for exploratory data display, they illustrate many of the paradigms of the field. This chapter presents some of this related work.

2.1 Data Display Techniques

To make an effective display of information, one that shows trends and relationships in the data well, it is necessary to choose the right form of display. Over these last 200 years, many good display techniques have been developed that make good use of human perceptual abilities. The earliest charts were cartographic, and consisted of some data overlaid on a map. The direct physical interpretation for the data made these charts easy to comprehend. The first instances of charts representing purely abstract quantities were developed by W. Playfair (1759–1823) and J. H. Lambert (1728–1777). An extensive account of this early work can be found in [67]. Here we present some representative display techniques.

One of the earliest uses of visual representation of numeric data was presented in Chapter 1 (Fig. 1.1). This chart was drawn in 1885 by C. J. Minard. It is a display of the progress of the French Army under Napoleon in the campaign on Russia in the winter of late 1812. It integrates the display of the position of the army with its strength, and combines this map with a graph of the temperature, providing a unified display of 5 variables [42]. This chart is also discussed in [67].

J. W. Tukey explored many of the techniques for data display, particularly in the field of statistical data. He developed the box plots (and many variations on it—Fig. 2.1) that represent multiple values of statistical experiments, windowed scatterplots of multi-dimensional data, and many other presentations of multi-dimensional data. Many of the current techniques of scientific visualization date from ideas proposed by him [69, 70, 71].

E. R. Tufte presents many techniques for the display of quantitative data [67, 68]. One example is the "rug-plot" (Fig. 2.2) which plots pairwise correlations of multi-dimensional data. With the addition of marginal distributions—the distribution of just one attribute—linking together the scatterplots, this is a valuable tool and gives a good impression of the relationships in the data. Each multi-dimensional data-point can be followed from scatterplot to scatterplot by means of the marginal distributions (the dotted line in Fig. 2.2). This gives an intuitive flavor for the nature of the display.

Another display technique presented is the "train-schedule" chart (Fig. 2.3), which plots the

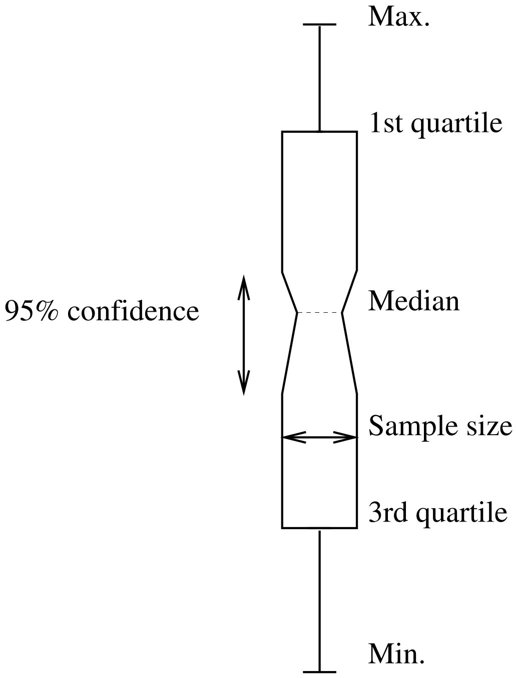

Figure 2.1: A notched box-plot

progress of trains on a time/distance chart. This kind of chart shows the speed of individual trains—the steeper the line, the faster the train—and is a valuable tool for arranging schedules and working out conflicts on single-track sections. It is used by many train systems around the world.

A number of general-purpose data display systems have implemented these techniques. However, they tend to limit themselves to a fixed set of display methods that cannot be readily combined or extended. We need a flexible way of specifying displays that will allow users to develop more such displays and also experiment with them. Pluto specifications for these displays are given in later chapters.

2.2 Computerized Data-Display Systems

There have been many systems that offer display techniques tailored to particular applications. In their application domains they generate very effective displays, and are very helpful data analysis tools. The biggest advantage of computerized methods is that the very powerful techniques of animation and interaction can be used. Their main drawback is that they tend to be dependent on the base application and cannot be easily generalized or re-used.

Two systems that use interaction and animation very effectively are the Cone Tree and the Perspective Wall. The Cone Tree (Fig. 2.4) is a technique to visualize large hierarchical data structures. It represents each node of the hierarchy by a three dimensional view of a cone in space. The children of each node are drawn as slightly smaller cones spread evenly about the perimeter of the base. The size of each cone depends on the number and size of its children; thus each node

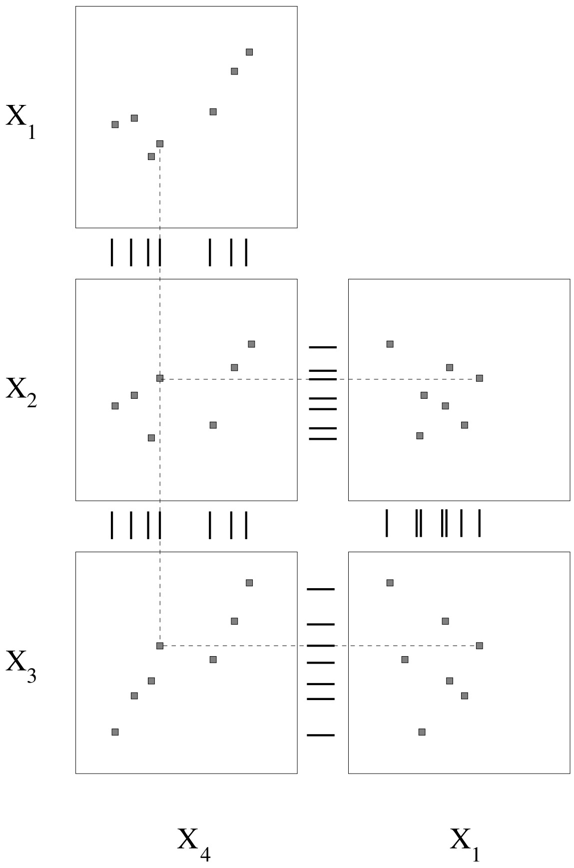

Figure 2.2: A rug plot

is drawn larger than its children. To display relationships between objects, the system can animate movement of the cones, so that the interesting object is brought to the front of the display smoothly [54].

Another innovative display technique is the Perspective Wall (Fig. 2.5). It integrates an enlarged, detailed view of a data display with an overall view for context. The data are represented as being displayed on a wall in front of the user, with the interesting components right in front, and drawn in detail. Further away from the center, the system uses perspective projection to smoothly reduce the size of the objects. The two dimensional view of the data is thus folded into a three dimensional object, the ends of the display away from the user. The distorted view of the data not currently the focus of attention is still enough to provide context. The user can move from place to place in the display, and the system smoothly scrolls the display [39].

These two techniques derive their utility to a great extent from the 3-D modelling, and the impression that the user is interacting directly with the objects. Our emphasis in this work is on flat (2-D) displays so such techniques will not be considered. However these techniques prove that the ability to let the users interact with the displays is very powerful; any general purpose data

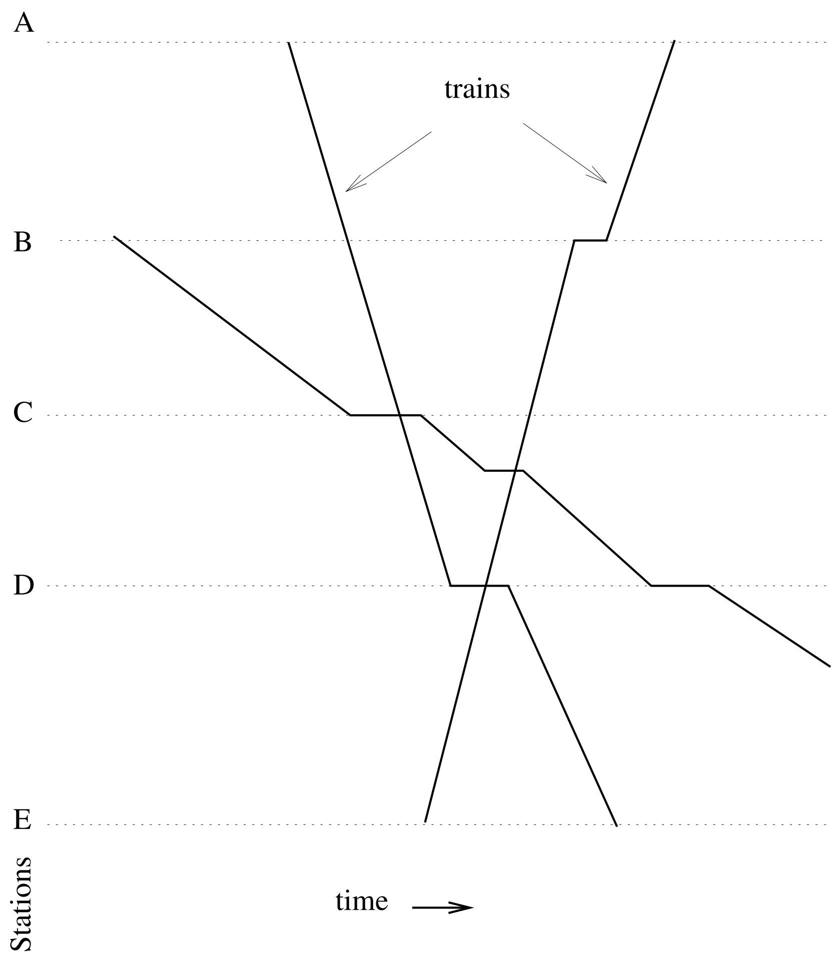

Figure 2.3: A train schedule chart

display system needs to support it. Furthermore, rather than pre-defined displays, there is a need for user-defined displays to be able to incorporate user-input.

2.2.1 General Purpose Data Display Systems

There are many systems that draw simple displays of numeric data. The utility *gnuplot* is one such system [73]. It can perform mathematical operations on data sets and then display them using a number of different techniques, like scatterplots or bar-charts. It is a very useful tool for data analysis, and many common display paradigms can be implemented. However, there is no way for an end-user to extend these styles. If a slight modification of a standard display is required—for example, if a histogram is to be plotted such that the tops of the bars are to be aligned with a reference curve, so that errors from the reference are visible at the base—this system cannot be used. New types of displays—for instance, Tukey's notched box-plot—cannot be included in the system without modifying the source code. Another display system is implemented by

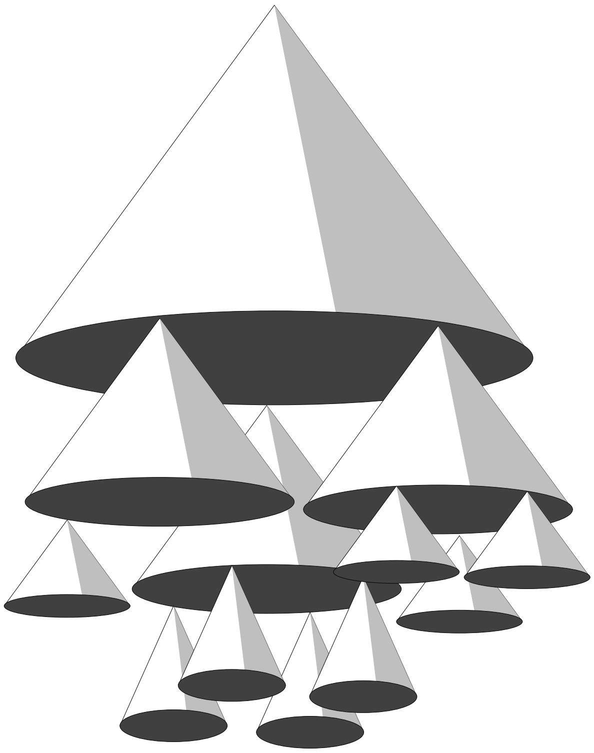

Figure 2.4: Cone Trees: A technique for the display of hierarchical data

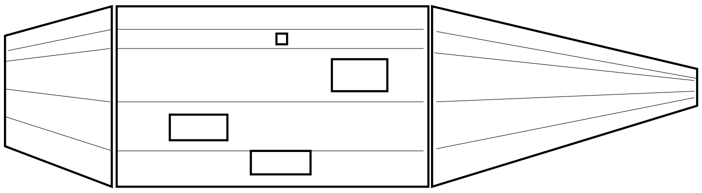

Figure 2.5: The Perspective Wall: Presenting context and detail for 1-D data

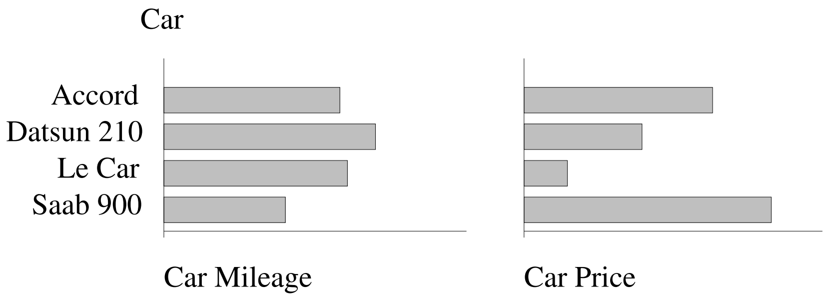

Figure 2.6: An APT "one-axis" composition

S, the statistical analysis package [4]. It offers many of similar features to gnuplot, including many standard data display techniques. However, it shares the same disadvantage as gnuplot—the built-in display techniques, though extensive, are finite. Small variations of pre-existing displays cannot easily be implemented; and there is no way to try out an innovative new display technique. However *S* does offer much more support for analysis than gnuplot or other graphical packages.

2.2.2 Automatic Display Generation Systems

Several systems have attempted to automatically generate displays without direct input from the user. For example, Incense [45] is a system to automatically create visualizations of program data structures. One of its most important contributions is its concept of *artist*, an object that is responsible for the display of program data structures. A special type of artist is a *layout*, which draws box-and-arrow diagrams for pointers, tables for structures etc. and tries to tailor its displays to the amount of space available.

Another automatic display system is APT [38]. It automates the display of relational information in an application independent manner. It provides support for many of the common relational display techniques like bar charts, scatter plots, pie charts etc. One of the important contributions of this work is that presentations are considered sentences of a graphical language, so that semantic aspects of the system can be rigorously encoded. Automatic *effectiveness* and *expressiveness* criteria are applied to the language that can be used to pick a display method for the type of display required.

APT also provides a simple algebra that can compose the languages created. For example, two graphs that represent two properties of a set of objects—say bar-charts that encoded price and mileage for a set of cars—can be composed together so that they would be drawn next to each other sharing the common attribute, the cars' names (Fig. 2.6). This is an example of a *single-axis* composition; a *double-axis* would be one where both axes are common, and is equivalent to an overlay.

The compositions provided by APT link together two presentations keeping in mind the semantics of the charts. Only charts that share one or both axes can be composed; in general, users should be able to compose displays independent of the structure of the objects being composed.

The emphasis of both these systems is on on automatic generation of displays with a minimum of user input. This technique is useful in situations like debuggers where specific display types are to be used, and releases the user from the effort of specifying the display. However this is not the best approach in a general situation. When analysing large amounts of data, the users need more control over the display. In general, we need to provide the user with complete control, without any restrictions about the objects that can be composed. At the same time, the automatic generation of displays is very convenient; any flexible data display system should incorporate means for re-use of commonly used techniques and idioms.

2.2.3 Visual Environments

The Grape system [48] implements a visual environment for manipulating bitmap images, typically generated from a ray-tracing graphics program. The user can describe a pipeline of operators to manipulate these displays by various transformations. The sequence of operators is quite similar to the operator tree hierarchy provided by Pluto, although they are not general-purpose compositions. The images manipulated by this system come from the application program and are specified in a conventional programming language. However, there is a need for more basic support: instead of offering an image manipulation environment, a complete display specification system is needed.

2.2.4 Program Display and Animation

Program display is a very rich field of research, utilising many of the display paradigms already presented. Perhaps the best-known program animation system is Balsa [9]. This is a system that generates very high quality animations of programs as they execute. It is intended to be a teaching tool, so that students can see algorithms in action. Programmers annotate their programs with *interesting events* which drive the animation. The visual presentations of the animations themselves have to be implemented by a programmer; furthermore, the program has to be executed inside the Balsa environment—arbitrary programs cannot be animated. Since considerable programming is required, this system is not suitable for end-users. Nevertheless this system generates high quality animations, including many different types of displays for different kinds of program data structures.

Tango [63] is another program display system, one that takes a slightly different approach from Balsa—it is designed to be easier for the programmer to use. It also uses an annotated program, with significant events driving the animation; however, the programming effort required to implement animations has been considerably reduced, at the expense of highly-polished final displays like Balsa's.

The displays that Tango provides are specified with Dance[64], a direct manipulation style graphical editor to specify paths and objects for animations. A graphical editor is provided that allows users to place objects and then specify movement by pointing at intermediate positions. This direct-manipulation style specification of the temporal aspect of Dance is one of its most innovative features. The editor provided by Dance is a closer to an enhanced WYSIWYG system than a direct-manipulation system with support for abstractions.

Both these animation systems have layout components that define the appearance of the animation. Balsa requires the user to specify the display in a conventional programming language, which is distinct from the WYSIWYG-style specification of Dance. We need to support a specification system between these extremes, one that is easier for non-programmers to use than conventional programming languages, but is more powerful than a WYSIWYG system.

Novis [49] and Tapestry [41] are two other program visualization systems that animate and visualize the execution of processor arrays. Both these systems animate programs from a higher level than most program animation systems—instead of showing low-level details of program execution, they show the activity between processors, providing a display of messages as they are exchanged. Novis, for example, simulates a set of basic processor elements and I/O queues arranged in some user defined topology. Messages and state information are encoded with color, and the system also provides some other features like showing wavefronts and detecting deadlocks.

Both these systems provide very useful visualizations. If more users had the ability to create displays like these, there would be many more such high-quality displays. We need to put this ability within reach of the end-users.

2.3 Graphical Languages and Specifications

The graphical notation we describe draws from earlier experiences in the visual language field. Most of the earlier systems attempted to provide a general-purpose graphical language, which have not been very successful, as they all suffer to some degree from the problem of scale—programs of usable complexity are hard to manage. The successful visual programming systems have been in specialized domains like designing visual layouts, where the advantage—a close correspondence to the application—overcomes the problems, like the large demands on screen space. WYSIWYG drawing programs are an excellent example of this type of system.

2.3.1 General Purpose Visual Languages

Many systems have attempted to provide a completely general-purpose visual programming language. Most of these systems suffer from the fact that large programs are very difficult to specify (scalability), and too much emphasis ends up being placed on small details and book-keeping. A representative example is GIL [52]. This is a system that helps students learn LISP by providing them with a visual environment and a structured visual editor. One goal of this system was to develop an appropriate mental model of program execution, both by using the familiar visual approach and with interactive help. It augments the graphical editor with a goal-based tutor. This tutor provides samples of input and output, and the student tries to complete the program from either end. This system can be useful for teaching the basics of programming languages by providing a visual metaphor. In particular, the goal-based tutor is a very useful accessory to the graphical display of the program. However, it is not a usable tool for programs larger than the equivalent of a few lines of code—larger programs do not fit on the screen well.

2.3.2 Data-flow Systems

InterCONS, the Interface Construction Set [62] provides a visual data-flow programming environment. Inputs to the program can come from a variety of input devices—locators, buttons etc. which allow some interesting programs to be described. However, it shares the drawbacks of many such systems—lack of scalability. Only small, simple programs can be described. It also suffers from the drawbacks of data-flow systems—expressions can be handled well, but more complicated control structures are harder to visualize.

InterCONS illustrates one of the earlier uses of interaction elements in a visual display and allows the implementation of some forms of exploratory data displays. The layout of the various elements is done by direct manipulation: after all the data-flow relations have been specified, the data flow arcs and any other computational devices can be hidden. The remaining objects form the display, and can be moved around to improve the appearance. This allows the appearance of the application to be easily specified by the user. However, it has the failing of WYSIWYG systems that the display cannot depend on the data. Other data-flow visual programming systems have also been developed, but they also suffer from similar problems of scale and expressibility; for example Pecan [53] and Programming in Pictures [51].

2.3.3 Composite Data-flow Systems

ConMan [20] provided a workaround to the problem of scale in data-flow systems. Instead of manipulating objects at the lowest levels (integers etc.) it worked at a larger granularity (3-D images) allowing the specification of many useful applications. The environment extends the pipeline metaphor of the UNIX shells to graphics pipelines. It allows graphical operators—in this case, that manipulate 3D images, like "shade"—to be connected in a directed acyclic graph. The images being composed are not created by ConMan but are provided by the user. The parameters to the operators are specified by attribute editors that provide a supplementary textual interface. ConMan proved the feasibility of many visual specification techniques, particularly the ability to hide parts of the display and working at a larger granularity than conventional data-flow systems. It can thus be said to be at a higher level of abstraction than data-flow systems.

Data flow systems like the ones discussed above were among the first uses of graphical specifications of programs. Instead of implementing a general-purpose system, we need to extend these concepts to graphical displays so that rather than describing the flow of data values, we can specify attributes in the system that depend on others.

2.4 User Interface Design Systems

The design of Pluto's specification system also draws upon experiences with user interface design systems. Several techniques have been described for describing the layout component of user interfaces. Some of these systems have used visual specifications, leading to some overlap between this area and the two areas outlined before. In particular, some user interface systems (discussed above) offer specification facilities close to visual programming (like Tango), and some offer interface support for automatic data display systems (like Incense and APT).

Syngraph [50] is an early system that allowed user interface to be specified by a textual notation similar to a context-free grammar in Backus-Naur form (BNF). Graphical primitives are terminals in the grammar, and included user-defined objects as well as pre-defined interactor objects. Actions are defined by rules with semantic meanings that are specified by Pascal statements attached to the non-terminals. Syngraph was among the first systems to use an abstract system for UI specifications. It freed the user from a lot of the interaction book-keeping, but the input to the system is not a trivial language—the user needs to be acquainted with context-free grammars and be able to describe the interface in that form. Also, the actual layout and placement of objects in the interface have to be specified with textual specifications. The abstraction that Syngraph offers is a very powerful technique for the description of displays. We need to provide this kind of power while offering an easier input model than a context-free grammar.

A system that uses a hybrid approach to specify an interface is vu [58, 59] which is part of the UofA* user interface management system. The interface is specified in a textual language that offered primitives to define various graphical input devices. These devices are automatically placed by one component of the UofA* system, but the user is allowed to modify the placement with a graphical editor implemented by vu. This approach is very useful when the specification is to be automatically generated and then modified by the user, as textual languages are easier to manipulate in a program, and abstract structures are easier to specify. For Pluto, we decided to give users the ability to completely specify the display to explore the limits of a purely graphical system. Using a hybrid approach would mean a greater language burden on the end-users.

Visual specification of User Interfaces dates from some of the earliest User Interface Management System (UIMS) work [11]. Other UIMS specification systems developed many of the layout paradigms of the field by proposing direct manipulation graphical editors to define interfaces [24]. One example is the Trillium system [21] which described an interface as a set of *frames*, each of which represents a state of the interface. Each frame was described by placing *items*, graphical interactors, directly on the drawing area. The interface can be tried out at any time—it emphasized a “try it” attitude and fast prototyping for an incremental, experimental approach to interface design. Any specification system should offer such a facility.

An alternative to the direct specification style—either a direct manipulation system or a textual system as the ones described above—is the demonstration and inference method, where the system infers properties of the specification without a complete and explicit description. An example is Peridot [46], a system for the specification of interfaces by demonstration. First, the user defines the appearance of an interface with a drawing editor. Relations between graphical objects are inferred by the system—for example if it noticed that two objects had centers close to each other’s, it would ask the user if that was a defining relation. Then interactions are specified by changing the presentation. The system tries to infer the rule that drove the change and again, prompts the user. The user does not have to explicitly specify layout rules; however, sometimes the system infers unintended constraints—artifacts of the present state of the drawing—and requires user guidance. Also, more complex rules are sometimes required that cannot be inferred by any such system. These drawbacks indicate that a direct specification style is more appropriate for larger displays.

Some UIMSs have provided slightly different ways of providing direct manipulation displays of application data. Jacob presents a system to specify interactive interfaces by means of a state diagram [32]. The input to the system is a text description of the layout of objects and state transitions describing their behaviour. While this is a useful way to think about interactive behaviour, even small interactions tend to have large state diagrams so it’s not a feasible system for large interfaces. It is also likely to be hard for end-users to grasp.

Pluto itself is an extension of a user-interface description system called Opus [29] that uses a visual specification—it specifies the layout of interactors on the screen and their behaviours. Pluto extends the notation provided by Opus by providing alternative views of the specification and by providing support for control structures. Opus relies on the application program for handling any external data sources. Some of the notational features of Opus (and hence of Pluto) were inspired by a dialog editor tool described in [13], which allowed for the appearance of an interface to change dynamically, by allowing distances between objects in the final display to be some proportion of other distances. We extend this notion to allow for the graphical specification of arbitrary expressions.

The notion of a *constraint*—a relation between values that is automatically maintained by the system—has been used by some systems. Thinglab [5, 6, 7] is a simulation system that offers many customization features, and allows the user to graphically set relationships between objects displayed on the screen. One problem of constraint systems is that it is very easy to over-constrain a specification so that no solution is possible. A later version of ThingLab extends the constraint notation to include constraint hierarchies, where constraints are given different priorities to resolve conflicts. This allows some constraints to be regarded as necessary and others to be optional. This constraint notation has been extended further by the Animus system [16] to include temporal constraints. This allows animated displays to be specified by describing relationships of various objects in response to events. In all these systems, constraints prove to be a powerful technique.

Using general constraints, users can specify systems that are over-constrained. While the constraint hierarchies allow one way of solving this, it requires that users learn about the semantics of the constraint systems. For end-user applications, a simpler system called *one-way constraints*—a system where an attribute can have at most one constraint defining it, and there are no cycles in the entire system—may be more appropriate, since they still allows us to create most types of displays, and by restricting an attribute to only one constraint we avoid the problem of overconstrained specifications. We feel that the advantages of the simpler semantics of one-way constraints outweigh their disadvantages.

One-way constraints have been used in the GARNET user interface development environment [44]. GARNET extends the kind of inference support found in Peridot by allowing inferred constraints based on user demonstration as well as by allowing the explicit creation of constraints.

The InterViews toolkit [37] offers a slightly different form of graphical composition than the ones provided in Grape. In InterViews, rectangular objects are composed together in *container* objects with variable sized filler called glue, like the boxes and glue model used by $\TeX$ [34]. Glue is a variable sized object that has a preferred, a maximum and a minimum size; it allows the description of layouts that maintain "reasonable" behaviour in spite of any size changes that may be enforced by the underlying window system. The boxes and containers are composed into a basic rows and columns configuration. Once InterViews objects have been composed together with glue, the user does not have to specify the response of the objects to size changes. In particular, in response to input, the interface can enforce a minimum or maximum size and maintain the internal layout. This behaviour is sufficient for many styles of layouts; however, there is a need to extend the simple row and column compositions to allow for more complex forms.

We have seen that although systems exist that offer some fixed number of displays, there aren't any that allow non-programmers to specify them. As described in the next chapter, we present a system that attempts to fill this need by providing a visual programming system that implements displays.

Chapter 3

Specification Framework and Overview

An important goal of this work is to offer a data display system that non programming end-users can employ that is still powerful enough for real-world problems. The system combines a direct manipulation approach with support for abstractions, so that deferred actions, i.e. actions that will be executed when the display is actually instantiated, may be specified. To be able to handle tasks of reasonable size, we offer a hierarchical two-view approach, which can overcome some of the problems of visual languages, the need for screen space and the difficulty of dealing with large programs.

3.1 A Display Framework

We have looked at many systems that offer various visual paradigms, both in specification and in data display. We have described the attributes that an effective display system needs to be both powerful enough to be usable and simple enough for end-users to learn.

The drawbacks of WYSIWYG methods for describing displays are many; yet most systems only offer some slight variation thereof. We need to be able to describe the objects of the display abstractly. Positions of the graphical elements in the specification should be assigned based on understandability of the specification itself. The two aspects—the specification and the final display—are in different domains, and have different requirements.

In any display of real size, there are often relationships between items that are implicit. In the data display, we do not want these to be visible, but in the specification, they play a very prominent role and need emphasis. These "working objects" need an effective representation that end-users can deal with.

