A USER INTERFACE TO EDS
USING THE X WINDOW SYSTEM AND XT

by

Karam Michael Noujeim, BSEE

A thesis submitted to the Faculty of Graduate Studies and
Research in partial fulfillment of the requirements
for the degree of Master of Engineering

Department of Electrical Engineering
McGill University, Montréal
Québec, Canada

November, 1990

Abstract

A user interface to EDS (Electromagnetic Design System) is designed using the X Window System and the X Toolkit. The grammar of expressions provided by the user is validated using a bottom-up parser. Equations are converted to postfix/prefix notation, and later displayed in binary tree representation inside pop-up widgets.

The interface allows the user to determine equation interdependencies, and to generate flow graphs for different sets of equations within the same application. Equations may be added or deleted from any set of entered equations at any time. Flow graph nodes and links may be queried for equation information. Retrieved information is displayed inside pop-up widgets.

Finally, the ground work for a user interface package that simplifies X Toolkit programming is outlined. Rules and methods governing user interface components are introduced.

Résumé

Une interface usager graphique est conçue pour le système de conception électromagnétique EDS à l'aide du système de fenêtres X, et du X-Toolkit. La grammaire des expressions fournies par l'utilisateur est vérifiée par voie d'analyse grammaticale ascendante. Les équations sont converties en notation suffixée/préfixée, et leur représentation en arbre binaire est affichée dans des fenêtres pop-up.

L'interface permet à l'utilisateur de déter: iner les relations entre équations, et de représenter des ensembles d'équations en forme de réseaux. L'utilisateur peut ajouter ou supprimer des expressions de n'importe quel ensemble d'équations. Les nodes et les branches des réseaux d'équations peuvent être interrogées. Les résultats sont affichés dans des fenêtres pop-up.

Enfin, des méthodes nouvelles pour simplifier la programmation en X-Toolkit sont discutées. Des lois et des méthodes gouvernant les différents objets constituant les interfaces usagers graphiques sont introduites.

Acknowledgements

I would like to thank my supervisor Professor D. A. Lowther for his help throughout the course of my master's research. His patience and concern helped me overcome numerous problems.

Special thanks to Peter Ashwood-Smith of Bell-Northern Research for his helpful comments and suggestions, and to Carlos Saldanha whose master's thesis provided a subject for my own. I also would like to express my thanks to Bell-Northern Research for letting me use their computer facilities after work hours, and on weekends. Many thanks to my fellow students in the CADLAB for creating a great environment for friendship and research.

I also would like to thank my parents Michael and Sonia for their support, and encouragement, and my sister and two brothers for their patience and caring.

Finally, I would like to express my thanks to Centre de Recherche Informatique de Montreal for their financial support.

Table of Contents

List of Figures

CHAPTER 1

Introduction

This dissertation describes the user interface designed for the Electromagnetic Design System (EDS). Before any dissertation objectives can be stated, EDS must first be introduced.

1.1 Introduction to EDS

EDS (Electomagnetic Design System) is a knowledge-based expert system aimed at automating the computer-aided design of electromagnetic devices such as transformers, actuators, and motors [9]. EDS allows expert designers to create knowledge-based models of electromagnetic devices. The solution to a device design problem involves three categories of knowledge [9][25]:

* A parameterized representation of the device. Device parameters are variables that represent geometric, functional, performance, and design calibration information.

* A mathematical model in which device parameters are coupled through a set of equations derived primarily from the underlying principles of electromagnetics.

* Design Logic. This consists of the mathematical model, and the expertise that a designer acquires through the years.

The algebraic model of a device is represented by parameters, equations, and constants [9]. An algebraic model may be graphically represented as a constraint network [26], where nodes represent equations, and links are the variables in the equations. An example of a constraint network is shown in figure 1.1 [9].

EDS uses data structures known as *frames* and *semantic networks* to represent complete constraint networks [9][27]. A frame is an item, and a list of item attributes. On the other hand, semantic networks are graphs of relations between items. The frame-based language used in EDS is **Knowledge Craft** [28][9], a super set of the programming language **Lisp** [29]. In **Knowledge Craft**, a frame is known as a *schema*.

EDS uses a simplifier to simplify expressions, and a symbolic solver to solve for a variable in a given equation [9]. The *rule* slot of an equation schema is populated by an input expression, only after this expression has been simplified. After the *rule* slot is filled, the input equation is solved for each of the variables, and the resulting clauses are inserted into the *clauses* slot. In cases where a solution is not found due to the lack of algebraic knowledge, the *variables-with-no-clause* slot is filled with the corresponding variable. In such cases, the designer may supply a clause to resolve the problem, or provide an alternate equation which is algebraically solvable for the variable [9].

a) A typical set of transformer equations.

$$E1:\ WA = 17.6 * P1 * S / (FREQ * B)$$
$$E2:\ P2 = EFF * P1$$
$$E3:\ P2 = V2 * I2$$

b) The corresponding constraint network.

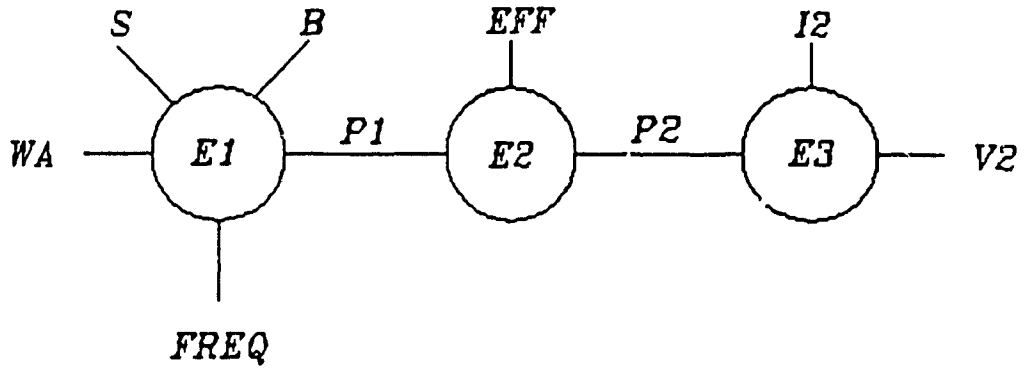


Figure 1.1 A conceptual algebraic model representation [9].

```
{{ <equation name>
        DESCRIPTION:
        RULE
        HAS-VARIABLES:}}
```

Figure 1.2 A generic equation schema [9].

```
{{ POWER-DISSIPATION
        RULE:           (EQUAL P (TIMES (EXPT I 2) R))
        HAS-VARIABLES: P I R
        CLAUSES:        (EQUAL P (TIMES (EXPT I 2) R))
                        (EQUAL R (TIMES P (EXPT I -2)))
                        (EQUAL I (EXPT (QUOTIENT P R) -0.5))
        VARIABLES-WITH-NO-CLAUSE:                    }}
```

Figure 1.3 The schema representation of $P = I^2 * R$ [9].

Interactive sessions in EDS are command line driven, and consist of text input and output only. EDS relies on a DECnet connection to access a **Knowledge Craft** shell running on a remote computer (the remote computer is actually located at the *Centre de Recherche Informatique de Montréal*, and is accessed from the *Computational Analysis and Design Laboratory* at McGill University).

1.2 Dissertation Objectives

At this stage, the objectives of this dissertation may be stated as follows:

1) Design a graphical user interface to the Electromagnetic Design System, capable of running across networks.

2) Display algebraic knowledge in the form of graphical equation networks, thereby providing electromagnetic device designers with a visual aid that helps them point out related equations, and variables common to those equations.

3) Provide a mechanism for performing set operations on equations. This includes creating, and graphically representing sets/subsets of equations, and adding/deleting equations from those sets/subsets.