We also need to encourage well-designed displays. Conventional programming languages have already proved the advantages of top-down design, information hiding, and type checking. The display should be similarly structured, divided into its component parts so that each part can be designed separately, and implemented and tested independently of the other parts.

At the same time, the user should have fine control over the layout of the display. Complete control over the details may be required when designing new display techniques, and as far as possible, the system should not hinder these efforts by imposing any restrictions. It should be possible for users to make mistakes, as long as these errors are easy to catch and to rectify.

These two needs—fine control over details and a broad division into component parts—are best handled by providing a two-view environment. One provides an overview that allows the display task to be divided into parts based on functionality; the other allows fine-grained control over the objects in the display.

Since the specification itself is an abstract representation of the display, there also needs to be a way for a user to be able to check intermediate stages. This will allow an experimental, incremental approach to display specification, with any mistakes being identified early. Ideally, a third type of view should display the realised display at any stage, with meaningful defaults provided for any unspecified components.

One of the major deficiencies of a graphical specification system is the amount of screen space taken up. Visual languages are inherently less space efficient than a textual system. This problem is made serious by the fact that most visual programming systems do not handle large programs very well. A hierarchical system deals with issues of scale quite well, while not giving up the advantages of a pictorial representation. The work presented here allows hierarchical specifications and simultaneously provides an overview of the hierarchy.

3.1.1 Semantic Framework

The data display problem has two central tasks—data selection and data visualization. The data sets in question are usually quite large; the user selects some interesting subset to display. The data selection process chooses the data to be displayed and sends it to the display component which creates a graphic of the selected data. (In this dissertation we will refer to the data to be displayed as a source or stream interchangeably.)

The basic operation in the system is graphical composition. For each data item from an input source, a small display is created. These smaller displays are put together to form the final display. The operations that put together these smaller displays to form larger ones are called *compositions*.

There are certain display types that do not fit this schema: any display that depends on a global strategy cannot be realised in this system. If the objects are to be composed in such a way as to optimise some global condition, we cannot use this method. It is known, however, that global optimizations of this sort are in general intractable. Examples of this nature include optimal graph layout, or arranging labels on a graph such that they do not overlap. These problems are best dealt with in a specialized pre-processor; the resulting layout can then be visualized with this tool. For example, a graph could be laid out using whichever layout algorithm is appropriate, and the resulting solution can then be displayed by Pluto.

Using Pluto to perform the final display has another benefit: the graph displayed (using the earlier example) can be interactive. That is, it can be displayed according to some layout algorithm, but then the user is allowed to drag the nodes and fine-tune the display.

3.2 An Overview of Pluto

We describe a system based on the requirements outlined above that allows users to easily describe interactive displays. Pluto is a system that implements techniques and a notation to specify displays easily and pictorially. It escapes the drawbacks of WYSIWYG systems by offering an abstraction of the display being created. It is a direct manipulation system that capitalizes on the strong points of such systems, while offering alternative strategies for their weaknesses.

- Since there is a direct correspondence between the elements of the specification and the displays to be specified, the notation is more natural than general-purpose visual programming systems. Visual objects and their attributes—like distances, sizes and positions—are specified by visual means. At the same time it overcomes the drawbacks of WYSIWYG systems by representing abstractions of the objects and by not restricting the placement and organization of these objects. The specification can be optimised for ease of design.

- While working on the specification, the user can, at any time, see the display as it would be implemented, even when the specification is incomplete. This is done by using the position of unconstrained attributes in the specification as a default value. The user merely has to select "Try it" from the menu, and the display appears almost instantaneously, effectively providing both an abstract representation and the realised version of the display being specified.

- The notation used is hierarchical. Minor details can be encapsulated and hidden so as not to clutter up the display. A hierarchical organization also encourages structured, top-down design of specifications, and alleviates the problems of scale.

- It supports the re-use of pre-defined objects, the use of libraries, in a natural manner. Commonly used idioms can be made available to users, and advanced users can develop useful components and techniques that the less proficient users can re-use.

- Smaller pieces of the display are combined through the use of composition operators organised into a tree. The visualization task can thus be divided into major components that are put together to form the final display.

- Two ways of interacting with the specification are provided. One is an overview of the entire specification and is provided by the composition operator tree. A more detailed set of *views* of the specification is provided by another component of the system. This allows the user to retain an impression of the overall structure while working on a smaller, more detailed view.

- Interaction elements can be included in the display, to encourage an exploratory approach to understanding the data. User input through the interactors can modify the presentation of the data, leading to a greater understanding of the relationships involved.

- The system allows the user to describe objects that are to be replicated a multiple number of times based on the amount of the data to be visualized. The repetition can also be performed some arbitrary number of times. This is the graphical analogue of a loop in a conventional programming language.

- Displays that depend on some condition can be described. The conditions can depend on the data stream or on user input. This is the graphical analogue of the 'if' statement.

3.3 System Architecture

Fig. 3.1 shows the architecture of the system. The user interacts with a program called Pluto, a graphical editor, to create or modify a specification. A program called ppc combines the specification for the display with the data and creates a textual script for a program called Penguims. Penguims handles all the drawing and interaction management of the final display. (Penguims and its language is described in Section 3.4). Ppc automatically invokes Penguims; the user does not have to handle Penguims code at any point.

There are three main components to a specification (Fig. 3.2):

1. The Query System

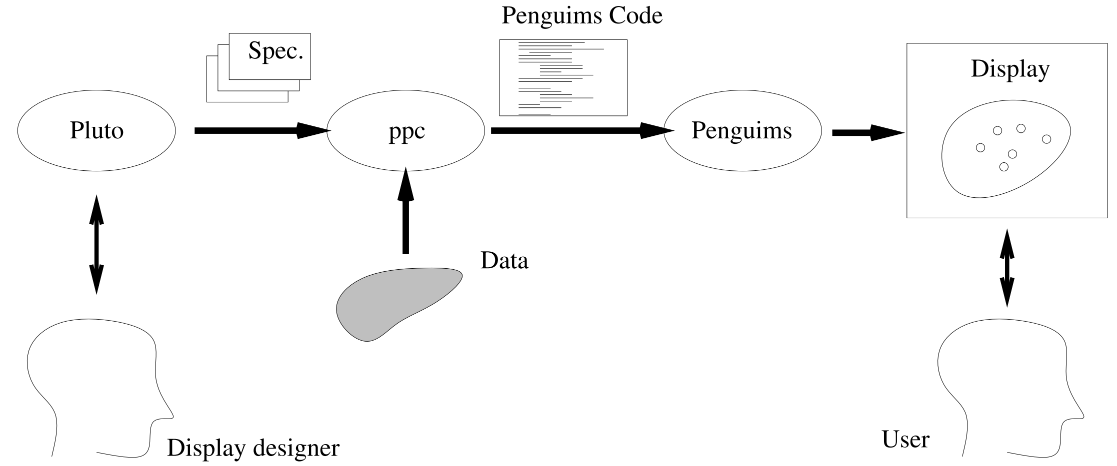

Figure 3.1: System Architecture

2. The Graphical Layout System
3. The Operator Hierarchy

Fig. 3.2 shows a data-flow diagram between components of the specification. Fig. 3.3 shows the system as it appears to the user.

Since the emphasis of this work is not on data management, the current system uses a relatively simple data manipulation system based on text files. This query system allows the user to specify the data source within the system and the way the interesting data are selected for display. For the data source, the user specifies the name of a file that the data is to be taken from. In most cases, the input is a sequence of a group of atomic types or *fields*—for instance, a table with records consisting of strings and integers—and the query system allows these fields to be named. The queries are formulated textually, and define a pattern or template. The data thus selected are available to be displayed by the rest of the specification.

The data selected by the query are fed to the *GLS*—Graphical Layout System, represented by rectangles in the middle section of Fig. 3.2. The GLS specification incorporates the data and addresses the detailed layout of the displays. In the example (Fig. 3.3) the window labelled "Main Window" shows the tree of composition operators. At the leaves of this tree are named boxes ("c1," "backdrop" etc.). Each of these boxes represents part of specification that is described with the GLS. Each such box is represented in detail in a different GLS *view* —a section of the specification described in its own window. The GLS views are shown in windows labelled "GLS." On the upper left of the figure we also see a query dialogue box—in this case it shows that the input comes from a file named "cars79.dat" and its first field has been named "car_name".

Each query is attached to a type of object in the GLS that handles repetition. When the specification is interpreted, these GLS objects and all their descendants are replicated multiple times to form the final display. Every replicated GLS object has available to it the data selected by the query. In effect, the query sets up a symbol table that is used to resolve references in the GLS specification. Since multiple queries may be used in a specification, each query creates a new

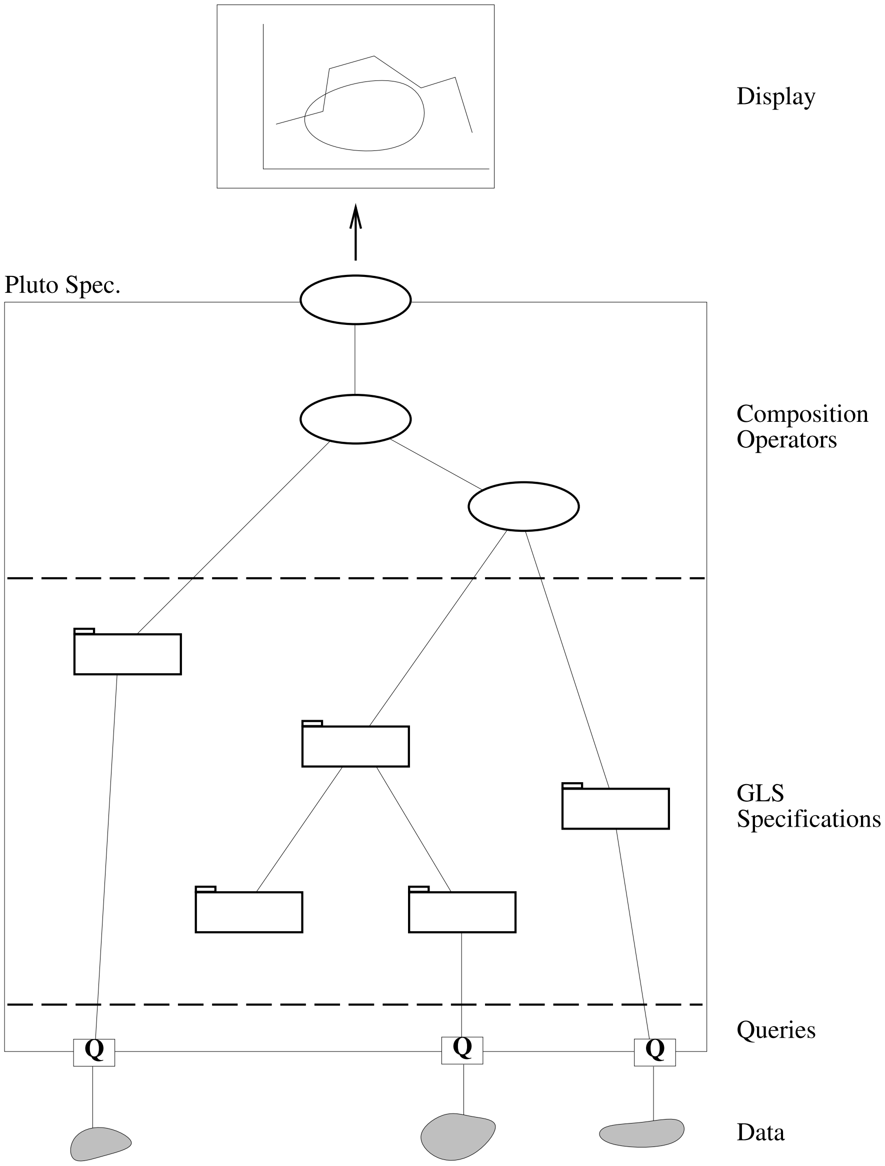

Figure 3.2: Components of a specification

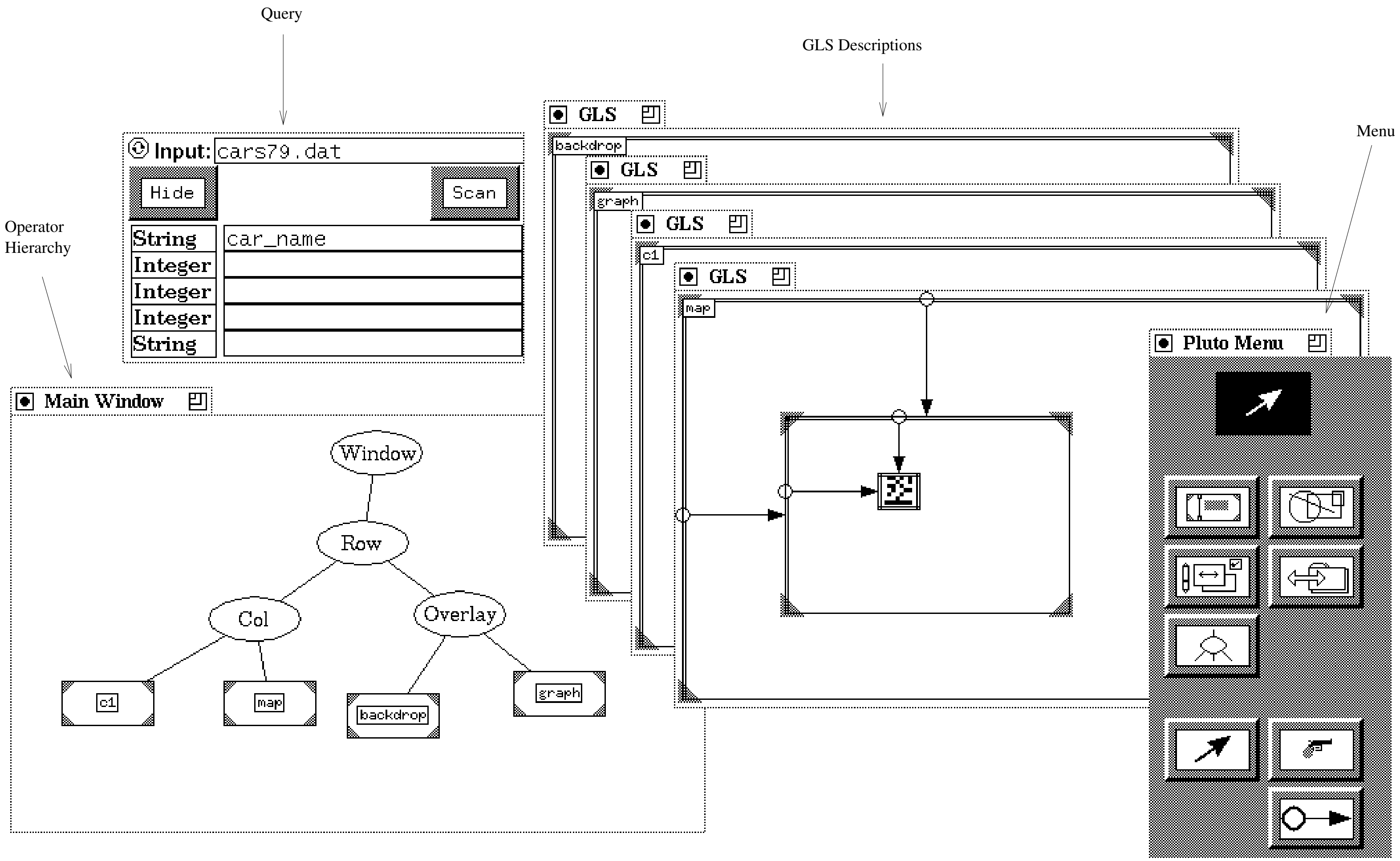

Figure 3.3: Overview of the system as seen by the user

scope, and normal scoping rules are followed.

The composition hierarchy is a tree of *composition operators*—operators that combine displays that have been specified by the GLS component, for instance to arrange them in a row. At the root of the tree is the window, representing the final result; the internal nodes are operations that compose one or more displays together, and the leaves are simple displays. The operators are drawn as ellipses in the display of the tree. Each operator has certain attributes that parametrise the layout of the images provided by its children. These compositions are similar to the compositions offered by the InterViews toolkit [37] in that a tiling model of composition is used, in a hierarchical rectilinear manner.

The user sees the operator hierarchy and each GLS view in a separate window (Fig. 3.3). The main menu is shown on the right with the grey background, and a query dialogue box is displayed just above the view of the operator hierarchy. In the figure, the data comes from a file named "cars79.dat" and its first field has been named "car_name."

3.4 Penguims

Pluto uses a general purpose interactive system called Penguims[28] for managing the interactor objects and the relationships between them. Like the NoPumpG system [36, 72] it is based loosely on a spreadsheet metaphor, with the basic element called the *cell*. As in a conventional spreadsheet, each cell has a value and optionally an equation; however, instead of being arranged in a rectangular pattern, Penguims cells are grouped into collections called *objects*. Each graphical object of a Pluto specification maps onto a Penguims object, and each attribute of a Pluto graphical object (like the boundaries, and others discussed further on) maps onto a cell in Penguims.

The value of each cell can be described by an expression using the values of other cells. If the value of any cell used in the equation of a cell changes, Penguims automatically re-evaluates the expression so that all cells stay up to date with respect to their defining equations. This automatic update mechanism provides the basic computational capability behind the *constraints* which form a central part of Pluto specifications. These will be discussed in detail in the next chapter.

Penguims interprets a textual language. The input consists of a set of named objects, each of which lists a number of cells. Each cell can have an initial value and an equation.

Each Penguims object can also have an *interface* section that comprises a set of *interactors*, objects that combine a screen appearance and behaviour. This defines the graphical appearance of the object and the relationship of the interactors to the cells of the object.

For example, Table 3.1 presents a short piece of Penguims code that consists of one object with seven cells. Cells are assigned initial values with the ":=" operator and equations with "=". (In this example no cells have both an initial value and an equation.) Its screen appearance comprises a horizontal slider and a rectangle (Fig. 3.4).

In the code of Table 3.1 the interface section has two items, a horizontal slider and a rectangle. The first two arguments to the slider are the cells that define its position on the screen. The third argument controls the visibility; in this case, the slider is defined to always be visible so the keyword "default" is used. The fourth argument is the cell that contains the position of the slider's thumb. The "controls" keyword in front of the cell name "length" signifies that the user controls the value of this cell via the slider. Sliders also have other parameters that control its other aspects, like the minimum and maximum values, the increments and decrements to use, the

```
controlled_line := object
   x := 10; y := 10;
   length := 50;
   rx1 := 25; ry1 := 100;
   rx2 = rx1 + length;
   ry2 = ry1 + 25;
interface
   h_slider(x, y, default, controls length);
   rect(rx1, ry1, rx2, ry2);
end object;
```

Table 3.1: A Penguims example—a slider controlling a rectangle

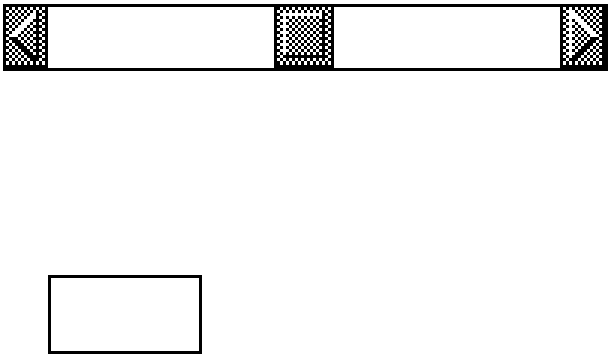

Figure 3.4: An interface created by Penguims

bitmaps to define its appearance etc. In this example, all these arguments are omitted and revert to their default values.

The rectangle takes four cells, one for each edge. The upper and left edges are both fixed, so the rectangle's upper left corner will stay at (25,100). The height of the rectangle is 25 units, as defined by the relation `ry2 = ry1 + 25`. The value of the cell "rx2" is obtained by evaluating an expression depending on the cells "rx1" and "length." If the value of the cell "length" changes (when the user moves the slider), Penguims will re-evaluate the equation and calculate the new value for the cell "rx2." This will cause the rectangle to be re-drawn with this new value. The slider therefore controls the width of the rectangle.

The system of equations in a Penguims specification forms a directed acyclic graph (DAG). Whenever some cells change, the system automatically re-evaluates all those cells that depend on the changed cells.

The system we present allows the user to create or modify a specification with Pluto, and then to translate that specification into a Penguims script that is executed to form the final display.

Chapter 4

Basic Elements and Layouts

The Graphical Layout System (GLS) supports the detailed specification of displays. Representations of atomic elements making up the display are supported in a GLS view, along with information on the relationships between them—layout and placement relations as well as interactive behavior.

4.1 Atomic Elements

Basic graphical elements (or interactors, also known as widgets [43]) in the specification are represented and manipulated in the GLS view by their bounding boxes—rectangles whose sides are parallel to the axes and which completely enclose the object. To distinguish between the various types of interactors, they have different appearances.

Each interactor has some additional parameters (called attributes or cells) that control its behavior. These are specified with an *attribute editor* that is usually hidden to save screen space but can be brought up at any time. The other attributes of these objects are properties like line-widths, color, actions, etc. Every object also has an optional name that can be specified in the attribute editor. The actions to be performed, the appearance and the initial state can be specified textually with this editor.

The interactors provided in the system are:

rectangles are objects that represent dynamic rectangles. Each edge that has not been fixed can be moved in the final display by the user. In the GLS they are represented simply as rectangles. Fig. 4.1 shows the appearance of rectangles in the GLS and in the final display.

Figure 4.1: A rectangle as represented in the GLS and in the final display

lines are represented in the GLS by rectangles with a line drawn on a main diagonal. The bounding box represents the size and position of the basic element and thus fixes it in the display. As for rectangles, any of the four positional attributes of the line that are not fixed can be moved by the user in the final display. Fig. 4.2 shows the appearance of rectangles in the GLS and in the final display.

pushbuttons are buttons that have an associated action. Every time the mouse is clicked over the button, the action is executed. They offer user-feedback—when the mouse-button is

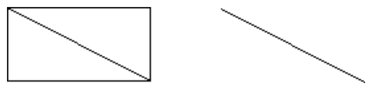

Figure 4.2: A line as represented in the GLS (left) and in the final display

pressed over it, its appearance changes (the image is highlighted), signifying that releasing the button will execute the action. Moving out of the button's extent removes the highlight, implying that releasing the button will not cause the action to be executed. Fig. 4.3 shows a pushbutton as it appears in the GLS (on the left) and as it may appear in a final display.

Figure 4.3: A pushbutton as represented in the GLS and in the final display

toggles are pushbuttons that maintain a state ("on" or "off"). Their appearance varies depending on their current state, and they provide user feedback just like the pushbuttons do. Two actions can be associated with them, one to be executed when the button is turned "off" and the other when it is turned "on." Fig. 4.4 shows a GLS representation of a toggle (on the left) and a sample toggle as it may appear in the final display. Fig. 4.5 shows an example of an attribute editor for a toggle.

Figure 4.4: A toggle as represented in the GLS and in the final display

radio-buttons are a set of two-state buttons that have the property that when any one button in the set is "on," all the others will be "off." Fig. 4.6 shows a typical radio-button set. Radio buttons are specified by using a set of toggles, and then setting the "Group" attribute for all of them to the same value, the name to be given to the radio-button set.

text objects display text, and optionally allow the user to edit it. They allow many editing actions, like selecting part of the displayed text and replacing it with typed input. Fig. 4.7 shows a GLS text object. In the final display, it appears as a string. The font, size and value of the string can all be specified with the attribute editor.

sliders are one-dimensional valuators—that is, they allow the user to specify a value within a fixed range. They can be either vertical or horizontal. They can be used purely as output

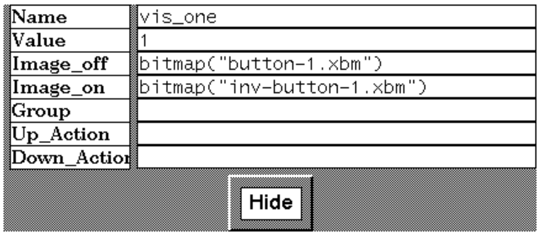

Figure 4.5: An attribute editor for a toggle

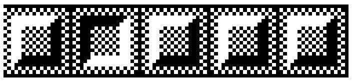

Figure 4.6: A typical radio-button set

---

devices by disabling user input, but are more commonly used for user input. The sliding image in the middle (known as the *thumb*) allows the value produced to be smoothly varied; the icons at the ends change the value in jumps, and clicking between those images and the thumb causes a large jump. The thumb moves correspondingly when any jumps are performed. Fig. 4.8 shows a typical slider. Fig. 4.9 shows the representation of a slider in a GLS specification.

bitmaps display a certain bitmap image on the screen at their current position. They can be made to accept user input, i.e. they can be dragged in two dimensions, providing a two-dimensional valuator. They do not use the "Value" cell; instead their bounds can be directly used by constraints. The bitmap to be used in the final display is specified with the attribute editor.

Interactors can be used for user input, and each of them provide some number of values. These values can be used in other parts of the display, making them interactive. Lines and rectangles accept user input by allowing the user to drag the relevant edge(s) in the final display. We will show examples for these in the next section.

Figure 4.7: A text display object in the GLS

Figure 4.8: A typical slider

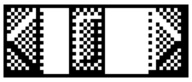

Figure 4.9: GLS representation of a slider

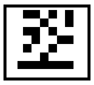

Figure 4.10: A bitmap interactor in the GLS

4.2 Constraints

The graphical elements in a GLS specification are placed on the display with respect to the positions of other elements or on the basis of user data. As illustrated in Fig. 4.11, the arrows from the edge of one bounding-box to the edge of another in a GLS specification represent these relationships, and are called *constraints*. These GLS constraints transmit values in one direction only. For example, in Fig. 4.11, the position of the left edge of the text object depends on the rectangle on the left, but the position of rectangle on the left is not set. The width of the rectangle is 20 units. Fig. 4.12 shows the display that would be created—note that unspecified edges (i.e. the ones drawn with a single line) default to their position in the GLS, so the rectangles in both figures have the same heights. The rectangle can be freely dragged, and the text object stays attached to its right edge. (The value "Hello, world!" used by the text object is specified with the attribute editor which is not shown in the figure.)

4.2.1 Specifying constraints interactively

Constraints are created by pressing a mouse button near the boundary of a GLS object. This object provides feedback to the user by highlighting the point on its boundary closest to the starting position of the drag. A line is drawn from this starting position that tracks the position of the cursor (this behavior is called rubber-banding). When the cursor is close to an attribute that can accept this constraint, the line snaps to the boundary (Fig. 4.13). Releasing the button on the mouse at this point will result in the constraint being created. Moving away from this edge without releasing the button will cause the line to resume tracking the cursor. If the mouse button is released while the line is not snapped to a boundary, no constraint will be created.

A constraint started on a vertical edge can only be connected to another vertical edge, and one started on a horizontal edge can likewise only be connected to a horizontal one. The user is notified of this through the snapping behavior—an incompatible edge will not cause the rubber-banded

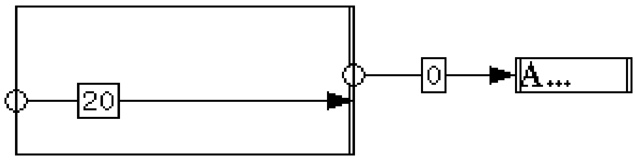

Figure 4.11: Two example constraints

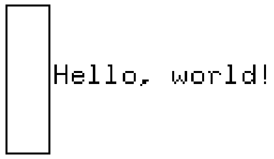

Figure 4.12: Result of constraints shown above

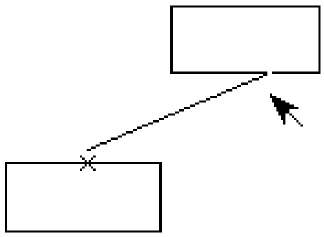

Figure 4.13: A constraint being created

line to snap to it. This snapping behavior can provide useful semantic feedback, informing the user what connections are valid for a particular constraint. It can also be used to verify the edge that will get the constraint, and to select the right edge if there are many compatible edges close together.

When an edge of an object has been fixed by a constraint, that edge is drawn with a double line, to indicate that it cannot accept any further constraints. Some objects cannot accept any constraints, so they always have doubled boundaries; others can accept multiple constraints, so their edges are never doubled. (These objects are described later in this chapter.)

Each constraint has an associated expression that allows the user to specify the relation between the two edges it joins—for example, in Fig. 4.11 the constraint defining the width of the rectangle has the expression "20." This expression is drawn in a label that can be hidden to reduce clutter, but can be displayed again by clicking with the mouse on the body of the constraint. The constraints can also be re-positioned by the user. By default, a constraint draws itself centered in the intersection of the extents of its source and destination. In some cases, this can lead to several constraints overlapping. To avoid this, the user can arrange the constraints by dragging them.

Some objects, like the interactors, have fixed sizes. Such objects can initially accept a constraint on any edge, but once an edge has been assigned a constraint, its opposing edge is also drawn with a double line, since it has now been implicitly constrained.

4.2.2 Semantics of the constraint system

The expression associated with a constraint defines its behavior. These expressions can use values from the specification, like "obj.x2*3" which means three times the value of the "x2" field of the object named "obj" (this notation is explained in detail further on), or values from the data stream.

The value of the expression is added to the source attribute, and the destination attribute is set to this value. For example, the rectangle of Fig. 4.11 has a width of 20 units, and the text object is placed so that its left side is against the rectangle's right edge.

By default, the expression in a constraint is blank, which is equivalent to the value 0. That is, the boundary being constrained has the same value as the source of the constraint. In general, an arbitrary arithmetic expression may be associated with a constraint, using values from other objects, from the data source or from user input.

Data that have been selected by the query process can be used in the labels of constraints with the notation "#name." This indicates that the value of the data field that has been labelled "name" by the query should be used there. When executed, the entire string is replaced with the corresponding value from the data stream.

Interactors in the system have certain pre-defined attributes. The current value of every interactor can be used by referring to it as a cell named "Value" attached to its name. For example, if we have defined a slider and named it "rect_width," the expression "rect_width.Value" represents the current value of the slider and we can use it as the label of a constraint (Fig. 4.14). The initial width of the rectangle will be 10, as specified in the slider's attribute editor. Among the other standard field names of each object are the names "x1," "x2," "y1," and "y2" which represent the boundaries of the object.

Objects that are constrained are automatically kept up to date. If any attribute in the system changes (usually based on some user input), any other values that depend on it directly or indirectly will be updated according to the expressions in the constraints that connect them. This is repeated, propagating the change through the entire display. Any circular dependencies among the constraints are broken by changing a value at most once during each propagation cycle (complete details can be found in [30]). In the example above, the text field will track the right edge of the rectangle continuously. If the rectangle changes size, the text field will move horizontally.

Fig. 4.15 shows an example of how unspecified edges of lines and rectangles work with the automatic update. The grey box represents some object that the user should be able to move freely in the final display. All the constraints have empty expressions (not shown for clarity) that default to "0." The top-left corner of the rectangle is not constrained which indicates that in the final display the user can grab it with the mouse and move it. Since the upper and left borders of the inner grey object depend on the rectangle's upper and left edges respectively, the grey object also moves. This causes its lower and right edges to be modified, which in turn change the values of the lower and right edges respectively of the rectangle. The net effect is that the rectangle always has the same size as the inner object, but the latter's position on the screen depends on the rectangle.

In the above example, the initial position of the rectangle and the dragged object in the final display would be the position of the rectangle in the GLS view. In general, however, we usually want to set an initial position explicitly for such objects. Using a constraint to do so is not sufficient, since that would fix the position of the rectangle and not allow the user to drag it. Instead, we

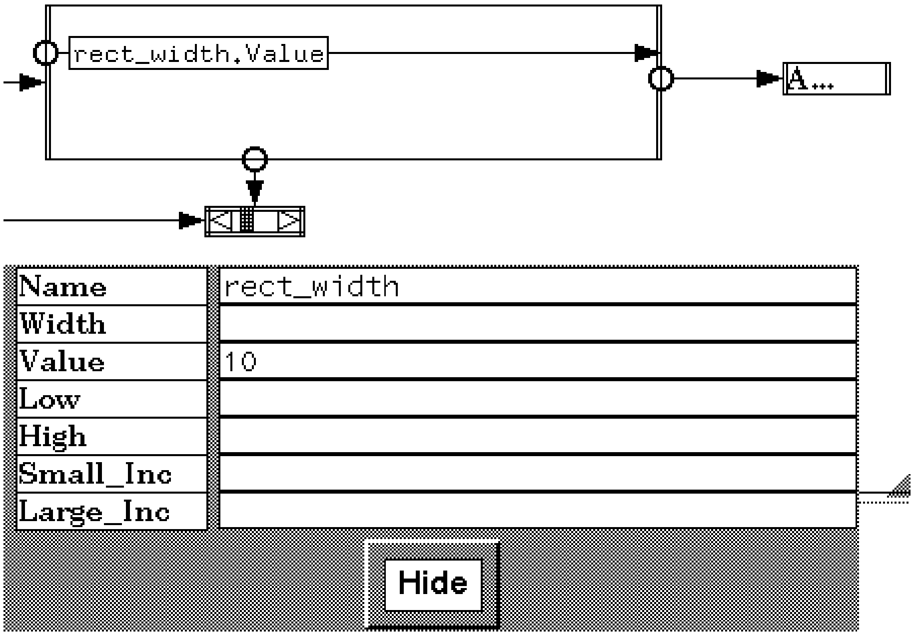

Figure 4.14: An interactor defining a behavior for two objects

prefix the expression in the constraint with "!". For example, if we wanted the initial position of the dragged object in the above example to be (10, 10) we would use floating constraints (ones not attached to any object as the source, described in the next section) with the expression "!10" for the top and left edges.

It should be kept in mind that the position of the objects on the GLS display does not necessarily represent the layout as it will appear in the final display; the objects in a GLS display can be freely moved to neaten up the specification and provide enough space so that the specification can be clearly understood. All the constraints will adjust their positions as necessary. Experience with earlier systems [29] has shown that this is the correct decision, since in the specification view, the space requirements are quite different from those of the final view. For example, displays tend to make frequent use of objects that abut each other, by means of a constraint with a zero value. If this constraint had a length of zero in the GLS view, it would be very hard to manipulate.

4.2.3 Floating constraints

A GLS constraint that defines an attribute A on the basis of another, B, defines a relationship of the form "$A \leftarrow B+$<*expr*>." We also need to allow arbitrary constraints to be specified in the GLS.

This can be done with a *floating* constraints, i.e. one not attached to a specific object as source. In that case the value of the expression is directly assigned to the destination attribute. These can, for example, be used to set objects at some absolute position on the screen. In most cases, a regular constraint should be used, since displays are usually specified in terms of distances between

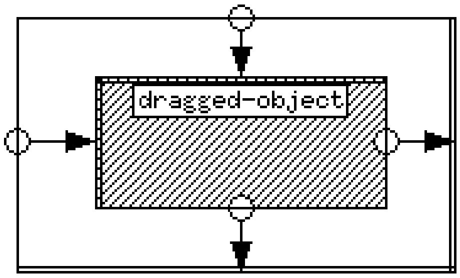

Figure 4.15: An object that can be freely dragged by the user

objects.

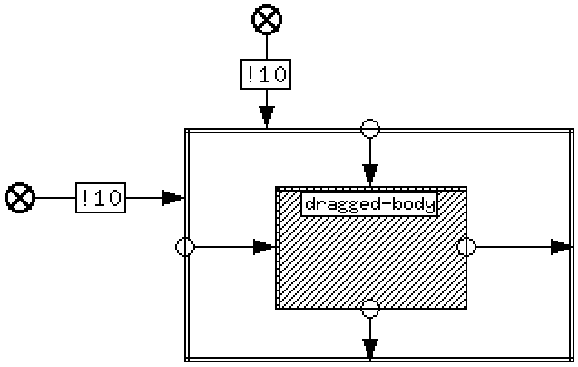

Figure 4.16: A floating constraint

Fig. 4.16 shows an example of how floating constraints may be useful. It shows the dragged object from Fig. 4.15 with two additional floating constraints. This means that the top and the left edge are set directly to the expression in the constraint labels. In this case, the expressions are '!10' which implies that the rectangle will initially be drawn at (10, 10) but will then be under user control.

4.2.4 One-way constraints vs. general constraints

The constraints of the GLS are graphical representations of one-way constraints. One-way constraints are not as powerful as more general constraints [5, 8, 35]. If two variables are related by a general (or two-way) constraint system, changes can propagate in either direction. These systems also allow attributes to be defined by more than one constraint. An example of an interaction that cannot be implemented by one-way constraints is shown in Fig. 4.17. It shows a hypothetical implementation of *drag handles* and represents a rectangle attached to four small bitmaps. In the final display created by this specification, the user can stretch the rectangle in any direction by grabbing one of the drag handles with the mouse and moving it. To see the two-way nature of the constraints, notice that if one drag handle is moved, say the upper-left handle, then the upper-right handle will also move, and vice versa.

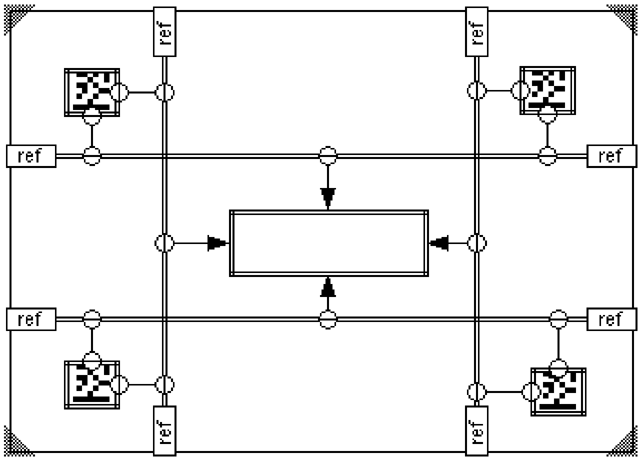

Figure 4.17: A hypothetical specification of drag handles

Two-way constraints are powerful, and for small examples like the one shown above, easy to use. Efficient solvers are available for general constraint systems [17] that can solve a system of constraints fast enough for smooth interaction. However, for large specifications, end-users tend to over-constrain displays, i.e. use too many constraints so that there is no solution. The behavior of large systems implemented with two-way constraints is also hard to understand.

One-way constraint systems can handle most types of displays. They can be solved very easily, so smooth interaction is not usually a problem. Also, they cannot be over-constrained since every value has at most one constraint. Combined with initial values, a one-way constraint system can always be uniquely solved [30].

One-way constraints have also been used in other layout specification systems [47, 66].

4.3 Grouping and Hierarchies: The Frame

The basic element of composition and re-use in the GLS is called a *frame*. It is represented by a rectangle with shaded corners, and may optionally have a name at the top-left corner. These objects do not appear in the final display; they serve an internal organizational role. The only attributes that all frames have are size and position. There are several types of frames, and each performs a slightly different function:

frame —The basic frame is a placeholder object. It can be used in a GLS specification as an invisible rectangle, to hold intermediate positions or sizes. Fig. 4.18 shows an example of a plain frame, represented by a rectangle with shaded corners.

parent-child —This object is similar to a procedure call in conventional programming languages. This type of frame establishes another level in the hierarchy. At the parent level it is drawn

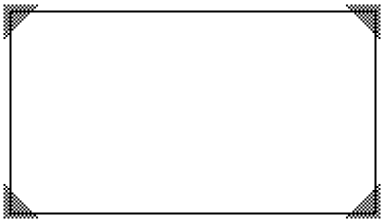

Figure 4.18: A frame

as a grey box with a text edit field; clicking in this field allows the name of this object to be specified. The internal structure of this box (the child object) is described in a different view, the child frame. This provides for encapsulation of part of the display. Since constraints cannot be drawn between different GLS views, this structure also enforces strict scoping. The only interface between the objects at the lower level and those in the parent level is the boundary of the frame.

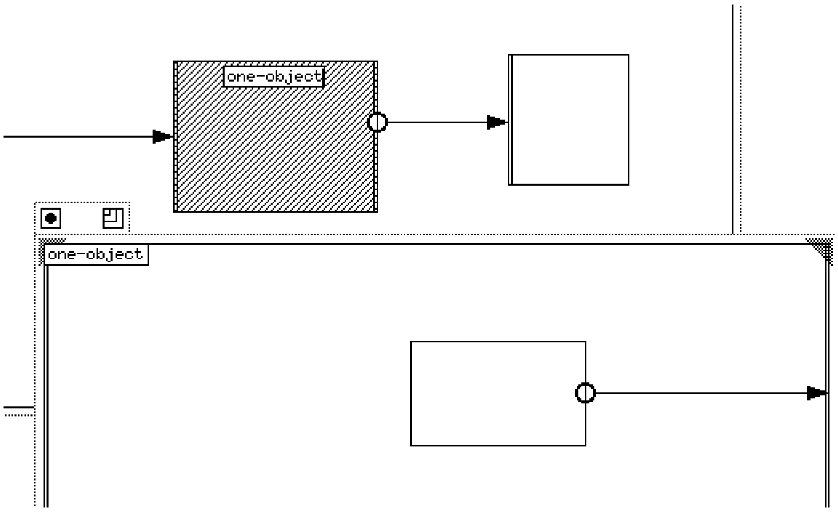

Figure 4.19: A “parent-child” object

Fig. 4.19 shows an example of the parent-child construction. The parent object is the grey box named “one-object” and is used in a GLS view. The frame labelled “one-object” is the child object and is shown in a different view. This construction can be considered a replacement—the grey parent box is replaced with the child frame and everything inside it. Note that since the parent object (the grey box) has a constraint on its left edge, the left edge of the child is also a double line. This indicates that the left edge of the child frame has a value, one that is coming in from the parent. Similarly, the right edge of the parent object takes its value from the right edge of the child frame.

To address the problem of scale, it is necessary that some method be provided of displaying large programs in the limited screen space available. Representing each node of the hierarchy in a different window lets the underlying window manager of the machine be used to manage the various views. Since users typically work on one part of a problem at a time, it makes sense to allow them to hide the other parts of the work by unmapping or otherwise hiding those windows. The operator hierarchy still provides an overview of the specification.

repeat frames —These objects allow part of the specification to be instantiated a variable number of times in the final display. The number of instances created can either depend implicitly on the size of the data-set, or be specified explicitly. Since each repeat frame processes a stream of data, it is also the most natural place to attach a query filter. Each repeat frame therefore has a query filter associated with it. This object is covered in detail in the next chapter.

conditional frame —These objects include a number of sub-frames, and a controlling expression. Depending on the value of the expression, one of the child frames is selected for the display. By default, there are two child frames, and the conditional behaves like an "if-then-else" structure. By increasing the number of child frames, a "case" construct can be specified. The next chapter discusses this construct in detail.

4.4 Reference lines

Reference lines are vertical or horizontal lines that are attached to a frame object. Like frames they do not appear in the final display. They are provided to make the specification easier to describe and provide a convenient vehicle for specifying several common computations. They have only one attribute, an x or y position, depending on the orientation. They are drawn as a line that can be dragged but stays inside the parent frame, and a label at the attachment point that serves to distinguish between the various varieties.

The different varieties of reference lines are:

plain —These are the simplest form of reference line. They do not have an intrinsic value, but can accept constraints. These objects are modelled after the reference lines used in drafting. When some parts of the drawing are to be precisely aligned, thin temporary lines are often drawn and used to guide the placement of the figures. After the drawing is completed, these guide lines are erased. Similarly, the reference lines used in Pluto are useful for aligning objects, but do not appear in the final display. When an object has many constraints leading from it, it may be better to set a constraint from it to a reference line, and then use that line as the source of all the other constraints. When used properly, reference lines can reduce clutter and make specifications easier to understand. Fig. 4.20 shows two equivalent GLS specifications, the one on the left without reference lines, and the one on the right with two of them, one horizontal and one vertical.

proportional —These reference lines take a position that is a fixed proportion of their parent frame. The position this reference line is moved to in the parent frame defines the proportion. By default, some "good" positions are provided: when the reference line is moved in the frame by dragging with the mouse, the line snaps to the positions corresponding to these fractions:

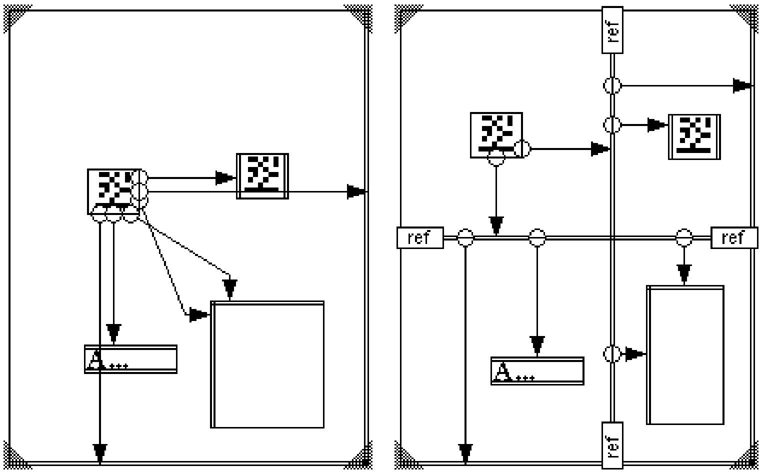

Figure 4.20: An example of the use of reference lines

1/10, 1/5, 1/4, 3/10, 1/3, 2/5, 1/2, 3/5, 2/3, 7/10, 3/4, 4/5 and 9/10. However, arbitrary real numbers between 0 and 1 can also be specified by the user. If the size or position of the parent frame changes, the position of the reference line is automatically adjusted to maintain the proportional relationship. Since the position of this line is determined by the parent frame, it cannot accept any constraints and is always drawn with a double line. The label on these lines at the attachment point to the parent frame is “pro.” An example of a proportional reference line is shown in Fig. 4.21; its ratio is 1/3. Note that it is drawn with a double line implying that it cannot accept any constraints.

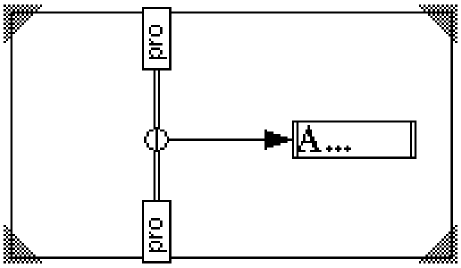

Figure 4.21: An example of a proportional reference line

operator lines —These lines can accept any number of constraints. All the incoming constraints are evaluated, and the object then computes a value by applying its function to them. The value of an operator line is the value thus computed. The currently implemented operator lines are “max,” “min” and “ave” which compute maximum, minimum and average of the values derived from the incoming constraints. Since they can always accept more constraints, the edges of these reference lines are never doubled. The labels of these reference lines indicate the value they compute. Fig. 4.22 shows an example of a maximum reference line used to ensure that the display is large enough to show all three of the slider, the text and rectangle objects. The reference line object has three constraints incident on it but is still drawn with a single line since it can accept additional constraints.

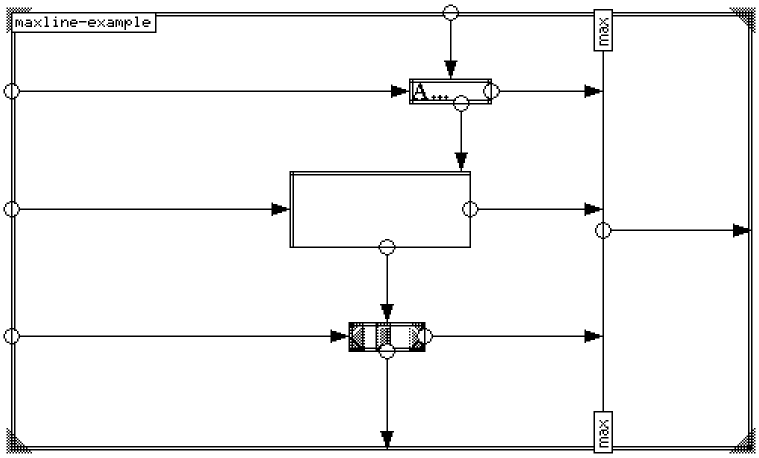

Figure 4.22: An example of a maximum reference line

The plain and operator lines can be freely dragged in one dimension by the user, although they will not move outside the boundaries of the parent frame. Their position in the GLS view does not correspond to the value they will take in the final display. The proportional line is the only one whose GLS position is significant; its position in the parent frame reflects the proportion it maintains.

All of these constructions taken together—atomic display elements, constraints, frames and reference lines—allow substantial flexibility in the specfications. A wide range of display types can be implemented, and the incorporation of interactors allows the easy specification of interactive displays. The next chapter considers in detail the constructs that allow these objects to be composed in flexible ways.

Chapter 5

Representation of Control Structures

To support a full range of visualization alternatives we need to be able to handle a variable number of data items (and corresponding graphical elements), as well as to handle variations in the structure of the display based on the data. To do this we must go beyond the one-to-one correspondence between specification objects and the graphic elements in the final display by introducing more abstract constructs. These constructs form the equivalent of what in conventional programming languages are control structures.

The query process usually generates a variable length stream of data values that need to be rendered. To describe the display that is to be created, we need to able to create new objects at execution time, rather than at specification time.

The structures presented in this chapter are more powerful than the simpler ones discussed in Chapter 4. It is expected that only advanced users will need to use these forms, since for most common visualizations (scatter-plots, histograms etc.) pre-defined programs can be used from the library. Most of the users that will need to use new and innovative displays will be technically sophisticated (engineers, scientists etc.), and do not have serious difficulties with such abstract concepts. This is borne out by the preliminary user testing (discussed in Chapter 8). Once the new displays are in use, the specifications for them will be available to the other less advanced users.

5.1 Conditional Elements

We need a way of representing conditional structures. Depending on certain aspects of the data, we may wish to draw figures that differ structurally from each other. For instance, we may choose to represent part of the data in an expanded or compressed form based on user input through an interactor, to implement a form of fish-eye view [18] of the data. Another application of conditional elements would be to implement "dynamic queries" [1]. These are displays that control the appearance of certain objects based on user interaction. For example, a display might be constructed with only those data elements whose values exceeded a threshold controlled by a slider. The system dynamically computes the condition and modifies the display.

In the chart showing Napoleon's army in Russia (Fig. 1.2) we see one use of a conditional structure. The lines connecting the points on the temperature chart to the main display are only drawn at those points for which a temperature reading is provided.

We propose the *conditional frame* (Fig. 5.1) to handle this requirement. Each conditional frame has an associated expression, and a number of child frames. Each of the child frames can have an expression as a label. Upon execution, the child frame whose label matches the parent's expression will be chosen for the display. If no value matches, no display will be provided. The example presented in Fig. 5.1 is similar to the case construct in conventional programming languages.

This notation handles both the conventional "if-then-else" type of structure, as well as the "case" type of structure. When first created, conditional frames are given two sub-frames; more

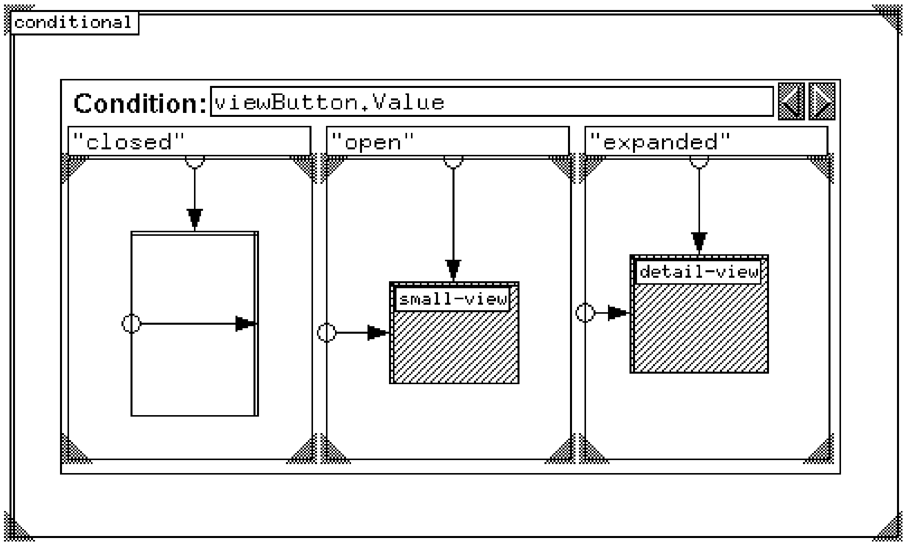

Figure 5.1: A conditional frame implementing a "case" construct

can be added with the buttons in the top-right corner. If empty, the labels of the child frames default to 0, 1, 2, 3, ... etc. If there are two child frames in the conditional (the default), the default labels are either 0 and 1 or *true* and *false* respectively, depending on the type of the control expression; therefore the default behavior of the conditional is like the "if-then-else" construction in conventional programming languages.

In the figure, the parent's expression (the value of the interactor called "viewButton") is evaluated. The labels for each of the three children are evaluated, and depending on the value of viewButton, one of three possible representations for the data ("closed," shown as a simple rectangle, or "small-view," a slightly more detailed view, or "detail-view," a fully detailed view) will be chosen.

5.2 Repetition

A variable number of objects can be described using *repeat frames*, which contain a representation of the next instantiation of the frame (and included objects) as a grey box labelled "Next" (Fig. 5.2). The repetition is unfolded to handle all the data values in the stream, with all the objects and relations that are inside the repeat frame being duplicated. Any constraints that use the value of the positional attributes of the outside of the repeat frame get the value assigned to the "Next" body of the previous instantiation. This is effectively a recursive version of the parent-child construct.

The example in Fig. 5.2 is a GLS description of a histogram that draws bars 10 units wide by using rectangles of width 10 whose heights are a data item that has been named "value" in the query filter. It demonstrates many of the features of the overall notation: constraints' labels can be optionally hidden (in this example, all hidden labels have the value 0); edges of objects that have been constrained, and therefore can no longer accept additional constraints, are drawn with a double line; and a "max" reference line is used to derive the top edge of the histogram.

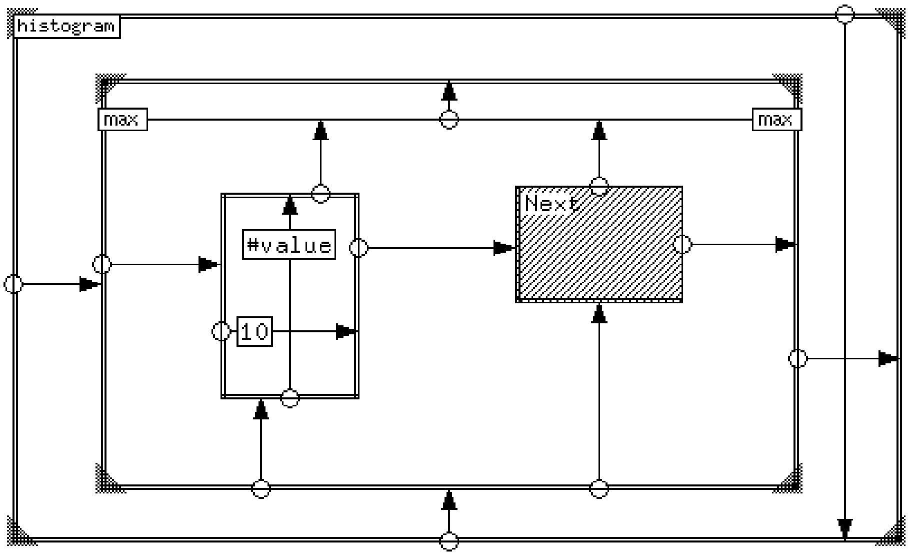

Figure 5.2: Pluto specification for a histogram

This specification can easily be modified to form a "hanging rootogram" [69], where we want to measure the fit of the data to some standard curve—say the normal distribution. Instead of comparing the tops of the bars to the standard curve, we can move all the bars up, aligning their tops with the curve, so that the bottoms of the curve will measure the deviation from the standard (Fig. 5.3). This is also a good example for illustrating the use of interactors to explore the data—the user could pick from a number of candidate reference curves (using an *n*-state device like radio-buttons) to form the rootogram, or use a continuous-input interactors (like sliders) to adjust the parameters of the reference curve.