The X Window System is used to accomplish the first objective. X is a multi-windowing, and network-transparent window system based on a client-server model. It was chosen for designing the EDS UI mainly because it allows graphical applications (clients) to run across networks. The next chapter introduces X concepts, and many of X's features.

User interface components (scrollbars, pull-down menus, pop-up windows, ...) used to design the EDS UI are introduced in chapter three, preceded by a brief review of object-oriented programming concepts.

The EDS UI design is introduced in chapter four. The functionality of the components selected for building the interface are discussed, followed by the EDS UI widget tree. Chapter four also discusses the resource manager which may be used by end users for modifying the look of the EDS UI to fit their own taste.

Expression grammar, parsing methods, and the approach adopted in this dissertation to parse input expressions are presented in chapter five. Equations are converted into post-fix/prefix notation. The derived prefix notation of an input equation is used to fill out the *rule* slot of an EDS equation schema. Binary tree representations of equations are displayed in pop-up windows. Drawing is done using graphics primitives from the Xlib library.

Chapter six discusses equation sets (represented as linked lists of structures in memory), and the method used for generating graphical equation networks of different sets of equations.

Variables common to related equations are determined. Variables of an input equation are extracted and are used to fill out the *variables* slot of an EDS equation schema. Storage and retrieval of equation information, as well as memory management are also explained.

Finally, the results of this exercise are discussed, and suggestions for further work are presented in chapter seven. Throughout the remainder of this dissertation, graphical equation networks will be referred to as flow graphs.

CHAPTER 2

Introduction to the X Window System

2.1 The X Window System

The X Window System is an industry-standard software system that allows programmers to develop portable graphical user interfaces [1]. X was developed at the Massachussets Institute of Technology (MIT), with support from Digital Equipment Corporation (DEC). The name, X, as well as some of the initial ideas originated from an earlier window system named W, developed at Stanford University [1].

One of the most important features of X is its unique device-independent architecture. This allows X-based applications to function in a heterogeneous environment consisting of mainframes, workstations, and personal computers [1]. Unlike many other window systems, X does not define any particular user interface style. It avoids dictating the look and feel of user interface applications by providing a flexible set of primitive window operations. User interface components such as button boxes and pull-down menus are therefore missing from the basic X Window System, and applications rely on higher level libraries built on top of the X protocol to provide these components [1]. The following chapter introduces one of these libraries, the Xt Intrinsics, built on top of Xlib.

2.2 The Client-Server Model

The X Window System architecture is based on a *client-server* model. A single process, known as the *server*, controls all input and output devices [1]. The server acts as an intermediary

between user programs, better known as *clients*. Clients communicate with the X server via a network connection using an asynchronous byte-stream protocol. Network protocols supported by the X server include TCP/IP, DECnet, and Chaos. The following are the tasks performed by the X server [2]:

- Controlling access to the display by multiple clients.
- Interpreting network messages from various clients.
- Forwarding user input to clients.
- Drawing text and graphics.
- Maintaining data structures, including windows, fonts, and graphics contexts, as resources that can be shared by clients.

The X architecture makes it possible for any number of clients to connect to any number of servers, provided that the X protocol is obeyed. Servers and clients can also run on separate machines located anywhere on a network. This use of the network is better known as *distributed processing*, and helps solve the problem of unbalanced system loads [2]. The user of an overloaded machine can arrange for some of the programs to run on other hosts.

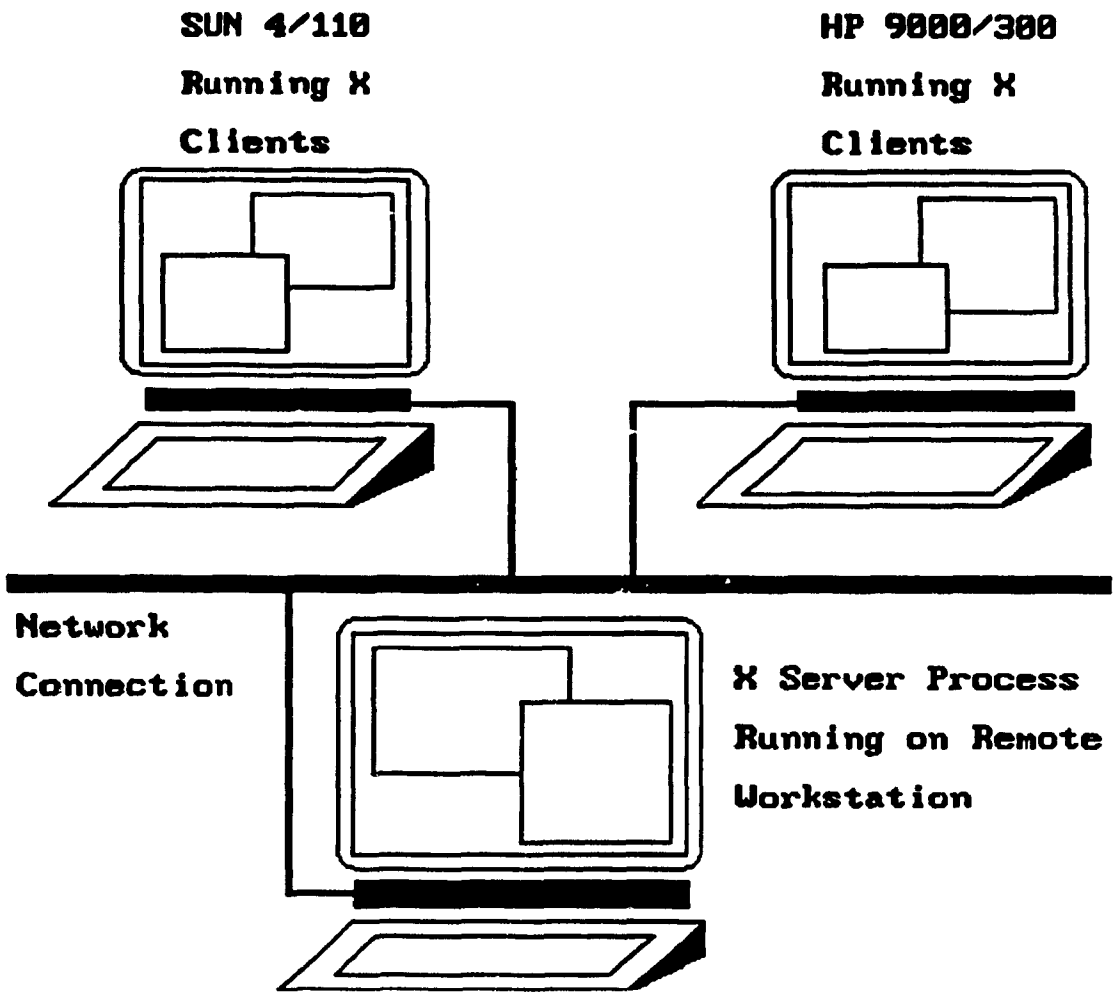


Figure 2.1 The client-server model.

X is a windowing system for bitmapped graphics displays [2]. It supports color as well as gray-scale and monochrome displays. While the terms *display* and *screen* are often used interchangeably to refer to a CRT, X defines a display as a workstation with a keyboard, a pointing device such as a mouse, and one or more screens. Normally, there is one display per central processing unit (CPU).

Any client application wishing to communicate with the X server must open a connection to this server using Xlib. Once this connection is established, the application can use any of the screens controlled by the server to display text or graphics. At any time, a server can deny client applications running on other hosts the right to connect to a display. This security mechanism provided by the X Window System works on a per-host basis [1]. Using the setup shown in figure 2.1, Knowledge Craft, and the X server may run on the remote workstation (located at *Centre de Recherche Informatique de Montréal*), while EDS may run as a client application on any of the host machines (located in the *CADLab, at McGill*) connected to the remote workstation.