Fig. 5.3 shows a complete interactive display created by the GLS specification of Fig. 5.4, Fig. 5.5, Fig. 5.6 and Fig. 5.7. The reference curve (in this case a simple parabola) has three parameters, that have been named Max, Height and Width. The first two control the position of the curve, and "Width" controls the spread. These values can be adjusted with the sliders, and the rootogram automatically updates the curve and moves the rectangles so that their tops stay aligned with the curve. The numeric readouts on the bottom right display the current value of each parameter. The upper edge of each rectangle in Fig. 5.5 is defined by a floating constraint that picks the "y1" value of the corresponding line-segment in the reference curve spec.

Fig. 5.4 shows how the curve is constructed. It is built up as a set of connected lines, with the positions depending on the parameter values from the three sliders. The lines themselves are named "Curve" (as indicated in the attribute editor) so that the rootogram rectangles can use their positions. The curve also uses the same input file to drive the repetition so that the curve and the rootogram both have the same number of elements, and they can be overlaid. The rootogram is made of a set of repeated rectangles. The height of each bar depends on the incoming data.

The top of each rectangle needs to be set to the position of the corresponding segment of the reference curve. To do this, we use the fact that each repeat frame also defines a symbol called "Serial_no" that represents the current index in the repetition, and the copies of objects that are

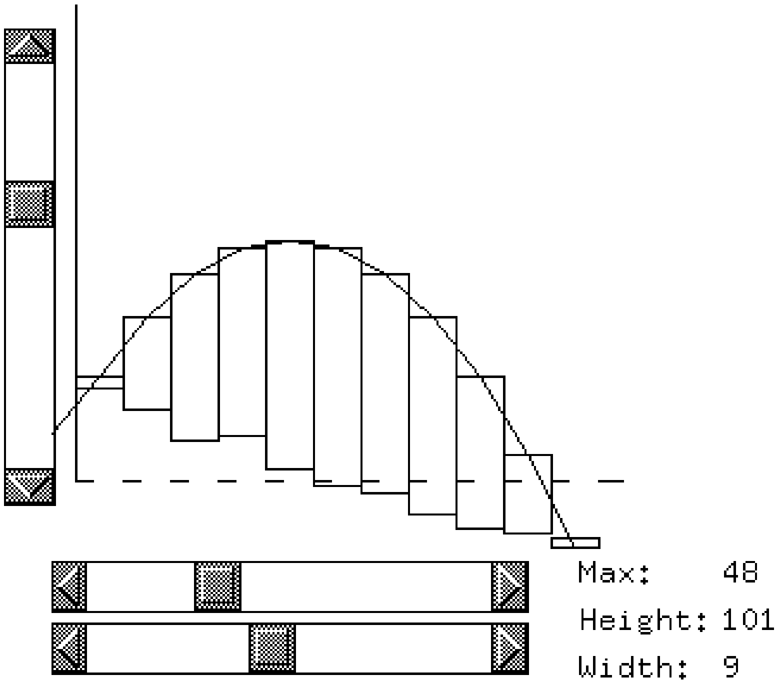

Figure 5.3: A hanging rootogram showing fit to a standard function

created for the repetition are named by appending the index to the name of the GLS object. For example in the reference curve (Fig. 5.4 the line is named "Curve" in the GLS spec.; the lines that are created will be named "Curve1," "Curve2," "Curve3" etc. Since the repeat frames in Fig. 5.5 and Fig. 5.4 replicated the same number of times, the specification for the rootogram bars can use the expression "Curve#Serial_no" to represent the corresponding line segment in the reference curve display. A floating constraint is used to constrain the top edge of the rectangle.

The controls for the display are shown in Fig. 5.6. Two proportional reference lines are used to place the six text objects—three that display the names of the attributes, and three to display the values of the sliders. Note that the vertical slider is placed with its lower edge aligned with reference to the upper edge of frame labelled "controls" which implies that this slider is drawn outside the boundaries established by that frame. In general, objects in the display need not be drawn inside the boundaries of their parent objects.

To form the final display, the GLS specifications described above are composed together with the operator tree shown in Fig. 5.7. The curve and the bars are overlaid to form the hanging rootogram, which is then placed in a row to form the final display.

5.3 Queries and Filters

A query object is attached to each repeat frame that controls the number of times the objects in it will be replicated. If the input to a repeat frame is a data source (as specified in a query object), the

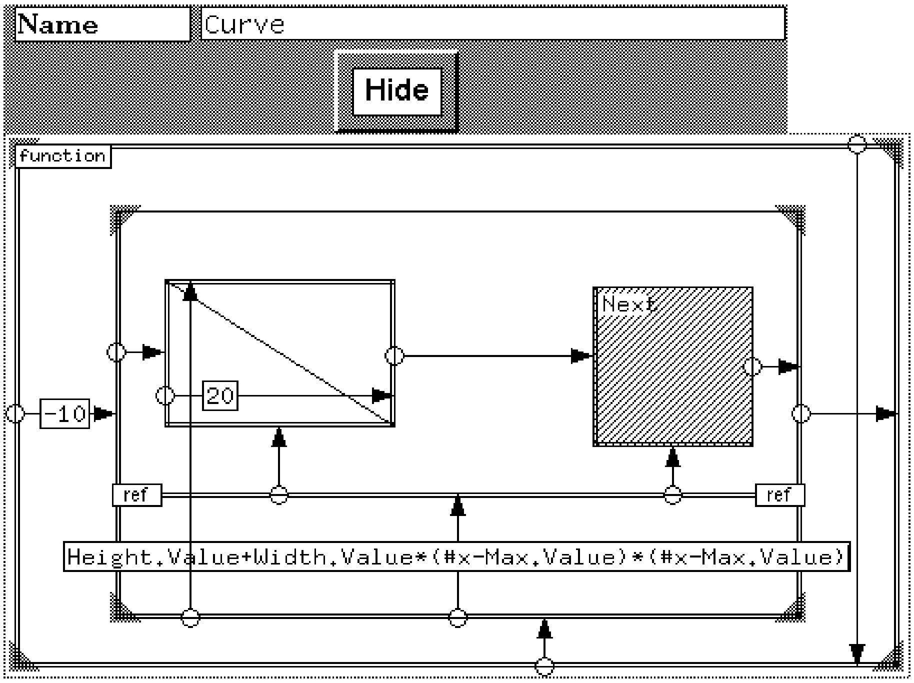

Figure 5.4: The reference curve for the rootogram

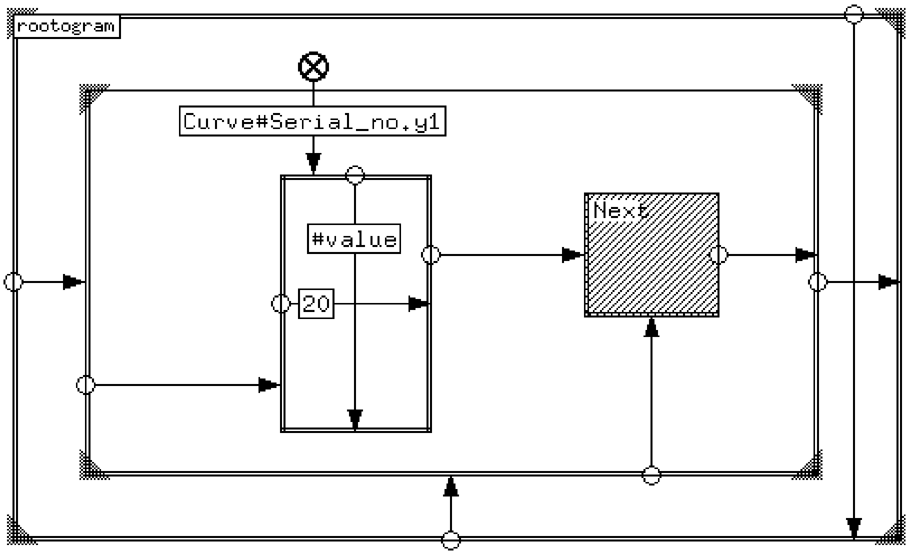

Figure 5.5: GLS specification for the rootogram bars

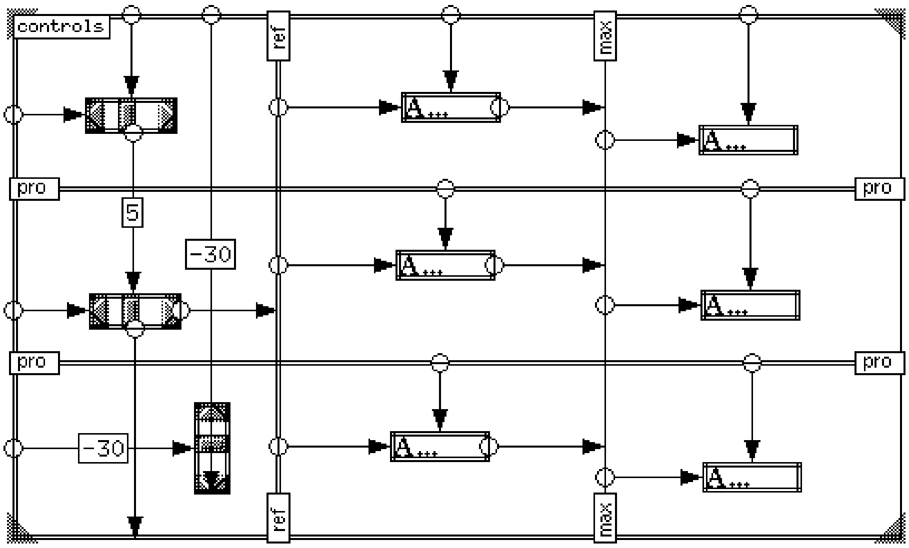

Figure 5.6: Controls for the rootogram

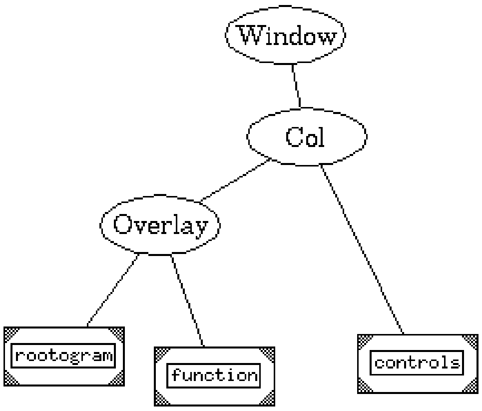

Figure 5.7: Operator hierarchy for the rootogram

repeat frame is repeated to handle all the data items in the stream. Sometimes, however, we need objects to be repeated based on some other criterion. In that case, the query object can be used to specify an expression that will be evaluated for the number of times to run the repeat. In the most common case, say to create a scale to use with a graph, this will just be a number, but in general, it can be an arbitrary expression that could depend on user input or on a data field specified by a repeat frame higher up in the GLS hierarchy.

The techniques presented in the earlier sections will work with essentially any data-selection mechanism. However, to provide a complete and useful tool for end-users, we must also consider that part of the data visualization problem.

Data Selection

In many applications, there is a large amount of data, only part of which is currently of interest. Therefore some method must be provided for picking out the interesting parts to visualize.

Also, the components of each of the data sets need to be separated into atomic elements like integers and strings, and there needs to be a way of referring to these atomic data elements from the GLS view.

There are two major approaches to selecting interesting data:

- Query languages, as used in relational databases. This is an approach well suited to the expert user, and is a very powerful method. Multiple relations and data-sets and complex operations can easily be handled. The queries themselves are compact and very expressive. An example of this kind of approach is SQL [15]. An interesting variant of this approach is to use a visual specification for the database queries [10, 14].

- Simpler pattern matching systems: this approach is simpler, but can be less powerful. These systems can range from simple text-based tools (like the AWK [2] command on Unix systems), to very powerful graphical "fill-in-the-blanks" tools, like query-by-example systems [74]. This approach is likely to be more popular with end-users, as the specification systems are likely to be easier to learn.

For end-users, high level query languages are difficult to use and understand, and require a long training period [33], while the pattern-matching filter is likely to be more intuitive, and easier to learn. A point and click data selection method has the advantages of direct manipulation—the continuous visible representation of objects of interest, in this case the fields in each data item. Instead of learning the syntax for the operations, users can click on the field they wish to use to restrict the data to display.

Visual query systems like Visual SQL [14] or Picasso [33] take care of many of the problems that end-users have with more advanced systems like SQL. Such techniques could be incorporated into Pluto instead of the prototype selection mechanism proposed here.

A Simple Data Selection Tool

In our application context, there is a smaller need for complex operations like joins on the data sets, so we don't always need the power of relational queries. A simple point-and-click data selection/naming tool can be adequate. We therefore chose this form of data selection tool for the prototype implementation.

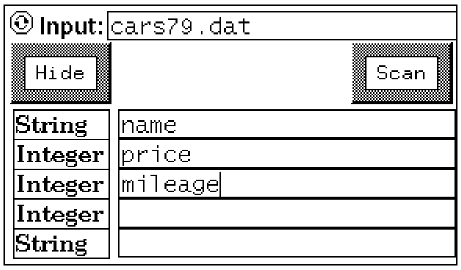

Figure 5.8: A query filter

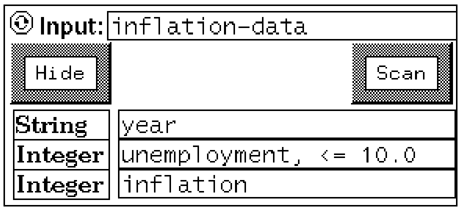

Figure 5.9: A query filter used for data selection

Figure 5.8 shows an example of the tool that associates the data fields with names and allows selection criteria to be specified. The user enters the source of the data in the text entry field at top, and presses the "Scan" button. The system then examines the data for the number of data fields and their types. For each field, a label indicating the type is displayed, along with a text entry field. The name for each field in the data may now be entered in this object. Each such name can be referred in the GLS by prefixing it with the '#' character. The query also defines a symbol called "Serial_no" that represents the current index in the loop when the repeat frame is interpreted.

The selection of the data can be specified at the same time. An arbitrary boolean condition can be associated with each data field. When the display is being drawn, only those data that satisfy the conditions will be selected. If no conditions are specified, all data will be matched and displayed.

In Fig. 5.9 the three components have been named. In addition, the field "unemployment" has an associated condition. Only the data that match this condition, that unemployment be less than 10.0%, will be chosen for display.

The data selection is implemented by translating the conditions to an AWK program. In the above case, the condition is that the third field be less than 10. This is equivalent to the following AWK script:

```
BEGIN        {FS = "\t"}
$2 <= 10.0   { print }
```

Multiple conditions on the input data are handled by conjunction. Simple conditions like the one used above are handled by just appending it to the AWK name of the field, in this case $2. More complex conditions can be specified by using the symbol "$" to represent the data field. To

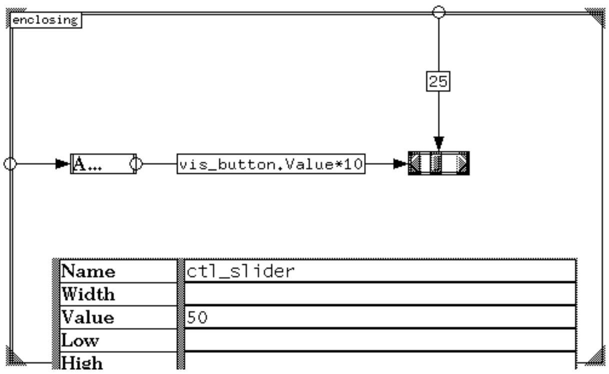

Figure 5.10: A Pluto specification

specify a range of acceptable values, we would use a condition like "$ > 5 && $ < 10". The name of the field is attached to the condition only if the "$" symbol is not used; thus the conditions "<= 10.0" and "$ <= 10.0" are equivalent.

Although useful as a proof of concept, this tool has many limitations where end-users are concerned—someone not acquainted with AWK may find it hard to specify the selection conditions. A visual selection tool is likely to be more useful for novices and occasional end-users.

Naming Data Fields

A query generates a stream of data items, each of which is an n-tuple. The selection tool, in addition to filtering the data, also allows the users to name each component of the n-tuple. The tool sets up a symbol table, associating each name with a value; any objects below the repeat frame (to which the query is attached) in the hierarchy can use the names. The appropriate values from the data source are used for each instantiation.

The names defined follow conventional scoping rules. Multiple queries may be used in a specification; if any field names are re-used, later definitions hide earlier ones.

Although dynamic queries in the style of [1] are best handled by an appropriate GLS specification like the dynamic rootogram presented earlier (Fig. 5.3), the query system provides a rudimentary way to execute simple dynamic queries. The query can be modified by the user, and then tried out (with the "Try It" menu option). Although this method will not provide the smooth feedback in the style of the dynamic rootogram, it may be appropriate for some uses.

5.4 Advanced displays with Penguims

While the above system can deal with most of the common display types, occasionally more advanced features may be required. Pluto specifications are translated into code for the Penguims system (Section 3.3). This system offers a number of additional features. Since each constraint and attribute of a Pluto object is translated into a Penguims cell, many of these features can also be made available to Pluto users.

```
ctl_slider := object
   x1 = title.x2 + vis_button.Value*10;
   y1 = enclosing.y1 + 25;
   x2 = x1 + width();
   y2 = y1 + height();
   Value := 50;
interface
   h_slider(x1, y1, default, controls Value);
end object;
```

Table 5.1: A Penguims example—a slider

In the example (Table 5.1), we see the Penguims code that implements the horizontal slider (shown with its attribute editor) in Fig. 5.10. The position of the slider is set by the constraints which map on to Penguims equations. The initial values of fields are set from the property list, and un-specified attributes are assigned defaults or omitted.

The constraints are solved by Penguims in a a lazy manner, so that only if a value is used is it computed. This algorithm is described in detail in [30, 25]. The constraint solver is thus fast enough to provide good user feedback. Since the one-way constraints of Pluto can contain arbitrary expressions, advanced users can access most of the features of Penguims directly from Pluto, without having to learn its full specification language.

The display or interaction component of an object is established by the *interface* section of the Penguims object description. The interface section contains calls to the interactors, which define the appearance and behavior of the object. In this case, the interface is *h_slider*, a horizontal slider. The first two arguments specify the position, and the third one the size—in this case, the width. The keyword *default* is used, signifying that the default size for a slider—200 units—should be used. The *controls* keyword in front of "Value" signifies that this interactor will control the cell named "Value"—as the user changes the position of the slider thumb, the value of the cell "Value" will change.

We note that the horizontal position of the slider is described by an equation that depends on the cell "vis_button.Value" (i.e. the cell named "Value" in the object named "vis_button"). If the user changes the value of that cell, the position of this slider will automatically change according to the equation.

Values of cells can also be *actions*, which are fragments of code that can be automatically executed on user input to certain interactors like pushbuttons. These code values are similar to the block values of the Smalltalk-80 language [19]. These are used in interactors like the buttons, which can have cells containing code values. When the button is pressed, the appropriate code value is executed. This feature can be used to provide advanced functions from Pluto.

In the example, the Pluto attribute "Value" was set to 50. This gives the initial value for the corresponding Penguims attribute cell and indicates that it should subsequently be free to change under user control. In the interface section, this cell is used with the "controls" keyword. The other cells all have equations that describe how the value should be obtained.

A one-way constraint system normally does not allow cyclic dependencies among constraints. However Penguims can handle cycles in the equations by evaluating them until the cycles is

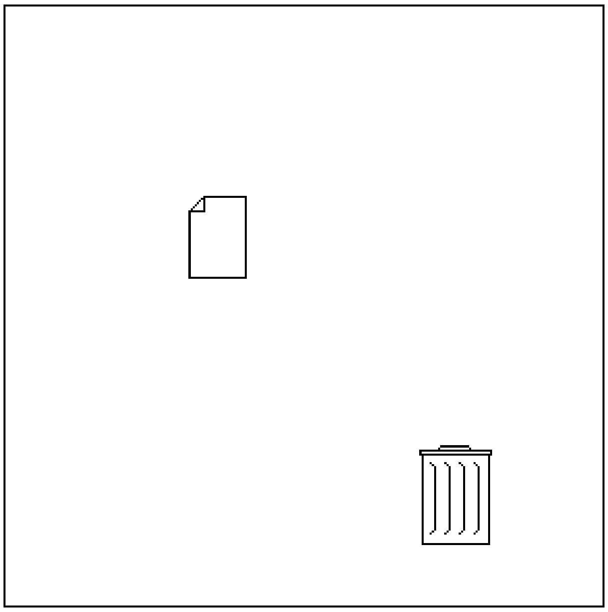

Figure 5.11: An animated trashcan

detected and then stopping. The advanced user can use cycles from a Pluto specification. If a cycle is created by just typing in equations, the cells involved will not have an initial value. To handle this, we use the fact that Penguims cells can have both an equation and an initial value. In general, an equation in Pluto (expressed either as a constraint or in the property list) takes the form "*expression* := *value*" which indicates both an initial value and the expression to use to update it.

5.4.1 An advanced interaction example

We now present the specification for an animated trashcan. The appearance of this display is shown in Fig. 5.11. It presents a trashcan object that is sensitive to objects being dragged into it. The document represents an object that can be dragged. When the document is dragged to the trashcan, the latter changes its appearance, i.e. draws itself with an open lid (Fig. 5.12). If the object is dragged out of the trashcan's extent while keeping the mouse button depressed, the appearance reverts to the simple trashcan with a closed lid.

To implement this interaction technique in Pluto, we use two additional attributes that bitmaps possess: *pickup* and *drop*. These are code values that are executed when the bitmap is picked up with the mouse button and dragged, and when the mouse button is released. We also need to use an additional cell that will keep the value of the currently dragged object, called the *drag focus*. The trashcan is actually a conditional object, and it picks one of three bitmaps, depending on these conditions:

- the drag focus is empty (nil); pick the plain trashcan.

- the drag focus is a document:

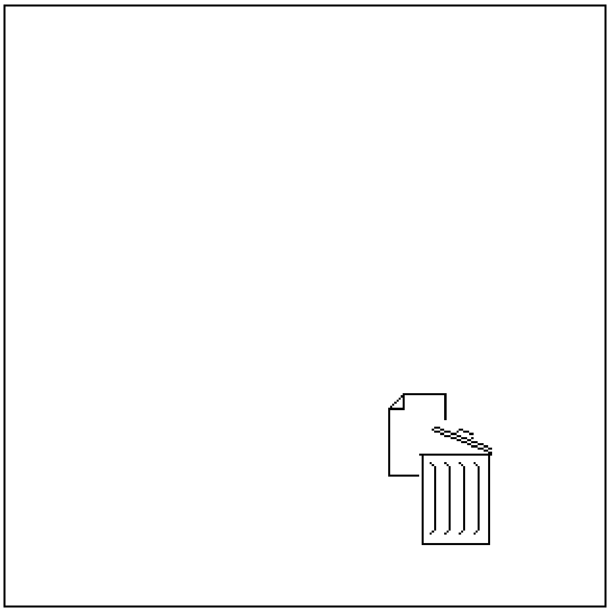

Figure 5.12: Dragging a document into the trashcan

- the object is inside the extent of the trash-can; pick the trashcan with the open lid.
- the object is outside the extent of the trashcan; pick the plain trashcan.

A document is implemented as a simple bitmap. In Fig. 5.13 it is represented by the bitmap object in the upper left, whose position is set to (10, 10), and the exclamation point is used to allow the user to drag the document freely. For the drag focus, we use a frame (in the upper right, with a floating constraint with the expression "!nil") as a placeholder and name it "drag" (as shown in the attribute editor). The frame does not have a graphical representation, and offers four cells that can be used in other relations. We use the *x1* attribute of the frame to hold a reference to the dragged object. The conditional object in the upper GLS view of Fig. 5.13 picks either the bitmap on the left, which is a plain trashcan, or the parent-child object on the right depending on whether an object is being dragged.

The child object is named "dragged" and is represented in the lower GLS view of Fig. 5.13. Depending on the position of the object begin dragged, either the image on the left (a trashcan with an open lid) or the image on the right (a plain trashcan) is picked.

When a document is picked up with the mouse, its "pickup" action is performed, which sets the cell holding the reference to point to itself (the dragged document). When the mouse button is released, the cell is cleared (Fig. 5.14). The trashcan object bases its image on the value of this cell. (In the the lower GLS view of Fig. 5.13, the "insideness" test has been simplified.)

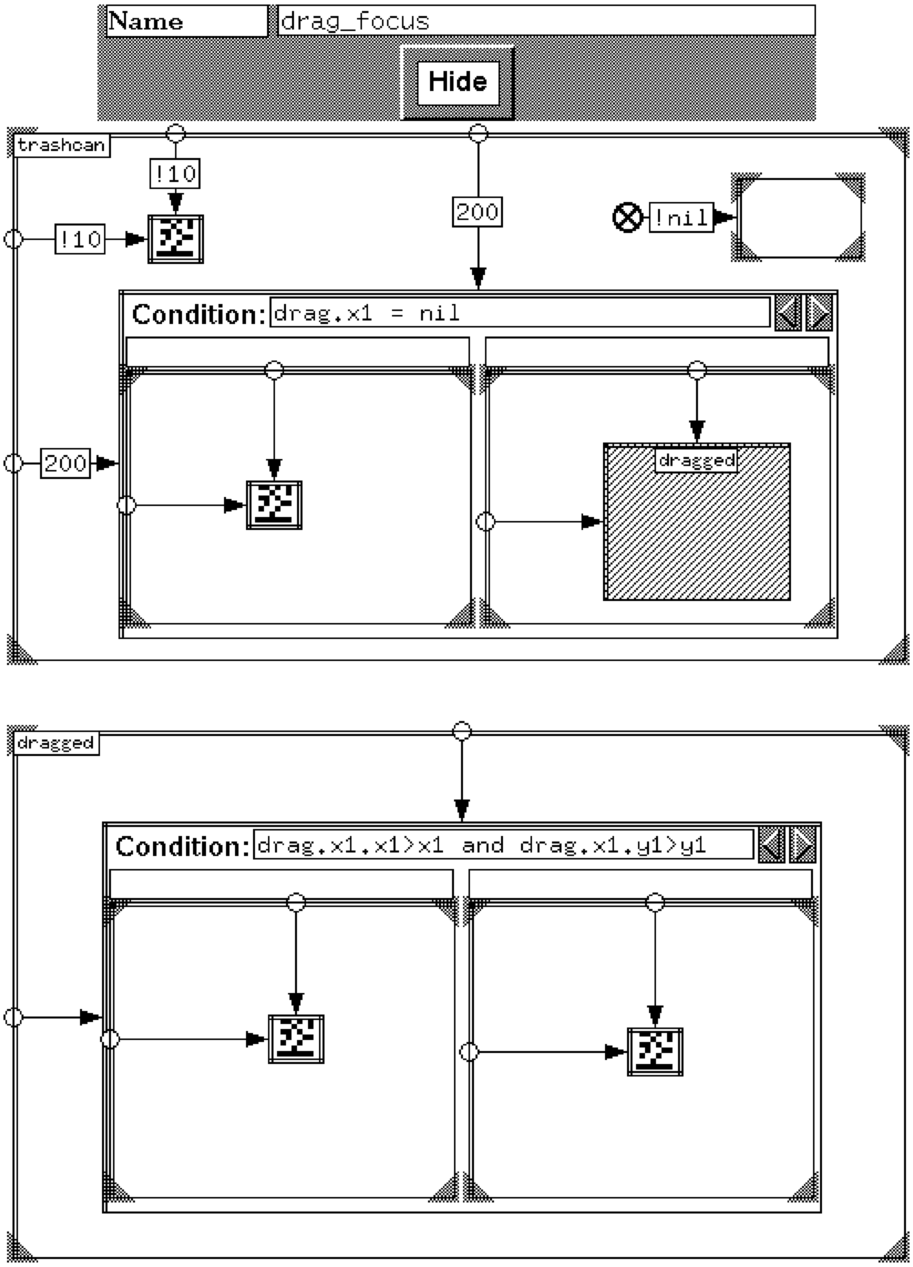

Figure 5.13: Pluto specification for the animated trashcan

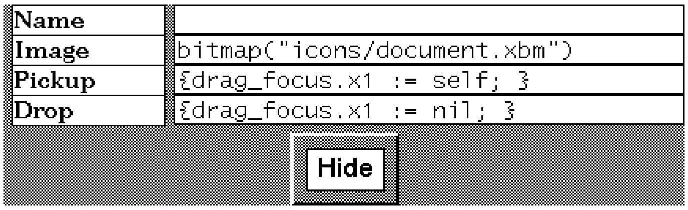

Figure 5.14: Attributes of a document

Chapter 6

Composition Operators

Composition operators provide a way of dealing with the problem of scale. The user can design the smaller components of the display separately, concentrating on them one at a time. When the smaller parts of the display have been designed with the GLS, they can be combined to form the larger display. The composition tree provides an alternative view, an overview, of the specification and offers support for incorporating pre-defined displays.

6.1 The Composition Tree

A composition operator is represented by a node in the composition tree. It is an operator that takes the displays created by its children and applies a specific operation to them. For example, the displays created by its children may be overlaid on one another, or they may be arranged side-by-side in a row. The resulting larger display is then passed on to its parent.

There are several operators that are pre-defined in the system. These are the more basic operations, like “row,” “column,” and “overlay”. These operators arrange their children in a row, a column or super-imposed on one another, respectively. The “window” operator simply causes its child to be drawn on the screen. This operator is the root of the tree for a complete specification. The user can also define new operators and use them in the display, share them with other users, or store them in a library.

The operators have other attributes that specify things like spacing between the images being composed, and borders or spaces around the composite image. These attributes are usually hidden in order not to detract from the representation of the hierarchical structure, but can be brought up at any time with an attribute editor, and edited by the user (Fig. 6.1).

Operators can take a fixed or variable number of arguments. Of the pre-defined operators, “window” takes only one argument but the others can all take a variable number of arguments. User-defined operators can be implemented in either way.

The nodes in the operator hierarchy are of two types, represented by ellipses and rectangles.

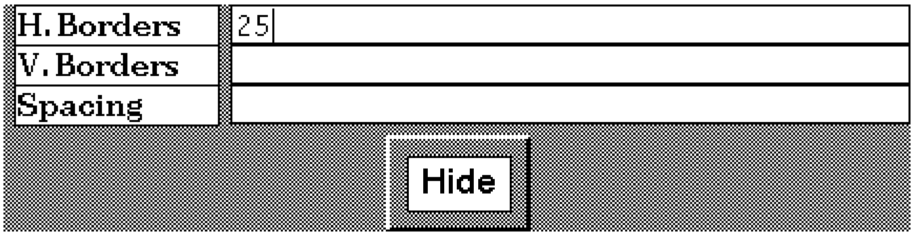

Figure 6.1: The attribute editor for a row or column operator

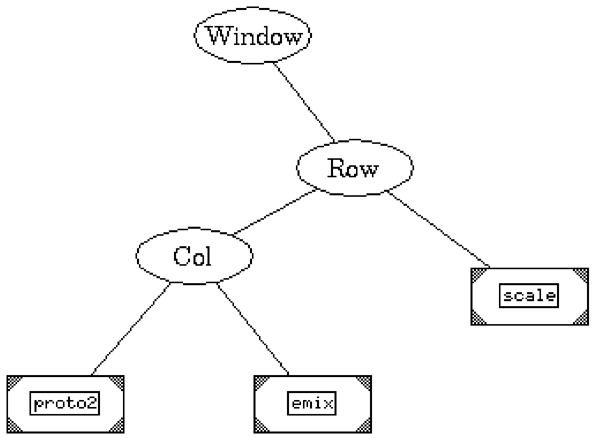

Figure 6.2: A composition operator tree

The ellipses are the built-in operators, which compose graphs and are selected from a library. They cannot be directly modified. The rectangles are representations of the structure of the GLS specification, and can be viewed in more detail in the GLS views. They can be directly modified.

Fig. 6.2 presents an example of a composition tree. It composes three GLS specifications together and displays the result on the screen. Fig. 6.3 shows three boxes (representing components of the display) laid out by the operator tree. The two boxes labelled "proto2" and "emix" are put together in a column, and the resulting picture is placed by the side of "scale" and then all three are displayed on the screen.

The order in which child nodes are attached to parents is significant. In the case of a "column" composition, the children are considered from left to right and arranged in a column from the top down. The hierarchy canvas enforces the tree-like appearance of the operator tree. Operators can be moved freely by the user as long as they do not go above their parents or below any of their children. The order of the children can be freely changed at any time; when the user selects "Try It" from the system menu or saves the spec, the current order of the children is used.

6.1.1 Implementation

The current prototype implements the built-in operators in one pass. A depth-first traversal is performed in the operator hierarchy. Each operator node sets places its first child according to its vertical and horizontal border attributes. The child is then drawn, and its lower right corner is found. The next child is placed according to the composition being performed. When all the children have been placed, the operator sets its own lower right bounds to the maximum of the lower right bounds of the children.

This algorithm is simple and works in most cases. However, it requires that the GLS specifications be based on the position of the the top left corner. In general we would like to let the

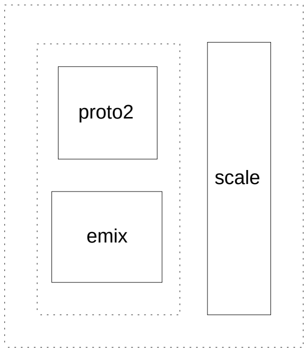

Figure 6.3: Boxes laid out by the composition tree of Fig. 6.2

user base it on whichever coordinates are convenient. This requires a two pass approach: first, the width and height of each child is calculated; then, all the children are placed according to the compositions required.

6.2 Multiple Views

The GLS section (Chapter 4) describes the detailed layout and specification of the display. It also has a hierarchical structure, and this structure is reflected in the display of the operator hierarchy. As described above, the built-in composition operators are drawn as ellipses, while those implemented with GLS specifications are represented by rectangles. Since composition operators in the tree are themselves described using the GLS notation, the operator hierarchy and the GLS displays can be considered to be two views of the entire specification. In practice, though, the operators are often specified off-line and stored in a library, instead of being specified alongside the display. However, as mentioned earlier, some operators are directly implemented in the system.

The different appearance of the GLS nodes in the tree indicates that they are slightly different from the built-in operators. They have more detailed views of the internals, and they can be modified directly. The leaf nodes also display the name of the corresponding GLS node.

The multiple-view paradigm can help users understand the organization of the display by showing the overall structure as they work on the details. Many systems present the user with just an enlarged, detailed view of a large display. This detailed view can be moved around by the user to navigate through the whole display. However, under this system users often get lost, i.e. they may find themselves in an unfamiliar part of the system without any idea how to get to the right place.

We can augment the detailed view by an overview of the whole system, showing the structure of the display, as sometimes done in the case of paper road-maps. By being able to refer to the overall structure at any time, the user can maintain an accurate mental model of the display, while still keeping a detailed picture of part of it.

This hybrid approach also helps with the problem of scale inherent in the construction of large programs—the two views, showing both an overview and a detailed view of the program, work like

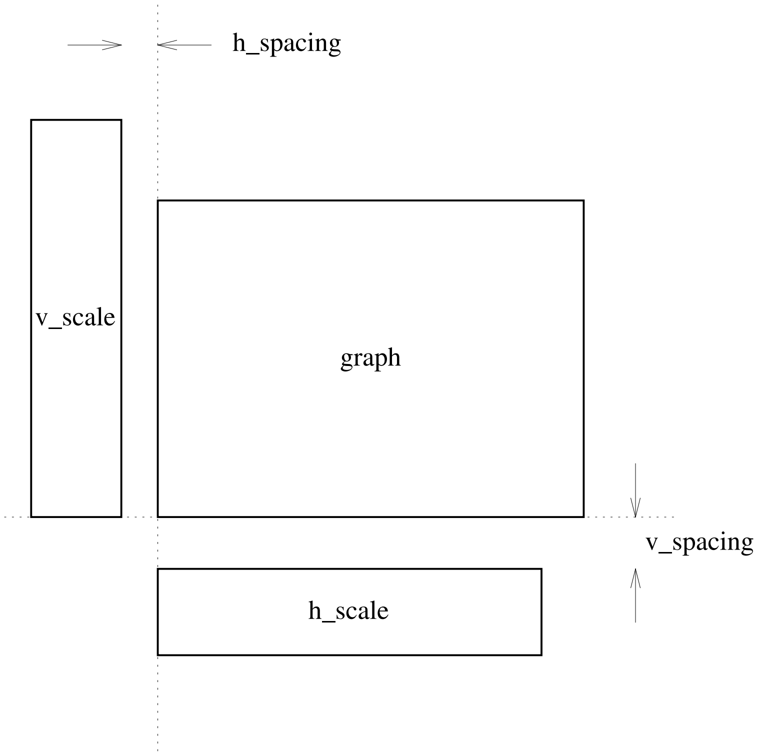

Figure 6.4: The ternary “add_scales” composition operator

fish-eye views [18, 39], that show detail in the area of current interest but preserve the context by also displaying an overview. In particular, displaying a hierarchical structure with a tree diagram and separate detailed views are very effective for end-users [65].

Another advantage of the two views is that they offer two different styles of interaction. The GLS view offers fine control over the layout and placement of objects. The operator view allows gross adjustments like changing the order of objects arranged in a row, or adding borders around some component of a display. If the “row” composition had been implemented by the user in the GLS instead of having it be a built-in operator, changing the order of children would have been much more tedious.

It is also much easier to add components at the operator hierarchy than in the GLS. For example, after creating a graphical display, we may decide that another component is to be placed between two others that have been arranged in a row. To do this in the GLS, many constraints would have to be re-positioned. In the operator view, though, we have only to add another child to the “row” operator.

The parameters of a basic operation can also be easily changed at the operator hierarchy, like the spacing between objects arranged in a row. In the GLS, that would involve editing the value of several constraints, but in the operator hierarchy we just need to edit the “spacing” attribute for the operator.

Similarly, there are other things better done in the GLS view than in the operator hierarchy. If

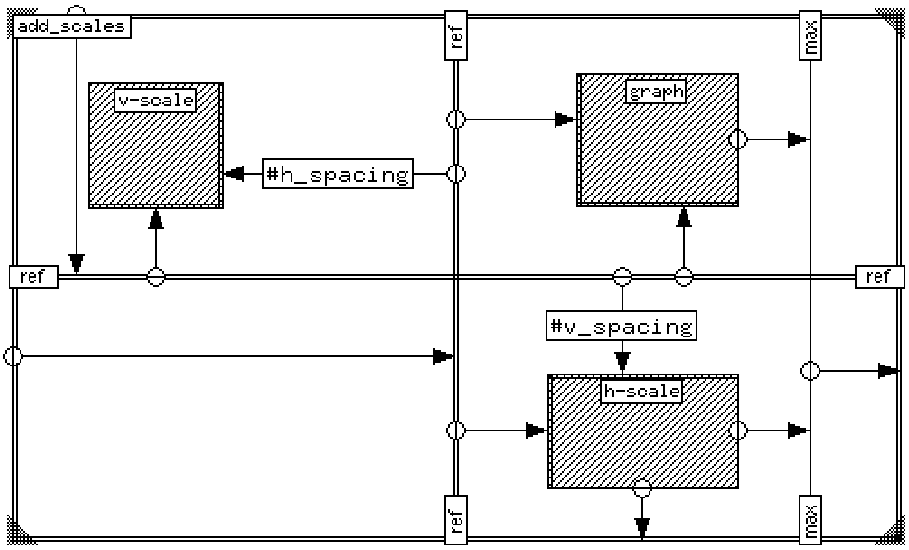

Figure 6.5: Creating a simple composition operator

the spacing between objects is non-uniform, it is not possible to do that in the operator view, but it is easily done with a GLS specification. Non-standard compositions are also done more easily in the GLS than in the operator hierarchy. For example, Fig. 6.4 describes a composition operator that puts three objects together, adding horizontal and vertical scales to a graph. This could be implemented by arranging the objects "v_scale" and "graph" in a row and then putting that in a column with "h_scale" with the right offsets, but it is more convenient (as well as more general) to do in the GLS view.

6.3 Creating New Operators

New operators can be created by the user and added to the library. The GLS notation presented earlier is easily extended to specify operators as well. This section presents this technique and an example of how this can be done.

6.3.1 Operators that take a fixed number of arguments

The parent-child construction can be used to create a simple operator. The arguments to the operator—the images it composes—are represented by the grey boxes of the parent object. As an example, consider the composition shown in Fig. 6.4. If we could implement this as an operator and use it in the hierarchy, the specifications would be much easier to understand.

Fig. 6.5 describes an implementation of such an operator. It is named "add_scales" and takes three arguments, the two scales (called *v-scale* and *h-scale*) and the graph. They are arranged so that both the graph and the horizontal scale have their left edges lined up. At the same time, the lower edges of the vertical scale and the graph are lined up. After the operator is defined and stored in a library, it can be used just like a pre-defined one.

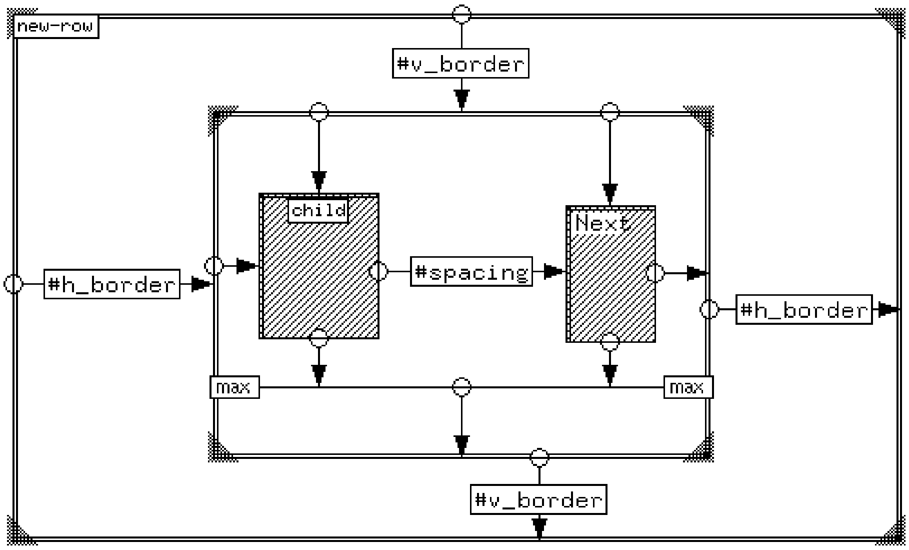

Figure 6.6: Creating an operator with a variable number of arguments

The parameters of the operator are specified in the same way as data arguments in a GLS specification of a display. In this example, the operator has two parameters, referred to as `#h_spacing` and `#v_spacing`. When this operator is used in a Pluto description, its attribute editor can be brought up just as for the pre-defined ones. This operator has two attributes, and the editor for it will have two fields. The labels for the fields will be the names used while defining the operator.

6.3.2 Operators with a variable number of arguments

Operators that take a variable number of arguments can be created by using the repeat frame notation. Instead of the objects inside the repeat frame being replicated based on either the size of the data stream or some fixed number specified by the user, it is the number of arguments this operator has when used.

Fig. 6.6 is an example of how such an operator can be created. We show how the equivalent of the "row" operator may be specified in this notation. The grey rectangle labelled "child" represents the image being composed. The distance between successive children is the value of `spacing`, and the resulting composite image has a border of `v_border` in the vertical direction and `h_border` in the horizontal direction. We also note that lower edge of the composite images is defined to be a distance of `v_border` away from the *maximum* of the lower edges of the composed objects. This is shown in Fig. 6.7.

The operator's three parameters (spacing between objects, and the two borders) can now be edited when this operator is used, by invoking the attribute editor just as in the earlier example.

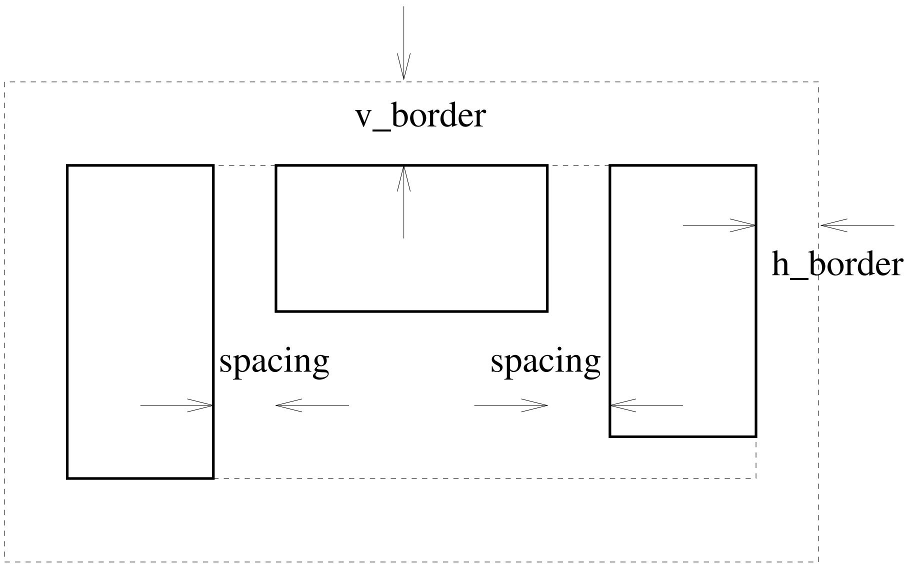

Figure 6.7: Result of the row operator

Chapter 7
Examples

In this chapter we present some examples of displays that can be created with Pluto along with the specifications used to implement them.

7.1 Napoleon in Russia

In [42], Marey describes a chart drawn by C. J. Minard in 1861 (Fig. 1.1) representing the march of Napoleon's army on Moscow in the winter of 1812. In one map, it shows the position and size of the army, direction of movement and the temperature and has been acclaimed as one of the best examples of graphical presentation of data.

Fig. 7.1 is a chart implemented by Pluto for the same data. It is made up of lines of varying width joined together (annotated with bars and a text tag for the numerical strength), and then overlaid on an image of a map. This graphic is placed above the chart for temperature and date.

The data for this chart consists of a sequence of records that contain the position, the strength of the army, the date and the temperature. The date and temperature information is not available for all the points. Instead of using line width to represent the strength of the army, we use rectangles whose height shows the strength of the army, along with a text label. These rectangles are then joined together.

The basic organization of the display is shown by the operator hierarchy (Fig. 7.2). A bitmap displaying the map of Russia ("map-image," simply a bitmap representing the rivers, their names and the names of cities) is overlaid with the lines representing the march of the army ("march-chart") to form the core of the display, which is then arranged in a row with a text string ("title") and the temperature chart ("temp-chart").

In Fig. 7.3 we present part of the Pluto specification to construct the chart. It describes the strength and position of the army. For each point (x, y) we calculate the points above and below it by n, the strength of the army. Each instance of the repeat frame is placed so that its left edge is at the x position of the previous point, and its upper and lower edges take the values of the rectangle representing the strength at the previous point.

The reference lines are placed at the three values of interest (x, y - $n/2$ and y + $n/2$). Now the three edges of the repeat frame and the three reference lines represent these values in the previous and the current iteration respectively. The rectangle in the middle with a height of "#n" represents the bar at each data point. The two lines (at the lower left and the upper left of the repeat frame) link up these rectangles. A couple of additional objects are also placed: a text object to display the strength numerically, and a parent-child object named "xref" to draw lines cross-referencing this chart to the temperature chart. The left, lower and upper edges of the "Next" object are set to the right values for the next data point.

The "xref" object adds lines to cross-reference the two displays (Fig. 7.4). This is implemented as a conditional object, since the lines are only drawn when a temperature value is present. When

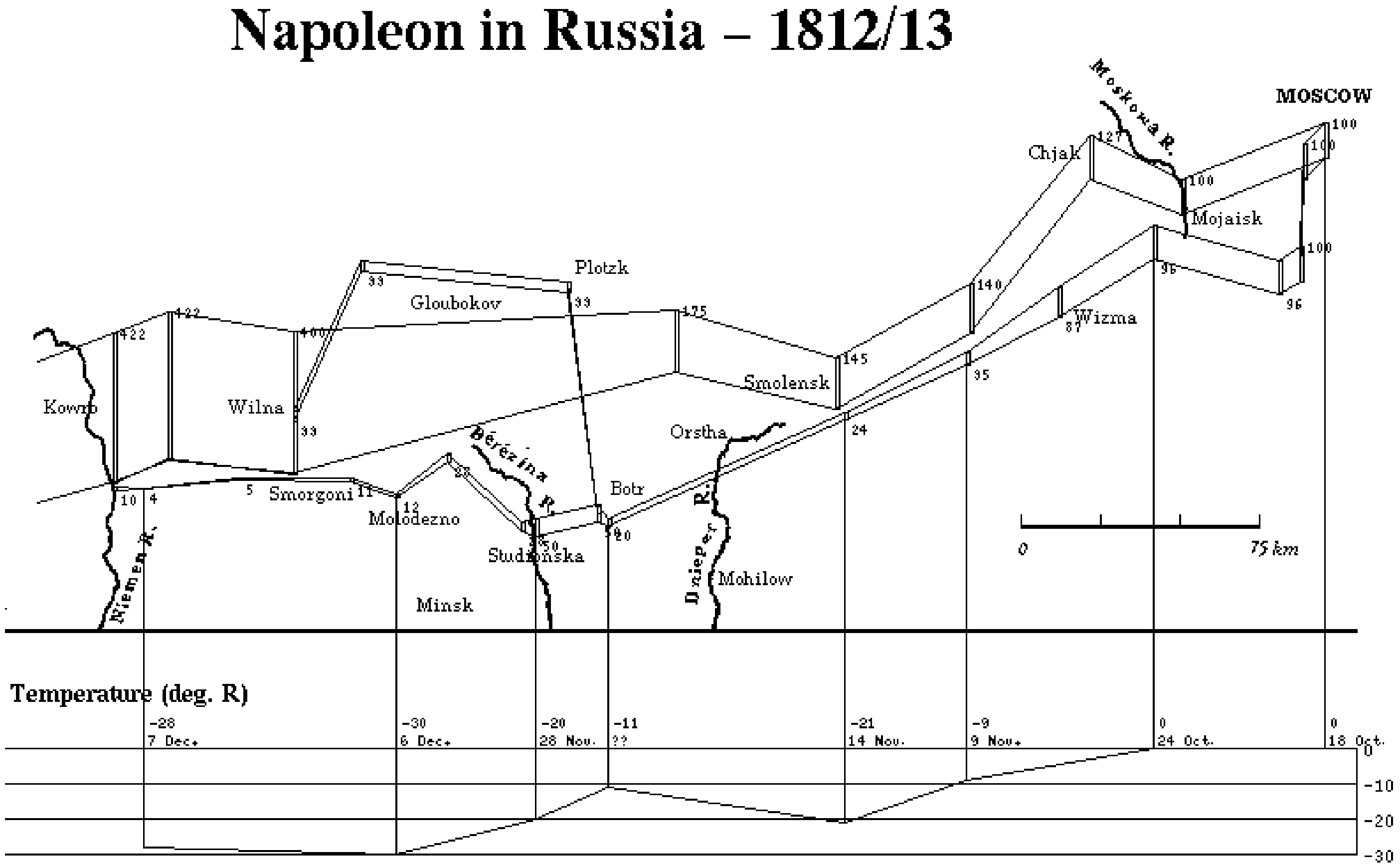

Figure 7.1: A graphic drawn by a Pluto specification

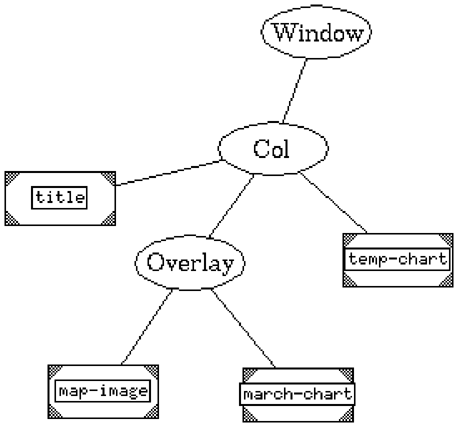

Figure 7.2: Operator hierarchy for Napoleon's march

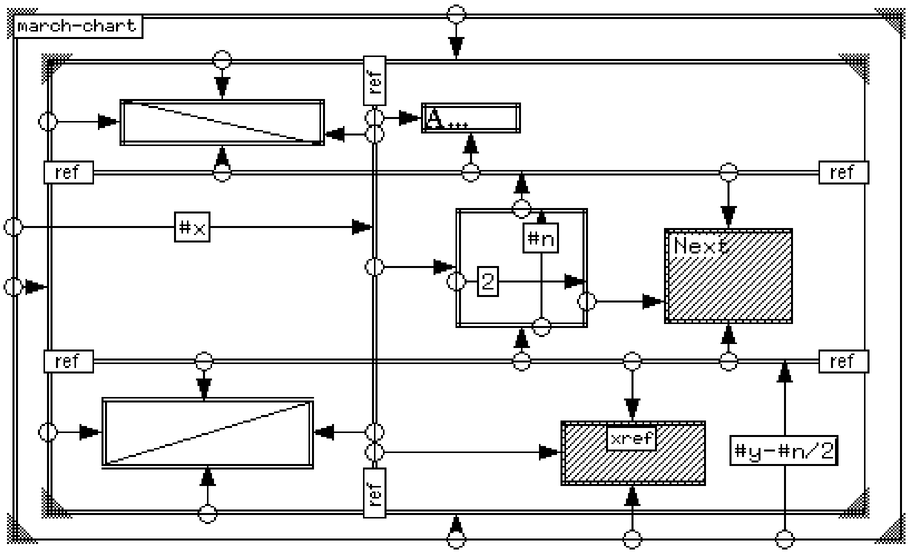

Figure 7.3: Pluto specification for the state of the army

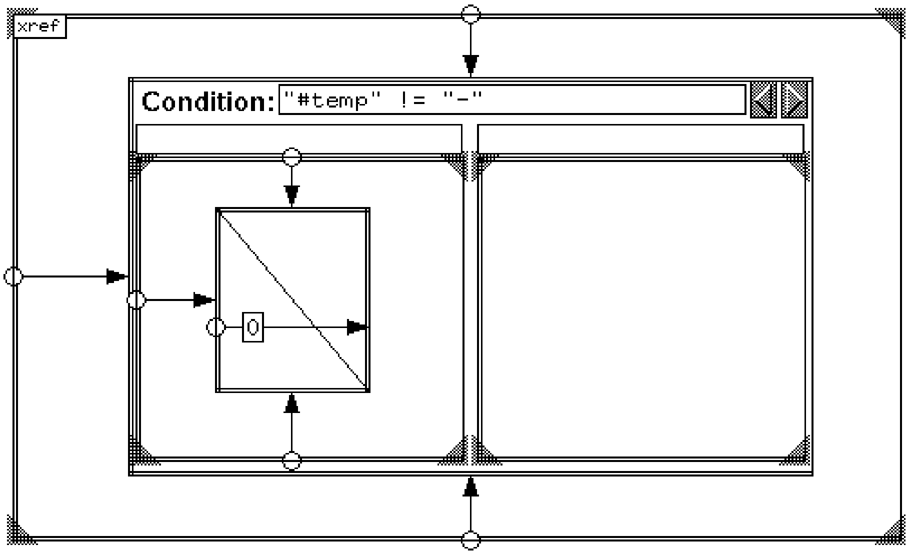

Figure 7.4: Lines to cross-reference between the two displays

such is the case (expressed by the condition “ "#temp" != "-" ”, i.e. the temperature expressed as a string is not just a dash), a vertical line is drawn from the bottom of the strength rectangle to the lower edge of the display. The temperature scale does a similar thing but upward, so that when the two displays are composed in a column these lines join together.

The temperature chart is specified in a similar manner to the march, with lines drawn from point to point. The date is specified by a text field. An additional text field shows the exact value of the temperature.

The temperature chart is overlaid with a scale (Fig. 7.5). The scale draws the horizontal grid as well as the labels. It illustrates a repeat frame with a fixed number as its argument, signifying that instead of taking the input from an external source, it is to be iterated a fixed number of times. Each instantiation of the repeat frame (and its children) can then use a value named “Serial_no.” This value is used for the labels of the scale. The attributes for the text display object are shown in Fig. 7.6 and shows how the value of the text label is calculated from the counter. In this case, the resultant strings are -30, -20, -10 and 0. The scale itself is made of horizontal lines that are spaced 20 units apart, with a vertical line to their right.