2.3 Resources And Requests

The X server maintains complex data structures, including windows, bitmaps, fonts, cursors, and colors, as resources that can be shared between clients. Client programs access these resources through resource identifiers, simply referred to as resource ID's. Resources may be created or destroyed by the server at the request of a client [1].

A client application wanting to use a facility provided by the X server, must issue a request to this server. A request is a single block of data sent by one of the clients to the server. Requests requiring replies from the server are known as *round-trip* requests [2]. Round-trip requests must be minimized since they lower the overall performance, and cause network delays. Typical client requests include querying the server about window attributes, or font sizes.

Client requests are placed in a queue, waiting for the X server to process them. The server and its clients run asynchronously with respect to each other, which improves the overall

performance of X, and cuts down the round-trip requests over the network connection. However, clients can ask the server to process requests synchronously. This causes poorer performance, since each client request suffers a round-trip over the network connection [1].

2.4 The Window Hierarchy

An X window is a rectangular area on the screen, with no title bars, scrollbars, or other window decorations. Except for the root window, every X window has a parent that is assigned to it at the time of its creation. The root window is the first window created by the X server as it starts up. In X, a window is contained within the limits of its parent. Window geometry includes a window's width, height, position, and stacking order. On the other hand, an X window may be of class **InputOutput** or **InputOnly**, and has characteristics referred to as *depth* and *visual* , which determine its color attributes.

X windows are organized in a hierarchy, better known as the *window tree*. The top window in the window tree is the root window. The root window occupies an entire screen, and cannot be resized, moved, or iconized. Figures 2.2 and 2.3 illustrate how a window hierarchy might show on the screen and in schematic form respectively.

Windows A and G are children of the root window, while windows B, C, and D are children of window A. Window G has windows H, J, and K as its children, and windows L and M are children of J and K respectively. Similarly, windows E and F are children of C and D.

2.5 Window Management

Client applications do not have direct control over window attributes such as window size and location. Although they can suggest values for these attributes, the final decision is made by a client program known as the *window manager*. The window manager allows the user to perform such actions as modifying the location and size of windows on the screen, reconfiguring the stacking order of windows on the screen, and starting new client applications [3].