7.2 Histograms and Hanging Rootograms

We have already seen the Pluto specification for a histogram (Fig. 5.2). It consists of a repeat frame that draws rectangles that are 10 units wide, and whose height depends on `#value`. The “Next” box is connected to the right edge of this rectangle, with the result that all the rectangles created are lined up next to each other. They all take their bottom edge from the bottom of the repeat frame, all of which take it from the base of the display; thus all the rectangles' bases are lined up.

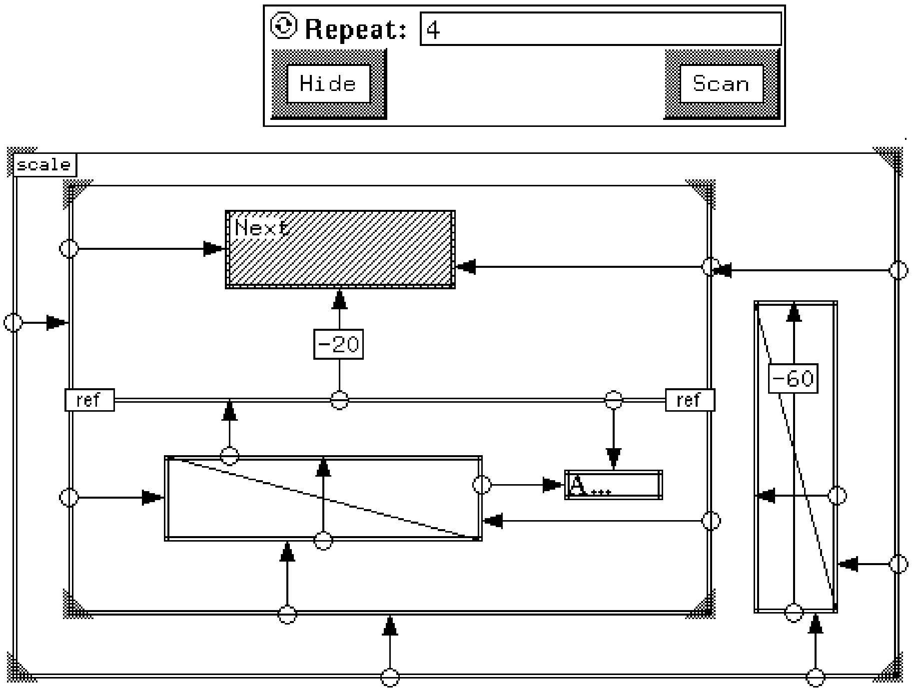

Figure 7.5: The scale for the temperature chart

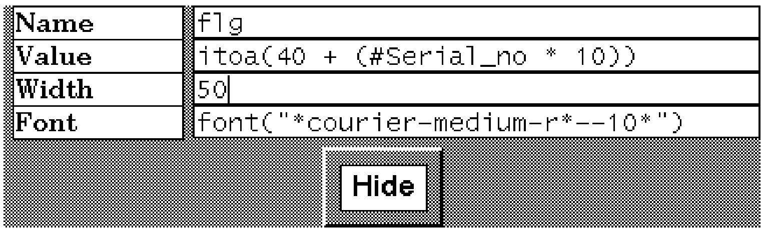

Figure 7.6: The value of the text labels in the temperature chart

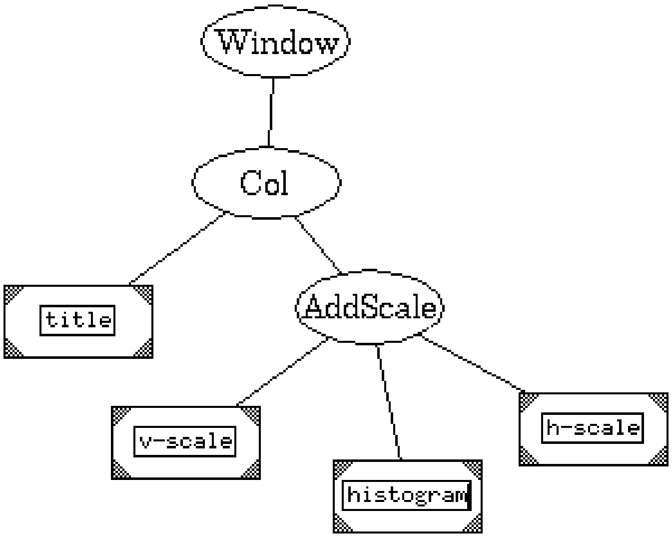

Figure 7.7: Operator hierarchy for the histogram

The basic histogram thus created is then composed together with scales and titles, to form the final display. To add the scales, the ternary operator presented in Fig. 6.5 can be used; the resulting image is placed in a column with a title string (Fig. 7.7).

The hanging rootogram was presented in Fig. 5.5. This is very similar to the histogram—the difference is that each rectangle takes its top edge from the reference function instead of taking its base from the base of the display. This specification is overlaid with a display of the reference function, and then combined with scales as in the histogram, and also with the sliders and text display fields to form the final display seen in Fig. 5.3.

7.3 Quartile plots

John W. Tukey in [69] devised this method (and other similar ones) of representing statistical quantities of a series of experiments. Each quartile plot (or *box chart*, as they were originally called) represents the statistical properties of one run of the experiment: the maximum and minimum values, the 1st and 3rd quartiles and the median (Fig. 7.8). The box charts are then put together, one for each quintuple of data, to form a "parallel schematic plot."

Each box plot can be represented by the GLS specification shown in Fig. 7.9. A rectangle 10 units wide is drawn with its upper and lower edges depending on the 1st and 3rd quartiles respectively. A horizontal line is drawn with its endpoints at the left and right edges of this rectangle, and whose vertical position depends on the median value. This box with line is drawn five units to the left (since the label for the constraint is -5), so that it is horizontally centered at the correct position. A vertical line (drawn to the right of the rectangle in Fig. 7.9) whose lower endpoint is the minimum data value, and whose top is the maximum, completes the box plot.

All the box plots are inside the repeat frame, and are drawn 20 units apart. Since the boxes are 10 units wide, this means they are separated by 10 units. A text label is also present inside the

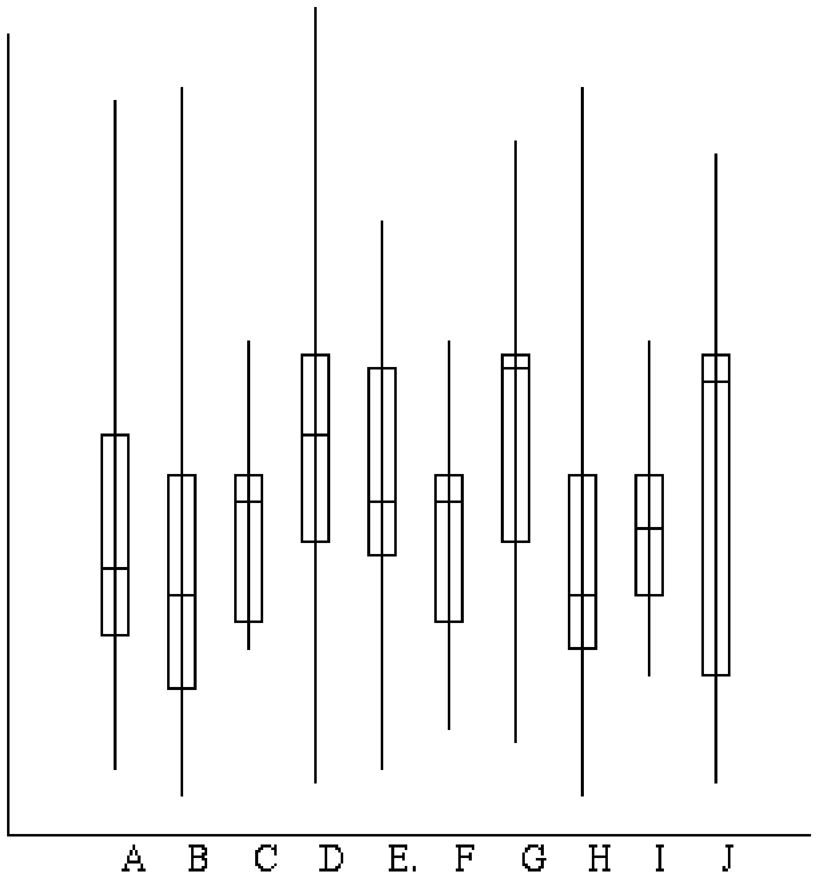

Figure 7.8: Tukey's Quartile Plots

repeat frame, drawn at the bottom of the display and at centerline of its box plot. This displays the value of the data field named `label` just below the horizontal axis for each box. (The axes are not shown; they are created simplay as two lines overlaid on the box plots.)

All of the other variations of box plots described in [69]—variable width box plots, notched box plots, etc. can likewise be easily specified in this notation.

7.4 Grid arrangements

One way to represent four-dimensional data is to use two dimensions to place the datum, and to draw a "whisker" whose length and direction represent the other two dimensions. This has an obvious interpretation in some data, like airstream flows in aerodynamic studies. When measurements are taken of flow velocities in an airstream of interest, the sensors are usually placed on a plane perpendicular to the mean flow direction. Each data point represents the position of the sensor, and the in-plane components of the velocity. Fig. 7.10 shows an example of such a display. These

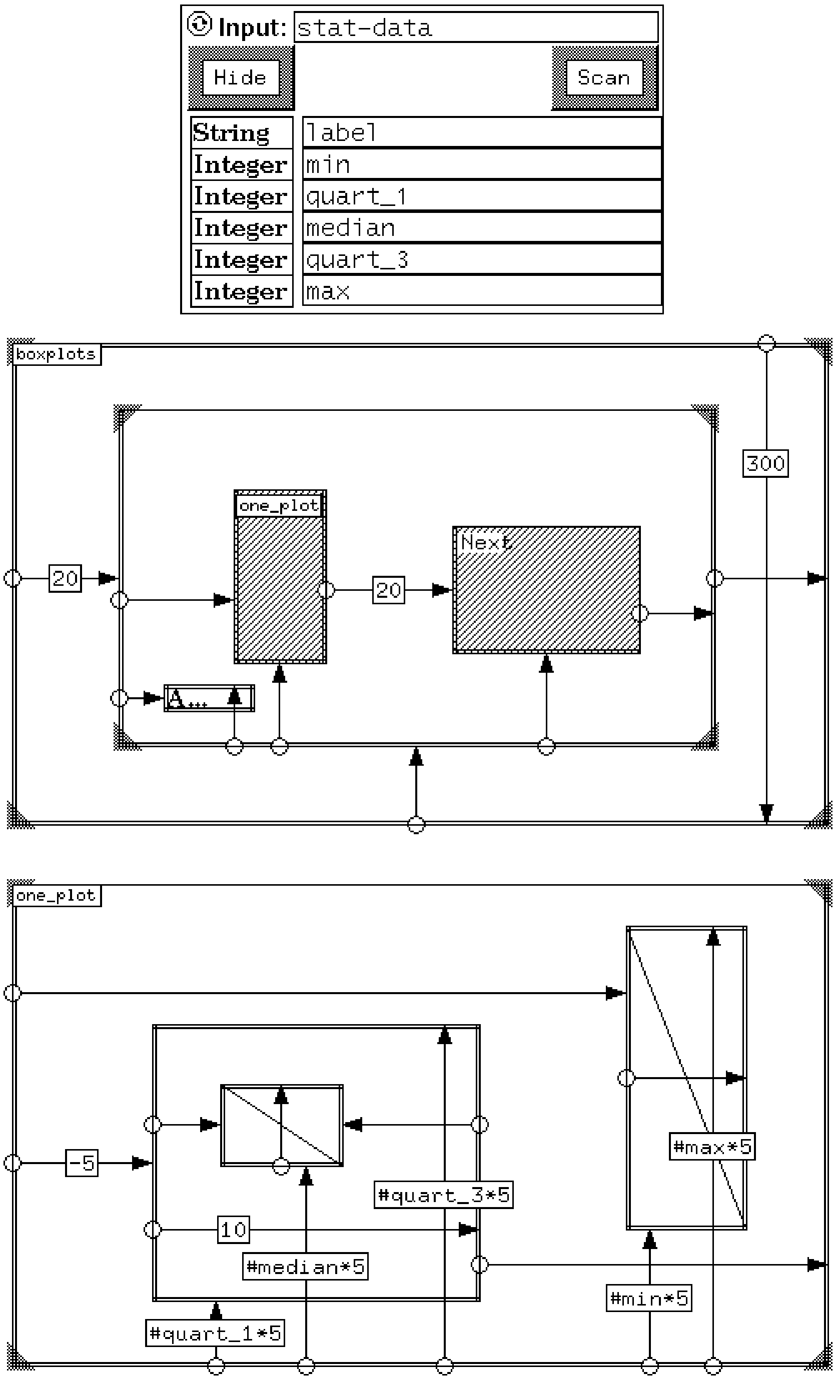

Figure 7.9: GLS specification for a quartile plot

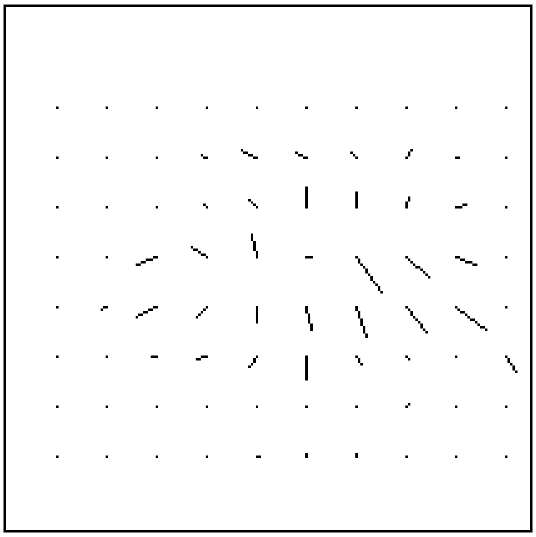

Figure 7.10: Four dimensional data—"whiskers" arranged in a grid

types of displays are useful for highlighting vortex flows in the stream.

This type of display can be implemented like a scatterplot, using two coordinates to explicitly specify the position of each whisker. Another way to describe this would be to consider the input as a stream that scans across the field like a raster. We now present this type of display.

Fig. 7.11 shows part of the Pluto specification that created the graph. The top frame shows the basic repetition structure. Each "Next" box is placed at the (*x2, y2*) position of the child object, "grid-elem."

The child object is the interesting one. At its heart is the conditional that handles the layout of individual grid point displays. The whisker itself is a line of the required size that is drawn at the position determined by the conditional, at its right top edge. The conditional's behaviour depends on whether its own left edge is beyond the extent of the grid square allocated to its display. Here n represents the number of elements in the grid horizontally. The spacing between grid points is 20; if the left edge of this conditional is in fact greater than the extent (that is, the next whisker would be drawn outside the bounds of its own grid square), the right edge is moved down one row and the left edge is moved back all the way to the left. Otherwise the right edge is over to the right by 20 units. This has the effect of placing the objects in a row until the right edge is reached, and then moving down by one unit and to the left edge, and continuing the process.

If the size of the field is not known, the value of n itself could come from user input, say from a slider. In that case, the user could interactively adjust the size of the field, and have all the grid points re-arrange themselves dynamically.

7.5 Interactive graph display

One powerful technique is to use a Pluto specification as the graphical output device for a general program. For example, there are many popular graph layout algorithms, each emphasising different

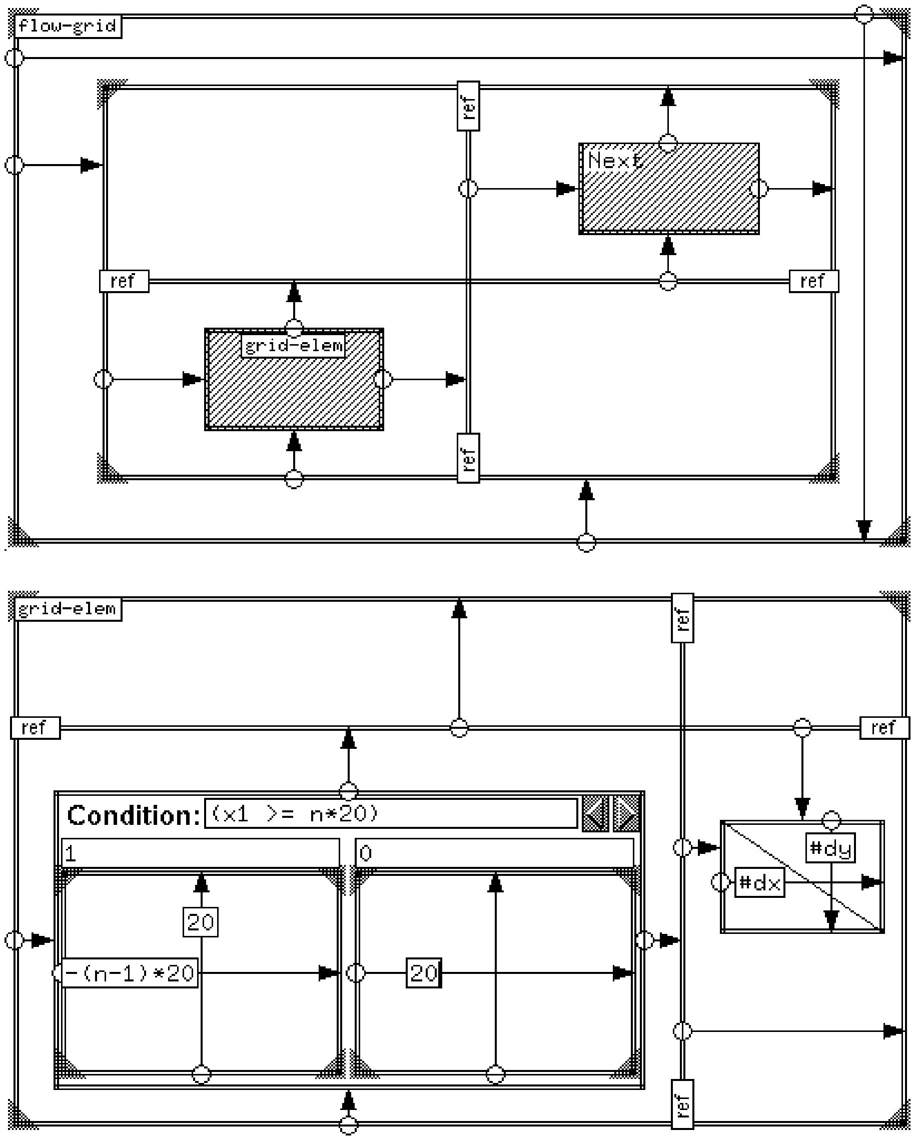

Figure 7.11: Pluto specification for a grid layout

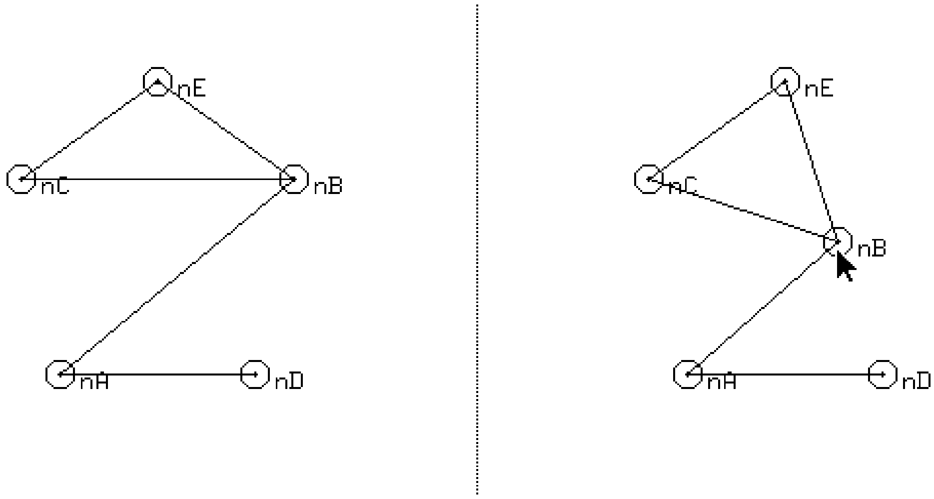

Figure 7.12: Dynamic graph display using Pluto

aspects of presentation. However, the general problem of optimal graph layout is intractable, so many heuristics and interactive techniques are used. A complete treatment of these can be found in [22].

One way to approach the graph display problem is to use an appropriate layout technique to place the nodes and draw the edges as straight lines between them. Pluto can now be used to display the resulting graph. However, instead of establishing static positions, the Pluto specification can take the layout created to establish initial positions for the nodes; the user can then freely drag the nodes around, and the edges will track the nodes. The user can thus explore the structure of the graph, or dress its appearance to suit his display needs.

In Fig. 7.12 we see an example of this method. On the left is the graph as arranged by a layout algorithm and drawn by Pluto. On the right, we see the same graph, but now the node marked "nB" is being dragged by the user. The edges are automatically redrawn to follow the movement of this node.

The input to the display has two parts:

- the node list, with the names and x and y positions of each node
- the edge list, specifying each edge with the names of the two endpoints.

Tables 7.1 and 7.2 present the input used for the graph display.

The display of the graph is formed by overlaying two displays, one for the nodes and the other for the edges. The node display is straightforward, with a repeat frame that draws an image of a node attached to a text label, and drawn at the specified position (Fig. 7.13). Since we want the to be able to drag the nodes in the display, we prefix the expression for the constraints with "!" (which indicates that this an initial value, and the user is then allowed to control it). Each node is also given a Pluto name depending on its label, `Node#label` implying that the name of the bitmap object is "Node" appended with the data field called "label" (Fig. 7.14).

Node	x position	y position
nA	100	100
nB	220	200
nC	80	200
nD	200	100
nE	150	250

Table 7.1: Node list (with positions) for a graph

From	To
nA	nB
nA	nD
nC	nB
nC	nE
nE	nB

Table 7.2: Edge list for the graph

The edges are just straight lines drawn from the position of the source to the position of the destination. We have named each node depending on its label, so we know the positions for this line (Fig. 7.15). The end-points of the lines have floating constraints, not attached to any object in this GLS view but using the positions of the nodes directly in the expression. Now if a node is dragged, the lines drawn from it will automatically track it.

7.6 Histogram with a "significant grid"

Tufte [67] shows some more interesting ways of presenting data. One of the techniques he demonstrates is to not use a grid over the entire chart; instead, only significant points of interest are shown. This is best illustrated with a slightly modified histogram, which is drawn without a grid or scales; instead, thin horizontal lines are drawn from the tops to the y-axis. Instead of explicitly drawing the axis, the values of the ordinate are shown instead, and form an implicit axis (Fig. 7.16).

This is a histogram augmented with a line and a text label. The value of the text label depends on the data field. Fig. 7.17 shows the GLS specification. One change that has been made is that instead of the bars abutting each other, they are separated by a small space, 7 units wide. Another slight change is that instead of using one rectangle for each bar, two are used, separated by a thin space. Since the rectangles are one unit wide, they appear as thick lines.

7.7 A dot-dash plot

An alternative to the scatterplot is the *dot-dash* plot. Instead of drawing axes to the scatterplot, short dashes are used at the position of each data point. This display simultaneously shows the joint distribution as well as the marginal distributions (Fig. 7.18). Several such dot-dash displays

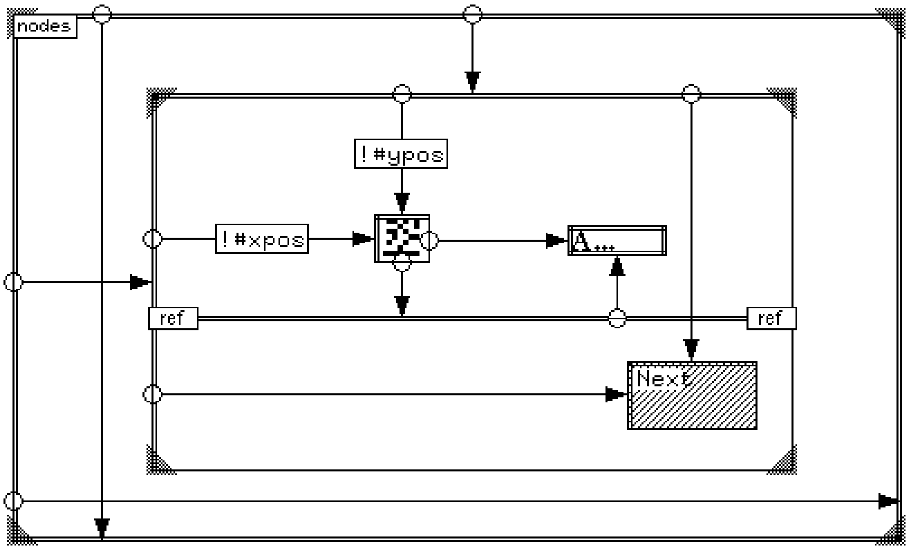

Figure 7.13: Specifying nodes that can be dragged

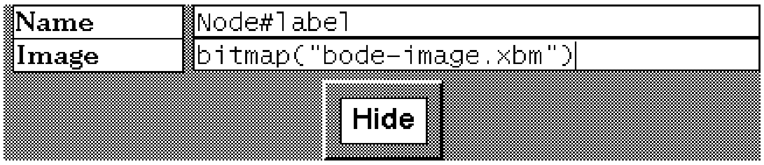

Figure 7.14: Attribute editor for the node

---

can be combined to form the rugplot display (Fig. 2.2), which can display pairwise correlations between several variables, with the marginal distributions providing a link between them.

This is a basic scatterplot with the addition of the marginal displays. The repeat frame has three elements: a rectangle that is the *dot* and represents the bivariate plot; and two *dashes* that show the marginal distribution, one per axis. The rectangle is placed at the point (x, y) and the two short lines are placed with one coördinate at the relevant axis and the other taken from the dot (Fig. 7.19).

7.8 A two variable line trace

Another technique is to plot two time-dependent variables directly against each other, and use the time values as an annotation. Fig. 7.20 is an example of this kind of display, showing how inflation and unemployment change with time. The years for which the data points are plotted are shown in numbers next to the points.

This chart is a sequence of lines, each drawn from the position of the previous point to the

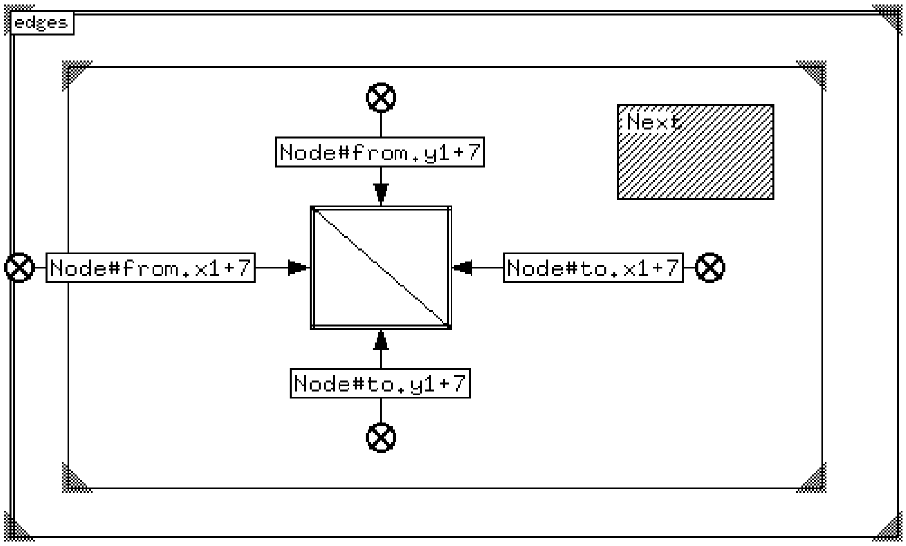

Figure 7.15: Specifying the edges

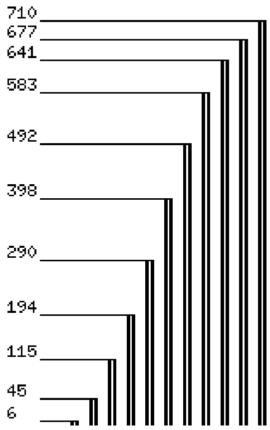

Figure 7.16: Implicit axes with the “significant grid”

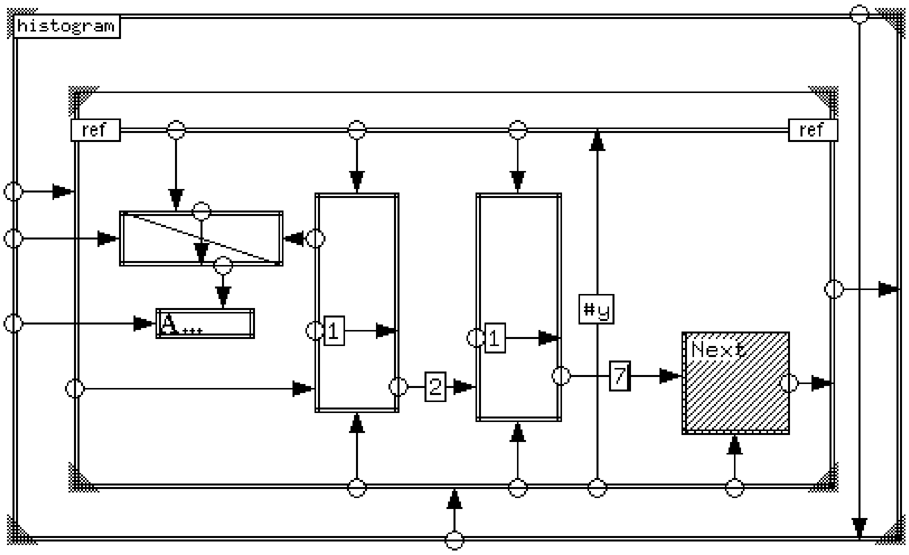

Figure 7.17: A Pluto specification for a histogram with a significant grid

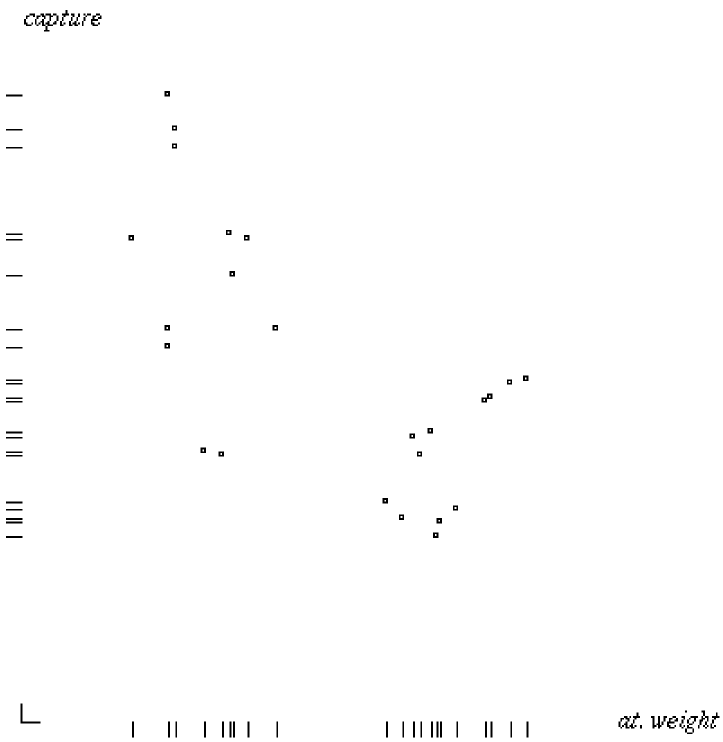

Figure 7.18: A “dot-dash” plot

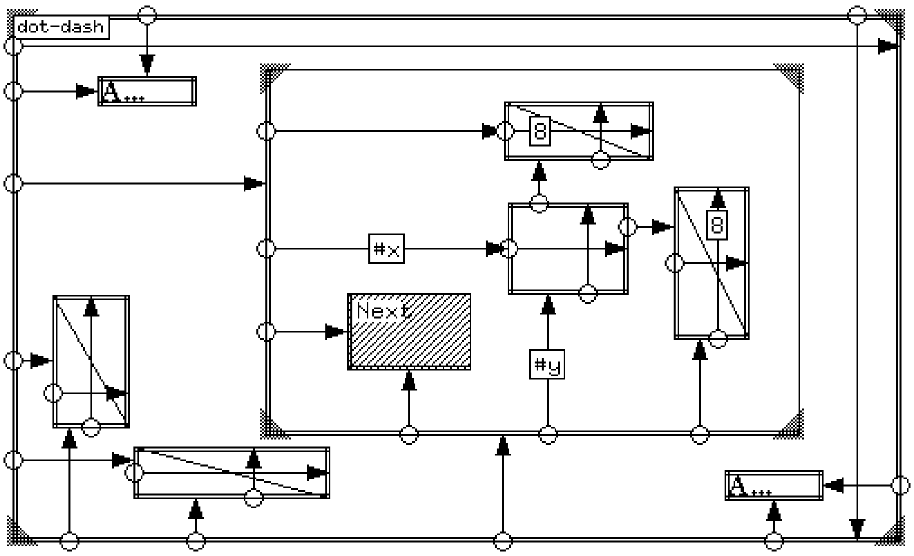

Figure 7.19: A Pluto specification for a dot-dash diagram

present one (Fig. 7.21). The input data values (#unemp and #infl) are both multiplied by 5 to scale the drawing to the required size. The text label, which will display the year, is drawn at the present position.

This display is overlaid with the scales to form the final display.

7.9 Using data as marker

Another use of the implicit axis is shown in in Fig. 7.22. Here a function is plotted, in this case $y = 1/x$. However, instead of plotting the points with some sort of marker, the value of the abscissa is used. Instead of drawing an explicit ordinate axis, the technique in Fig. 7.16 is used again, except that only a subset of labels are selected to be drawn in order to avoid overlaps. Small markers are drawn at the y values of the data points, simultaneously showing the position of the axis and the marginal distribution. The problem of selecting the best set of labels to draw is hard; instead, we allow the user to control each label directly by clicking on the markers, which are actually two-state toggles. This reduces the clutter on the axis.

The basic specification for this example is quite simple (Fig. 7.23). A text label is drawn at the position specified by the data. To reduce clutter, we use a reference line whose height is the value of the data field "yval" and use it to set the positions of the other objects, a text label that displays the actual value of the ordinate and a toggle. We have chosen to reduce the clutter of overlapping labels in the final display by providing a toggle for each label. The image of the toggle also serves to display the position of the axis and the marginal distribution of the y values. The image of the toggle also serves to display the position of the axis and the marginal distribution of the y values.

Fig. 7.24 shows the attribute editor for the controlling toggle. It is named "v_button" and has an initial value of 1, i.e. all the labels will be displayed to start with. The text label for the ordinate

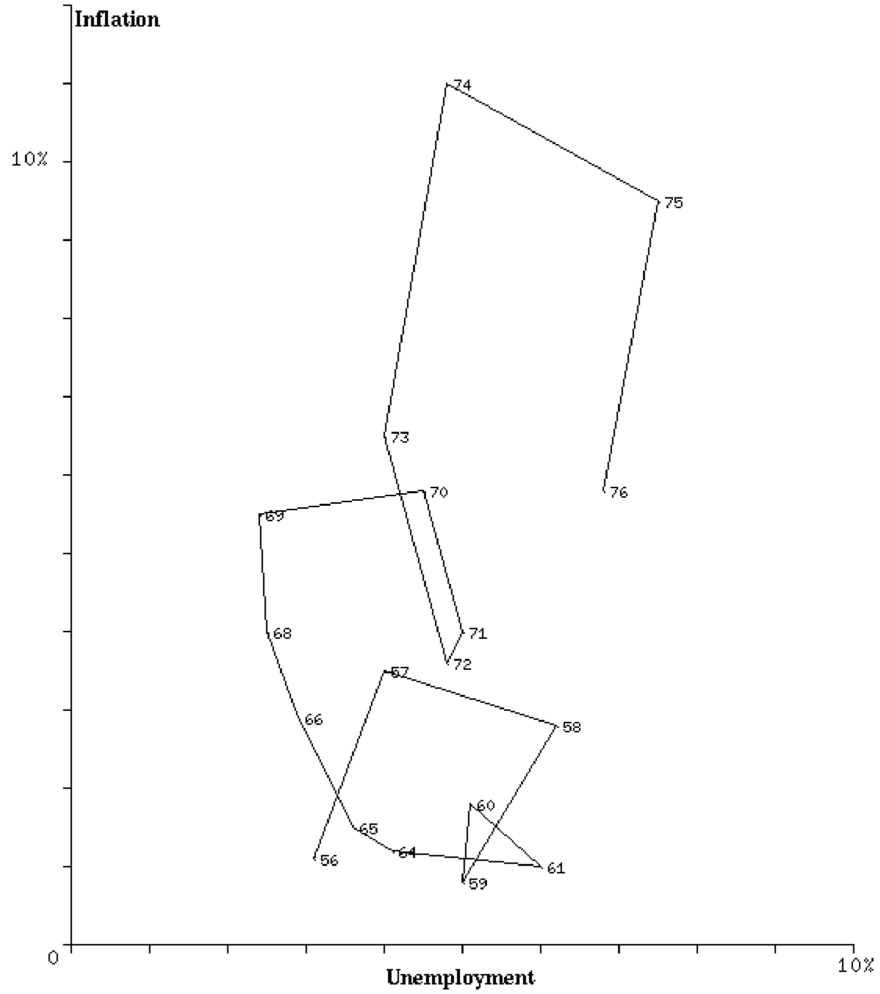

Figure 7.20: Unemployment and inflation in the U.S., 1956-76

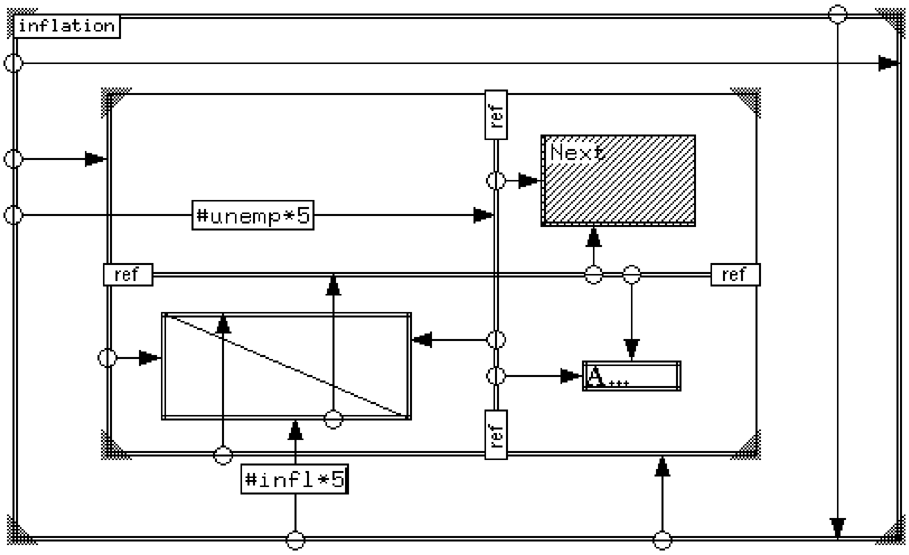

Figure 7.21: Pluto specification for a two-variable trace

is specified in a conditional, so the label will only be drawn if its toggle is “on.”

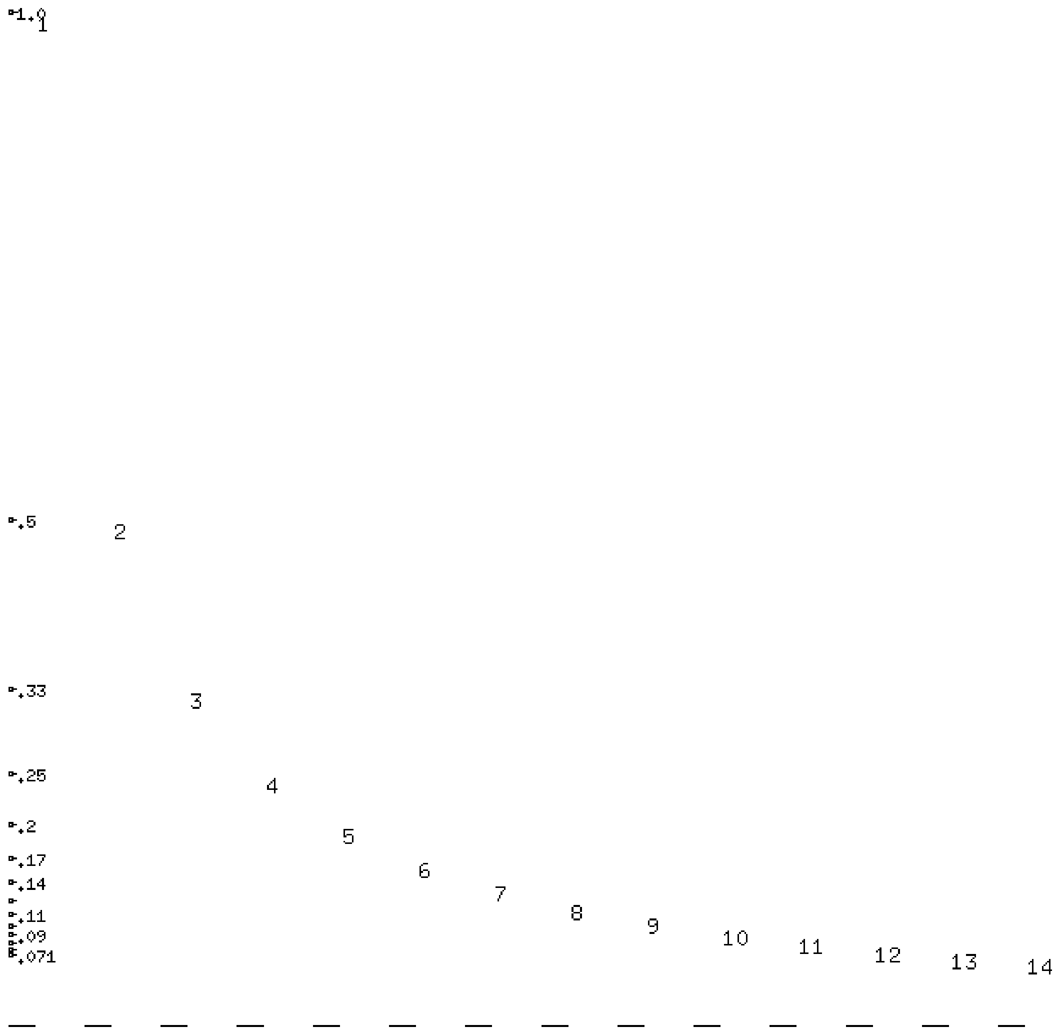

Figure 7.22: Using the data itself as a marker

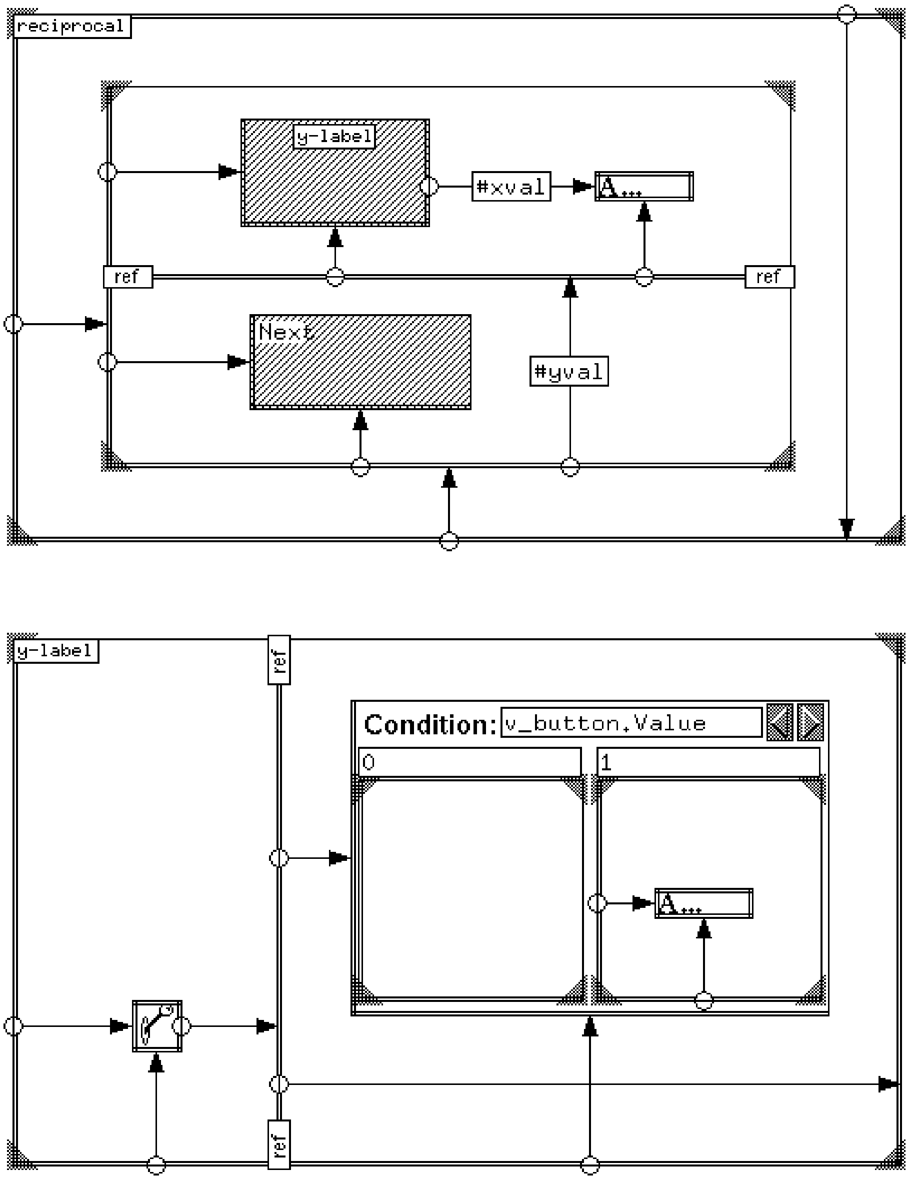

Figure 7.23: Pluto specification to draw a function with data markers

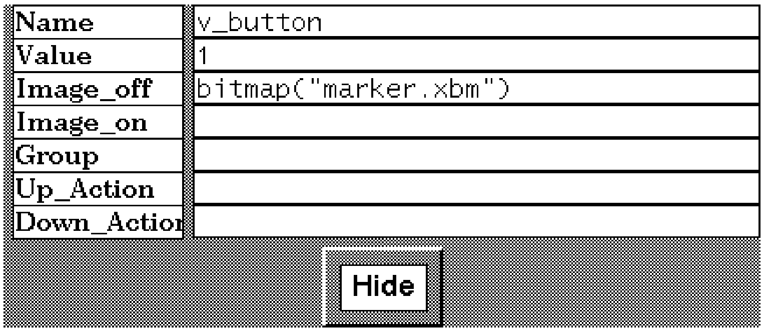

Figure 7.24: Attribute editor for the toggle

Chapter 8
User Experience

Preliminary user studies were carried out with five subjects. Although simple time on task measurements were taken, these studies were primarily designed to obtain qualitative information regarding the strong and weak points of the system for a range of real users. Since there are no other systems presently available that offer a functionality similar to Pluto, we could not do a direct comparison. Subjective impressions of the users were recorded in two main areas:

- Modifiability—given a specification for a display, how easy is it to modify it? This is of primary interest to the *workers* (Chapter 1) in the population. A worker will pick a display specification from the library, and modify it to fit the task at hand. How much work is it to perform such modifications?
- Completeness—can useful displays be created easily? This is of primary interest to the *tinkerers*. When a user wants to design a new display, can the system provide the necessary support?

8.1 Experimental Results

Of the five subjects, two (referred to here as A and B) were undergraduate students in biological sciences. They had had no experience with computers, except for occasional word-processing etc. The subject referred to as C was a professional researcher in evolutionary biology. D was an experienced programmer, but with no experience with writing graphics programs; and E was a graduate student in computer science, and an experienced graphics (Xlib/Xtoolkit) programmer.

A brief tutorial introduction was given to all the subjects, and then they were allowed to familiarise themselves with the system. This was done by letting them experiment freely with the GLS notation for about 15 minutes. At the end of the familiarization period, two basic tasks were set:

1. Given a Pluto specification for a plain histogram (such as the one described in Fig. 5.2), modify it to display an additional data field. This field was to be represented by the width of the histogram bars. In addition, the bars were to be separated by 10 units, and text labels were to be added to represent the height of each bar.
2. Implement the display shown in Fig. 7.20. This plots inflation against unemployment for a period of 20 years. The years are indicated by annotating the trace with the years. (They were not required to add the scales or titles.)

During the familiarization period, the subjects were allowed to ask questions about any unclear aspects, or to point out deficiencies in the manual. (This feedback was used to refine the manual.) Once the tasks were started, there was no further intervention.

Subject A

This user had no previous experience with computers or programming. After the familiarization period, the specification of the histogram was shown, and the first task was described. It took the user a total of 15 seconds (with two "Try It" actions) to modify the widths of the bars to reflect the additional data, and another 5 seconds to change the spacing.

On the second task, this user faced a conceptual hurdle. Breaking down the organization of the chart to lines drawn in sequence took about 5 minutes of thinking. It took an additional minute to realise that one end-point of the line was fixed by the previous iteration, but the other was fixed by the data; after that, it took about 25 minutes to describe the graphic. Some more experimentation was required to get all the details of the display right.

The mental model of the "Next" object seemed easy to grasp, as long as there was only one other object begin repeated. In both the tasks, it was regarded as being a representation of the line or rectangle being repeated. Adding the textual annotations caused a slight amount of confusion: it was then unclear as to what exactly the "Next" represented. This was cleared up by experimentation: selecting "Try It" again showed the realised behavior of the specification. After that this user was comfortable with the notation, and decide to go back to the histogram and add a slider to control the vertical scale of the display.

Subject B

This subject was also a biology undergraduate. The first was task was considered easy, and was completed in five minutes. Intermediate stages were displayed four times before the display was completed.

The second task posed a significant difficulty. This user could not create the abstraction of the loop structure, even though in the earlier task one had been successfully modified. The greatest trouble was felt in positioning one end-point of the line at the previous point's location. After fifteen minutes of experimentation and several failed attempts, the subject gave up and the solution was described. The user was able to understand the solution almost immediately, and was then able to modify it by adding the text labels for the years.

B also said that using and experimenting with the system was fun. The immediacy of the "Try It" operation encouraged experimentation and exploration, which this user took full advantage of. This user also tried various modifications to the basic histogram display, like decorating the bars with numeric labels denoting the actual values.

Subject C

Subject C, the research scholar, had had considerable experience with commercial graphing and data analysis packages, but no programming experience. Due to time constraints, this subject could not attend a tutorial session, and could not spend time familiarising himself with the system either. In spite of that, the first task was considered very easy, and was completed in the time it took to change the expressions in the constraints (a total of about 15 seconds). Related tasks such as changing the scales and annotating the graph were also accomplished in under one minute. In all of these, the first "Try It" attempts were completed displays.

The task of drawing the connected lines presented an unusual difficulty: in the horizontal direction, the user had no trouble creating the correct constraints. However, joining the lines

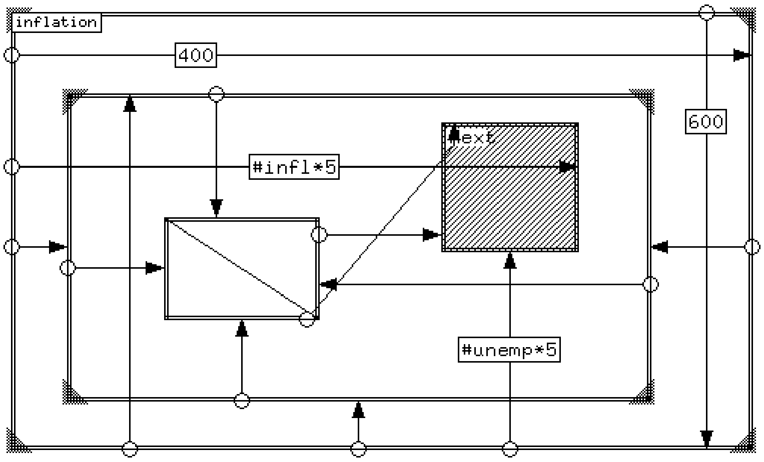

Figure 8.1: Specification for a line trace as drawn by subject D

vertically caused more difficulty. In particular, instead of establishing the end-point of the line at the position specified by the data, the length of the line was set to (x, y). This mistake was not noticed due to a combination of an unspecified attribute and a data value. It was suggested to the subject that he start again from the beginning. This time, selecting "Try It" at an intermediate stage immediately revealed that the lines were connected end-to-end horizontally but not vertically, and the specification was soon completed. The time spent on this task was about 15 minutes. Upon completion of the task he remarked that since the graph to be drawn was not a function (in the strict mathematical sense, that is, not a one-one relationship from unemployment to inflation) he expected the task to be hard, but could see that in fact it does not make a difference.

This user was the closest in terms of the target end-user group—technically sophisticated, but not a programmer. It is encouraging that even with almost no instruction, this user was able to modify pre-existing specifications easily, and could also describe new displays.

Subject D

An experienced programmer, subject D had no trouble with the notation. The concept of the repeat frame as a recursive object was easy to understand and use. On the first task, this user succeeded on the first attempt. Adding text labels took a little longer, but was not felt to be hard. This task was completed in about 30 seconds.

On the second task, some confusion was felt as to exactly what the "Next" box represented. In the histogram example, the box was felt to be the next rectangle, instead of a virtual object, and this did not pose a problem. It did cause some confusion for the linked lines exercise though. Once this was cleared up, this user in fact implemented the display in a novel manner: constraints were drawn to all four edges of the line object from the repeat frame, and then the "Next" box was constrained with the data (Fig. 8.1) [1]. The total time spent on this display was about 20 minutes.

This user was familiar with gnuplot, the graphing utility. At first, there was some resistance

[1] Actually, this specification is not quite correct; it draws a spurious line segment at the beginning.

to the suggestion that Pluto could be used as a general-purpose graph drawing system. Once the ease of modification of pre-defined display techniques became apparent, however, the subject expressed enthusiasm for the notation. The ability to fine-tune the display (for example, changing the displacements for text labels and to adjust spacing and scaling parameters) was felt to be the biggest improvement.

Subject E

This subject was a graduate student in computer science with extensive graphical programming experience. In some respects, being used to describing dynamic displays and user interfaces in a conventional programming language (C++), this was the most interesting subject. How this user approached a graphical specification was interesting.

The first task was considered easy, and did not present any challenges. The total time taken to design the display was about 15 seconds.

One of the difficulties felt by this user was in leaving edges unconstrained. The double-line behavior was felt to be useful, but not visible enough. Since Pluto tries to find sensible defaults for unconstrained objects instead of aborting the "Try It" function, some confusion was caused early on. It may be that experienced programmers expect error messages from the compiler instead of defaults that are "almost" right. Once the user realised that any single line boundary is potentially a mistake, progress was faster. The lines were joined up without much trouble. Realising that one end-point of each line relative to the previous line, but the other depended on the data in absolute terms, the display was completed. The total time taken for the basic display was about 15 minutes.

This user was familiar with graphical toolkits, and remarked on how much easier it was to describe a dynamic display with Pluto. This user spent the most time with the system, continuing to experiment even after both the tasks had been completed. The support provided by Penguims was also found to be interesting, and many of the examples presented in Chapter 7 were tried out. After about an hour of experimentation, E decided to use Pluto to create displays for E's dissertation research.

8.2 Summary

It can be seen that naïve users did not have any trouble understanding display specifications, and were able to make small customizations. One of the undergraduate subjects (A) could also create a new display from start to finish, although the other one had some trouble with the abstraction. The research scientist (subject C) was technically more sophisticated, as well as having strong mathematical skills and was able to handle both tasks quite easily.

It is apparent that the repeat structure in particular may be hard for some users to master. However, it is encouraging that even among the non-programmers, two out of three of the subjects could create a display using the construct, even with only a total of about an hour of instruction and experience.

The experienced programmers of course did not have any trouble with the recursive notation, although expressing it graphically took some time. Both of them were able to use the system quite effectively. User D expressed satisfaction at being able to perform some fine tuning that his conventional tools did not offer. User E remarked that using Pluto was much easier than using the X-windows libraries for rapid prototyping of interfaces.

Our experiments with users indicate that this notation can help end-users design and develop interactive displays. Although some users will not have the sophistication to use all the features of the system, they can still use it effectively by modifying displays that other more experienced users have created. Experienced programmers can also benefit from this as it reduces the tedium that is a large part of writing graphical interactive programs.

Chapter 9
Conclusions and Future Work

We have presented a notation and a system to allow end-users to be able to specify interactive displays easily. Our experience with the system, and observing non-programmers using the systems has provided us with some valuable insights.

9.1 Implementation Status

A prototype implementation of the Pluto system has been completed, although some of the features have not been included, *viz.* the support for the pre-defined libraries. It runs on a variety of Unix systems running the X11 window system [55]. The system is implemented in about 12,000 lines of C++ using a high-level user-interface toolkit developed at the University of Arizona, called Artkit [23].

9.1.1 System organization

The system is implemented as two programs. Users of this system deal mainly with Pluto itself, as an intelligent visual program editor. It is responsible for providing semantic feedback during object creation and editing. For example, it forces objects to stay inside the bounds of their enclosing objects, and only allows constraints to be created between compatible objects. Since it is implemented in C++, it is extensible at that level. All the objects in a visual specification are C++ objects, and they can be easily supplemented by using the object inheritance offered by the language. If any extensions are made to the underlying system (Penguims), compatible ones can be made to Pluto to take full advantage of them.

The other program is called *ppc*. A specification can, of course, be executed from within Pluto by invoking the "Try It" action. Once a display has been created, though, a user does not need to start up Pluto to run it. Instead, ppc can be invoked to run it. To the user, it appears to be a system that interprets the saved Pluto specification. The user can also execute the name of the saved file directly—the `#!` interpretation facility of Unix systems then automatically invokes ppc.

Ppc combines the Pluto specification with the data to be visualized. As has been mentioned earlier, the specification is structured as a tree with the "Window" operator at the root. This tree is traversed in a depth-first manner, and each object writes out Penguims code for all the constraints it uses. For each "repeat" frame in the spec., the children are repeated as many times as required. At each iteration, the values in the symbol table are changed to the current datum, so that any data field names used in constraints can be resolved.

When the complete specification and the data have been translated to Penguims code, the Penguims interpreter is invoked to actually implement the display.

9.1.2 Performance

Running on a SPARCstation IPC and X-Windows, the system can handle displays of a practical size. For instance, a hanging rootogram specification with three user-controlled parameters and 25 values can be fully drawn in under one second, from selecting "Try It" from the menu to having the complete display come up.

The interaction speed itself is a function of the underlying constraint engine, in Penguims. In the above rootogram example, the rectangles move smoothly based on user input for up to about 25 bars. This includes re-drawing the standard function and updating the text display labels for the three controlled values.

9.2 Limitations

There are some inherent limitations to this notation, although they do not seem to detract significantly from the utility of the system. For simple displays like unadorned scatter plots or histograms, specialized software like common charting programs are easier to use. In many cases, a quick impression of trends may be sufficient, and it may not be worth the effort of specifying a display in Pluto from scratch. However, quite often the initial cursory display reveals interesting trends that need to be explored more deeply, requiring a more powerful tool. Indeed, with a good library of commonly used displays, Pluto can be as easy to use as a data-graphing tool. The specialised data display programs can be augmented with an extensible tool like Pluto, allowing that more detailed exploration without the need for writing writing another specialised display system.

One of the drawbacks revealed in the user tests is that the system provides defaults for unconstrained attributes silently, that is, without a warning message. Some users find this confusing; we are now adding warnings when unconstrained edges are found while trying out a partial specification.

Another limitation may be that the notation is inherently rectilinear. Other forms—circular displays, etc.—cannot yet be represented. For instance, it is hard to represent a pie-chart in this system.[1] For the large majority of displays, though, a rectilinear system is adequate. Furthermore, the incorporation of geometrical transformations, especially rotations, at each step of the hierarchy (as 3D graphical toolkits like PHIGS provide) will provide support for non-rectilinear displays as well.

9.3 Summary

While visual languages for general purpose programming have many limitations, we find that in the graphical application domain they are natural mode of specification. By incorporating interactors (objects encapsulating appearance as well as behavior and ability to accept input) into the displays, an exploratory approach to data comprehension is encouraged.

Visual layouts and dependencies draw upon the users' experiences of the physical world, and greatly increase understanding. While textual languages are more compact and general purpose, they require their users to have a grasp of details like syntax etc. which is an undue load on the occasional or novice user.

[1] However, to quote Tufte: "Given their low data-density and failure to order numbers along a visual dimension, pie-charts should never be used" [67].

Due to the strong correspondence of the objects of the language with the objects of the application, this method is especially suitable for novice users. The success of WYSIWYG systems among end-users is well-known. We find that extending the WYSIWYG concepts with some support for abstractions greatly increases their usability and power, while not significantly reducing their accessibility to the end-user who is familiar and comfortable with abstract representations of data.

At first, there were some doubts as to whether non-programmers could readily grasp concepts like loops and recursion, or abstract representations for objects that do not yet exist. We find that scientific users have no difficulty with these concepts, and can in fact create useful displays.

The notation, in spite of some limitations, is powerful enough for the description of many (if not most) displays used by users.

9.4 Extensions

Some of the limitations of the present system can be significantly reduced with the following extensions:

Non-rectilinear displays: we need to develop a notation that can represent non-rectilinear relationships to end-users. Alternative coordinate systems may be one means of representing these. In particular, a polar coordinate system can represent objects like pie-charts.

Geometrical transformations: to make the hierarchical organization more powerful, transformations can be applied at each level of the hierarchy. Other systems that offer transformations (like the PostScript language and PHIGS) have been very successful. In particular, since with graphical displays, screen space is always a scarce resource, a complete implementation of recursive displays can be specified, extending the automatic scaling described in [45]. (A limited form of scaling can be performed in the present design by using proportional reference lines.) Another application of scaling would be in fish-eye views, where the interesting part of the display can be displayed in an enlarged and detailed view. The perspective wall display technique (Fig. 2.5) can also be simulated with support for scaling. However, it is not yet clear if hierarchical coordinate systems will be successful with end-users.

An extension of the geometrical transformations described above would be to include non-affine transformations. Very little work has been done with displays using them, chiefly because computer-assisted display tools have only recently become popular. In the past, when displays had to be drawn by hand, the computational effort involved precluded any experimentation in this area. However, now that powerful machines are available to end-users, we may be witnessing the arrival of a new class of innovative display and design techniques.

Appendix A
Pluto: Other Details

This appendix presents some details of Pluto that were not covered in the main chapters.

A.1 Display Framework

All objects are represented by their bounding boxes. The edges of bounding boxes can be related by constraints. The destination edge of a constraint is set to be the source edge added to the value of its expression. For each object, the size and placement is specified with constraints; the other attributes are specified by an attribute editor that can be hidden or displayed on command.

Every object is assigned a parent when it is created. At the root of this hierarchy is the *window*. In the operator tree, the parent/child relationship is explicitly visible. In GLS views, the place an object is created specifies its parent—the closest enclosing parent object is chosen. Parent objects are either repeat frames, sub-frames of conditional objects, or the child frame of a parent/child construction.

Whenever an object is manipulated (created, moved or re-sized) its parent is highlighted. A square appears in the top left corner of a parent object if any of its children is manipulated. Thus the parent of any object can be found by clicking on the object. Furthermore, an object is never drawn outside the boundaries of its parent. While being moved, an object cannot be dragged outside the boundaries of its parent; this can also be used to find the parent of an object.

A.2 Interactors

Lines, rectangles and bitmaps have simple behaviours that have already been described. This section covers the other interactors in more detail.

A.2.1 Sliders

(Horizontal sliders are described here; vertical sliders are the same except for the orientation.) A slider has five regions: the left and right arrows, the thumb, and the regions between the thumb and the arrows. The thumb allows direct control over the value of the slider, by pressing the mouse button on it and dragging it to either side. The arrows allow fine control over the value; clicking in each arrow increases or decreases the value by a small amount. Clicking in the space between the arrows and the thumb signifies a large increment or decrement.

The attributes for the Pluto sliders are:

width the width (in screen pixels) of the slider.

low, high the lower and upper limits that the value of the slider may take. They default to 0 and 100.

small_inc, large_inc the small and large increments used in the interaction described above. They default to be 5% and 20% of the range between the low and high values.

A.2.2 Buttons

Both types of buttons (pushbuttons and toggles) can take as an attribute a code segment. A code value is a segment of Penguims code surrounded by curly braces, as in `{ V.state := 0; }` . The pushbutton takes just one; the toggle takes two, one for each state transition. The pushbutton only takes two attributes: the image and the action. The action can be either a code value or a cell reference which evaluates to a code value. The toggle takes two image attributes, one for each state. If only one image is provided, the other defaults to its inverse.

Images can be read in with a call to the function `bitmap`, which takes a string as an argument, which is interpreted as the name of a file that contains an image (bitmap).

A.2.3 Text objects

Text objects have two attributes besides the name and the value. These are width of the object and the font to use. If the width is not specified, the string is drawn in as much space as is required. If a width is given, the object will never draw outside the boundaries established by the size of the font and the width.

The string to display is taken from the field named *Value*. This can be any expression that evaluates to a string value. To convert integers to strings, the function `atoi` can be used. Since the data-field references are simply replaced with the value, the frequently used expression `atoi(#name)` can be written simply as `"#name"` to just display a data field named "name."

If the font field is left blank, the default font (for X windows, set in the X resources file) is used. Specific fonts can be used with the *font* function. It takes a string as an argument which is the name of a font in the underlying system. Under X-windows, an example of such a call would be `font("*-times-*-r-*-14-*")`.

A.3 Menus and Interactions

Fig. A.1 shows the Pluto main menu. There are three parts to the menu: the current mode display, the pull-down menus and the buttons.

The current mode displays a graphical representation of the current mode. The pull-down menus select object creation modes and the I/O functions. The buttons select major manipulation modes.

A.3.1 Pull-down Menus

These are the structures of the pull-down menus:

Frame Menu Contains the GLS objects that are invisible in the final display. These are: frame, parent-child, repeat frame, conditional; and the, reference lines: plain, minimum, maximum, average, and proportional.

Interactor Menu Contains the interactors: the buttons (toggle and pushbutton) and the sliders (horizontal and vertical).

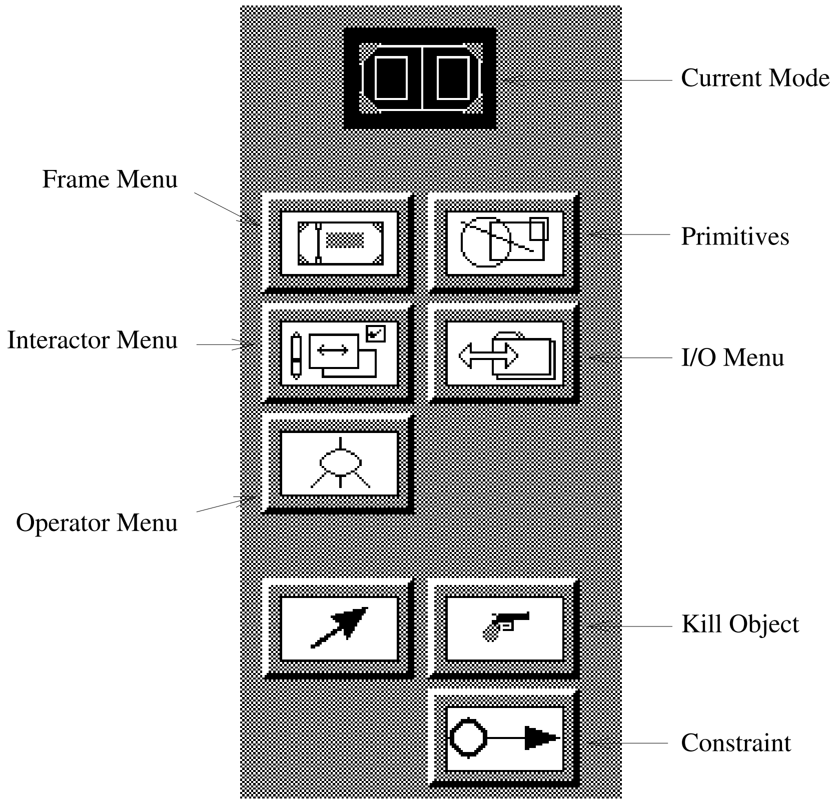

Figure A.1: The Pluto main menu

Primitives Contains the interactors that are also used to display values: lines, rectangles, bitmaps and text objects.

I/O Menu Contains the external operations: read, write, try it, include, clear and quit.

Operator Menu Contains the pre-defined operators: row, column, overlay and leaf.

A.3.2 Mode Buttons

The mode buttons control the major interaction mdoes.

Manipulate Allows the manipulation of objects. All variable-sized ojects can be re-sized by dragging the corners. All objects can be moved by dragging any edge.

Holding down the "Control" key and clicking on an object brings up the auxililary attribute editors. For most objects, they control attributes like the name of the object. For repeat frames, they bring up the query editor.

Kill Allows the removal of objects and constraints. The cursor changes to the skull-and-crossbones as a reminder. After one object or constraint is killed, the mode reverts to manipulation.

Constraint Allows constraints to be created. Start a drag near the edge of an object; a cross appears on that edge as feedback. (If there are several edges close by, this allows verification that the intended edge was chosen.) Drag the mouse to the destination; a rubber-banded line is drawn. When a compatible edge is approached, the line will snap to it; releasing the mouse button will create the constraint. The rubber-banded line will snap to the closest compatible edge; this can be used to select the right edge when there are many such edges close together.

Clicking on a constraint in either the constraint mode or the manipulation mode opens its attribute editor and sets the text focus to it. Clicking a constraint with an opened label closes it.

By default, constraints are drawn at the mid-point of the intersection of the extents of the edges begin joined. Holding down the control key allows them to be re-positioned in one dimension.

A.4 Constraints

The labels in constraints describe the relation it implements. The general form of a label is:

$$\begin{array}{c} expr_1 \; \textbf{[:= } \; expr_2 \; \textbf{]} \\ \textbf{!} \; expr \end{array}$$

In the first form, the label is an expression optionally followed by an initialisation clause. The attribute specified by the constraint is initialised to the value of the second expression; thereafter, it is the value defined by the first expression. These expressions are only re-evaluated if a value they depend on (directly or indirectly) has changed.

In the second form, the attribute is initialised to the value of the expression. Thereafter the value is controlled directly by the user. This form of constraint only applies when it is possible for the user to directly control the attribute, i.e. for lines, rectangles (both size and position) and for bitmaps (only the position).

A.5 Operators

Operators can only be created in the operator tree window. The type of operator is selected from the operator menu. The mouse is then pressed in the operator that is to be the parent of the new one. Dragging downwards creates a rubber-banded line that tracks the mouse; releasing the mouse button places the new operator at that position. A line is drawn to the parent. The system enforces the restriction that all the children of a node are drawn below it.

For some operators (*row* and *column*) the order of the children is significant. This order is defined by the positions of the children on the screen, reading from left to right. Children may be re-arranged by dragging them with the mouse to the required position.

Operators can be arbitrarily re-arranged by dragging with the mouse, as long as the parent-child relationship is preserved. No operator can be dragged above its parent or below any of its children.

REFERENCES

[1] C. Ahlberg, C. Williamson, and B. Shneiderman. Dynamic queries for information exploration. In *CHI 92 Conference on Human Factors in Computing Systems*, pages 619–626. ACM, 1992.

[2] A. V. Aho, B. W. Kernighan, and P. J. Weinberger. Awk—a pattern scanning and processing language. *Software—Practice and Experience*, 9(4), 1979.

[3] P. S. Barth. An object oriented approach to graphical interfaces. *ACM Transactions on Graphics*, 5(2):142–172, Apr. 1986.

[4] R. A. Becker and J. M. Chambers. *An interactive environment for data analysis and graphics*. Wadsworth, 1984.

[5] A. Borning. The programming language aspects of ThingLab, a constraint-oriented simulation laboratory. *ACM Transactions on Programming Languages and Systems*, 3:353–387, Oct. 1981.

[6] A. Borning. Defining constraints graphically. In *CHI 86 Conference on Human Factors in Computing Systems*, pages 137–143, Apr. 1986.

[7] A. Borning and R. Duisberg. Constraint-based tools for building user interfaces. *ACM Transactions on Graphics*, 5:345–374, Oct. 1986.

[8] A. Borning et al. Constraint hierarchies. In *OOPSLA 87 Conference on Object-Oriented Programming Systems*, pages 48–60. ACM, Oct. 1987.

[9] M. H. Brown and R. Sedgewick. A system for algorithm animation. In *SIGGRAPH 84 Conference on Computer Graphics*, pages 177–186. ACM, July 1984.

[10] C. G. Burgess. A database interface aid. In *Workshop on Visual Programming*, pages 72–76. IEEE, 1984.

[11] M. Buxton, W. Lamb, D. Sherman, and K. Smith. Towards a comprehensive user interface management system. In *SIGGRAPH 83 Conference on Computer Graphics*, pages 35–42. ACM, Jul. 1983.

[12] S. K. Card, T. P. Moran, and A. Newell. The keystroke-level model for user performance time with interactive systems. *Communications of the ACM*, 23(7):396–410, July 1980.

[13] L. Cardelli. Building user interfaces by direct manipulation. In *UIST 88 Symposium on User Interface Software and Technology*, pages 152–166. ACM SIGGRAPH, Oct. 1988.

[14] B. Czejdo, V. Reddy, and M. Rusinkiewicz. Design and implementation of an interactive graphical query interface for a relational database. In *Workshop on Visual Languages*, pages 14–20. IEEE, 1988.

[15] C. J. Date. *A Guide to the SQL Standard (Second Edition).* Addison-Wesley, Aug 1989.

[16] R. Duisberg. Animating graphical interfaces using temporal constraints. In *CHI 86 Conference on Human Factors in Computing Systems*, pages 131–136, Apr. 1986.

[17] B. N. Freeman-Benson, J. Maloney, and A. Borning. An incremental constraint solver. *Communications of the ACM*, 33(1):54–63, Jan. 1990.

[18] G. W. Furnas. Visualizing complex information spaces. In *CHI 86 Conference on Human Factors in Computing Systems*, pages 16–23. ACM, 1986.

[19] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, Reading, Mass., 1983.

[20] Paul Haeberli. ConMan: A visual programming language for interactive graphics. *Computer Graphics*, 22(4):103–111, Aug. 1988.

[21] D. A. Henderson. The Trillium user interface design environment. In *CHI 86 Conference on Human Factors in Computing Systems*, pages 221–227, Apr. 1986.

[22] T. R. Henry. *Interactive graph layout: the exploration of large graphs*. PhD thesis, University of Arizona, Tucson, AZ 85721, 1992.

[23] T. R. Henry, S. E. Hudson, and G. L. Newell. Integrating gesture and snapping into a user interface toolkit. In *UIST 88 Symposium on User Interface Software and Technology*. ACM SIGGRAPH, Oct. 1990.

[24] S. E. Hudson. UIMS support for direct manipulation interfaces. *ACM Computer Graphics*, 21(2):120–124, Apr. 1987.

[25] S. E. Hudson. Semantic feedback in the Higgens UIMS. *IEEE Transactions on Software Engineering*, 14(8):1188–1206, Aug. 1988.

[26] S. E. Hudson. Adaptive semantic snapping—a technique for semantic feedback at the lexical level. In *CHI 90 Conference on Human Factors in Computing Systems*, Apr. 1990.

[27] S. E. Hudson. Incremental attribute evaluation: A flexible algorithm for lazy update. *ACM Transactions on Programming Languages and Systems*, 13(3):315–341, July 1991.

[28] S. E. Hudson. User interface specification using an enhanced spreadsheet model. Technical Report GVU-93-20, Georgia Institute of Technology, 1993.

[29] S. E. Hudson and S. P. Mohamed. Interactive specification of flexible user interface displays. *ACM Transactions on Information Systems*, 8(3):269–288, Jul. 1990.

[30] Scott E. Hudson. Incremental attribute evaluation: A flexible algorithm for lazy update. *ACM Transactions on Programming Languages and Systems*, 13(3):315–341, Jul. 1991.

[31] E. L. Hutchins, J. D. Hollan, and D. A. Norman. Direct manipulation interfaces. In D. A. Norman and S. W. Draper, editors, *User Centered Systems Design*, pages 87–124. Lawrence Erlboum Associates, Hillsdale, New Jersey, 1986.

[32] R. J. K. Jacob. A specification language for direct manipulation user interfaces. *ACM Transactions on Graphics*, 5(4):283–317, Oct. 1986.

[33] H. Kim, H. Korth, and A. Silberschatz. PICASSO: A graphical query language. *Software: Practice and Experience*, 18(2):169–203, Feb. 1988.

[34] D. E. Knuth. *The TEXbook*. Addison-Wesley, Reading, MA, 1984.

[35] W. Leler. *Constraint Programming Languages: Their Specification and Generation*. Addison-Wesley, 1988.

[36] C. H. Lewis. NoPumpG: Creating interactive graphics with spreadsheet machinery. Technical Report CS-CU-372-87, University of Colorado, Aug. 1987.

[37] M. A. Linton, J. M. Vlissides, and P. R. Calder. Composing user interfaces with InterViews. *IEEE Computer*, 22(2):8–22, Feb. 1989.

[38] J. Mackinlay. Automating the design of graphical presentations of relational information. *ACM Transactions on Graphics*, 5(2):110–141, april 1986.

[39] J. D. Mackinlay, G. G. Robertson, and S. K. Card. The Perspective Wall: Detail and context smoothly integrated. In *CHI 91 Conference on Human Factors in Computing Systems*, pages 173–179. ACM, April 1991.

[40] A. MacLean et al. User-tailorable systems: Pressing the issues with buttons. In *CHI 90 Conference on Human Factors in Computing Systems*, pages 175–182. ACM, Apr. 1990.

[41] A. D. Malony and D. A. Reed. Visualizing parallel computer system performance. Technical Report UIUCDCS-R-88-1465, University of Illinois, Sep. 1988.

[42] E.-J. Marey. *La Méthode Graphique*, page 73. Gauthier/Villars, Paris, 1885.

[43] J. McCormack and P. Asente. An overview of the X toolkit. In *UIST 88 Symposium on User Interface Software and Technology*, pages 46–55. ACM SIGGRAPH, Oct. 1988.

[44] B. Myers et al. The Garnet toolkit reference manuals: Support for highly-interactive, graphical user interfaces in Lisp. Technical Report CMU-CS-89-196, Carnegie-Mellon University, Nov. 1989.

[45] B. A. Myers. A system for displaying data structures. *Computer Graphics*, 17(3):115–125, Jul. 1983.

[46] B. A. Myers and W. Buxton. Creating highly interactive graphical user interfaces by demonstration. In *SIGGRAPH 88 Conference on Computer Graphics*, pages 249–258. ACM SIGGRAPH, Aug. 1986.

[47] B. A. Myers, B. Vander Zanden, and R. B. Dannenberg. Creating graphical interactive application objects by demonstration. In *UIST 89 Symposium on User Interface Software and Technology*, pages 95–104. ACM SIGGRAGH, 1989.

[48] T. Nadas and A. Fournier. Grape: An environment to build display processes. In *SIGGRAPH 87 Conference on Computer Graphics*, pages 75–84. ACM, Jul. 1987.

[49] C. D. Norton and E. Glinert. A visual environment for designing and simulating execution of processor arrays. In *Workshop on Visual Programming*, pages 227–232. IEEE, 1990.

[50] D. R. Olsen and E. P. Dempsey. Syngraph: A graphical user interface generator. *Computer Graphics*, 17(3):43–50, Jul. 1983.

[51] G. Raeder. *Programming in Pictures*. PhD thesis, University of Southern California, Los Angeles, CA, 1984.

[52] B. J. Reiser et al. A graphical programming language interface for an intelligent lisp tutor. In *CHI 88 Conference on Human Factors in Computing Systems*, pages 39–44. ACM, May 1988.

[53] S. P. Reiss. Graphical program development with Pecan program development systems. *Sigplan Notices*, 19(5):30–41, May 1984.

[54] G. G. Robertson, J. D. Mackinlay, and S. K. Card. Cone Trees: Animated 3D visualizations of hierarchical information. In *CHI 91 Conference on Human Factors in Computing Systems*, pages 189–194. ACM, Apr. 1991.

[55] R. W. Scheifler and Gettys J. The X window system. *ACM Transactions on Graphics*, 5(4):79–109, Apr. 1986.

[56] B. Shneiderman. The future of interactive systems and the emergence of direct manipulation. *Behaviour and Information Technology*, 1:237–256, 1982.

[57] B. Shneiderman. Direct manipulation: A step beyond programming languages. *IEEE Computer*, 16(8):57–69, Aug. 1983.

[58] G. Singh and M. Green. Designing the interface designer's interface. In *UIST 88 Symposium on User Interface Software and Technology*, pages 109–116. ACM SIGGRAPH, Oct. 1988.

[59] G. Singh and M. Green. Chisel: A system for creating highly interactive screen layouts. In *UIST 89 Symposium on User Interface Software and Technology*, pages 86–94. ACM SIGGRAPH, Nov. 1989.

[60] D. C. Smith, C. Irby, et al. Designing the Star user interface. *Byte*, 7(4):242–282, Apr. 1982.

[61] D. N. Smith. Building interfaces interactively. In *UIST 88 Symposium on User Interface Software and Technology*, pages 144–151. ACM SIGGRAPH, Oct. 1988.

[62] D. N. Smith. Visual programming in the Interface Construction Set. In *Workshop on Visual Languages*, pages 109–120. IEEE, 1988.

[63] J. T. Stasko. *TANGO: A Framework and System for Algorithm Animation*. PhD thesis, Brown University, 1989.

[64] J. T. Stasko. Using direct manipulation to build algorithm animations by demonstration. In *CHI 91 Conference on Human Factors in Computing Systems*, pages 307–314. ACM, Apr. 1991.

[65] P. Sukaviriya. Dynamic construction of animated help from application context. In *UIST 88 Symposium on User Interface Software and Technology*, pages 190–202. ACM SIGGRAPH, Oct. 1988.

[66] P. A. Szekely and B. A. Myers. A user-interface toolkit based on graphical objects and constraints. In *OOPSLA 88 Conference on Object-Oriented Programming Systems*, pages 36–45, Sept. 1988.

[67] E. R. Tufte. *The Visual Display of Quantitative Information*. Graphics Press, Box 430, Cheshire, CT 06410, 1983.

[68] E. R. Tufte. *Envisioning Information*. Graphics Press, Box 430, Cheshire CT 06410, 1990.

[69] J. W. Tukey, R. McGill, and W. A. Larsen. *Variations of box plots*, volume V: Graphics of *The Collected Works of John W. Tukey*, chapter 5, pages 63–77. Wadsworth and Brooks/Cole, 1988.

[70] J. W. Tukey and P. A. Tukey. *Graphic display of data sets in 3 or more dimensions*, volume V: Graphics of *The Collected Works of John W. Tukey*, chapter 5, pages 189–289. Wadsworth and Brooks/Cole, 1988.

[71] J. W. Tukey and P. A. Tukey. *Some Graphics for studying four-dimensional data*, volume V: Graphics of *The Collected Works of John W. Tukey*, chapter 5, pages 171–188. Wadsworth and Brooks/Cole, 1988.

[72] N. Wilde and C. H. Lewis. Spreadsheet-based interactive graphics: From prototype to tool. In *CHI 90 Conference on Human Factors in Computing Systems*, pages 153–159, Apr. 1990.

[73] T. Williams, C. Kelley, et al. *Gnuplot V3.2*. Free Software Foundation, 1000 Mass. Ave., Cambridge MA 02138, USA, 1992.

[74] M. M. Zloof. QBE/OBE: A language for office and business automation. *IEEE Computer*, 5(14):13–22, May 1981.

菩提本非樹

明鏡亦非台

何處惹塵埃?