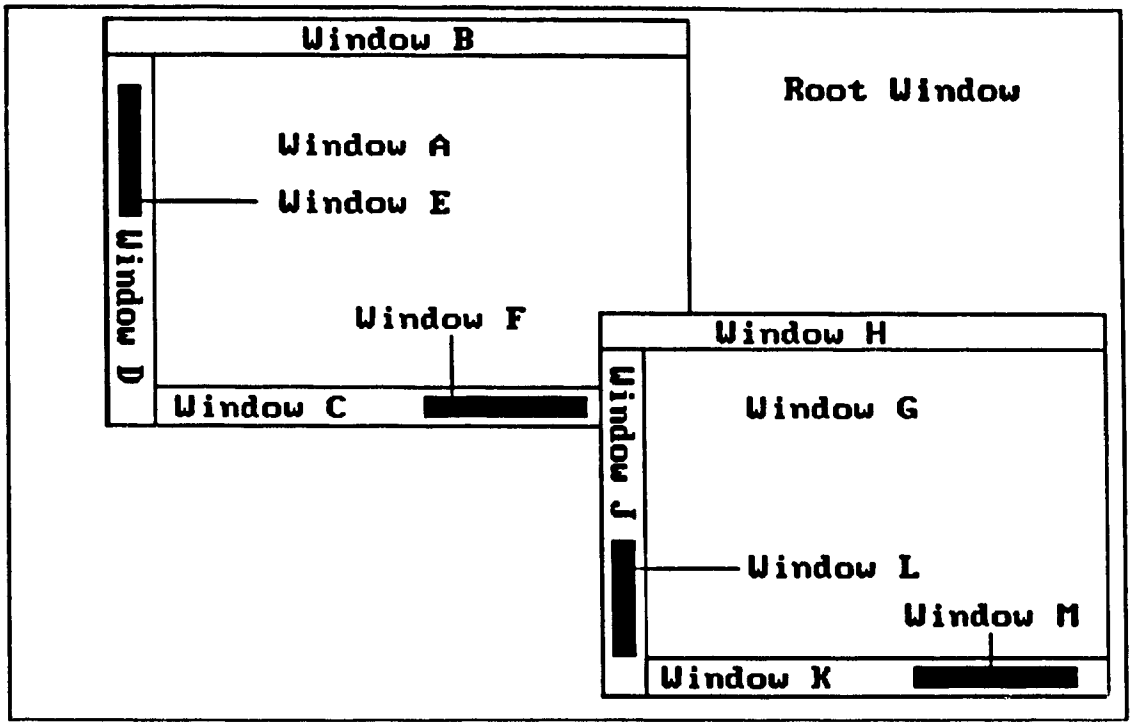


Figure 2.2 A window hierarchy example.

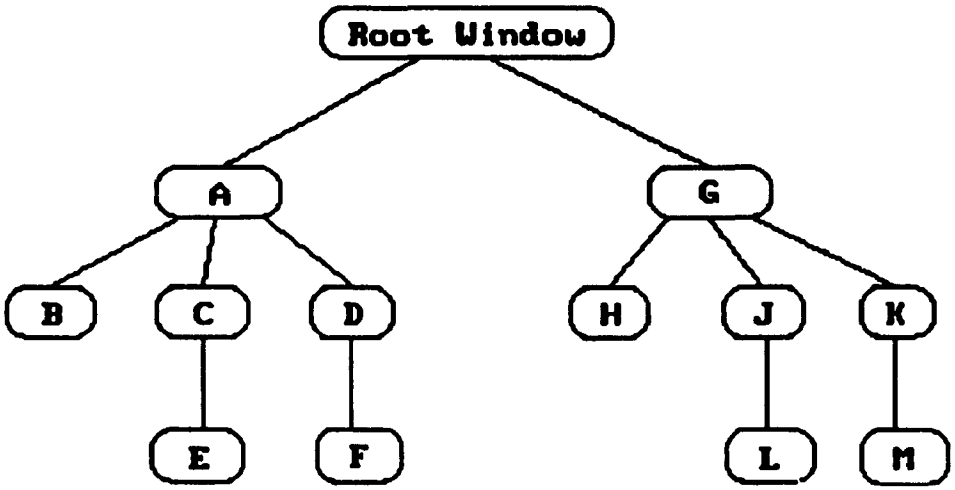


Figure 2.3 Window tree for window hierarchy example.

The window manager enforces a *window layout policy* which consists of a set of rules governing the behavior and look of windows on the screen. The following are some of the different public domain window managers that have been designed, mostly by companies:

- The Siemens RTL *tiled* window manager, designed such that only transient windows are allowed to overlap.
- The *real-estate-driven* window manager, designed such that the input focus is active in whichever window the pointer currently happens to be in.
- The *listener* window manager. All keyboard input focus is assigned to a single window after the window is selected with the pointer.

2.6 The X Coordinate System

The origin of every X window is located at the upper left corner of that window. The x-coordinate increases toward the right, and the y-coordinate increases toward the bottom.

The width, height, and location are expressed in pixels. Placing text, graphics, or subwindows in an X window is made with respect to that window's origin. As a window moves, its coordinate system moves with it, permitting applications to place text or graphics without regard to the window's location. Each X window is assigned a unique ID, and routines wanting to access a particular X window must refer to its ID.

2.7 Window Mapping and Visibility

A request to an X server by one of its client applications to create a window, does not necessarily make the window visible on the screen. This is due to the fact that when an X server creates a window, it allocates and initializes the data structures that represent the window, but does not invoke the hardware-dependent routines that display the window on the screen [1]. By issuing a *map* request, clients can ask the server to display a window on the screen. However, the window still might not be visible for any of the following reasons [1]:

- The window is completely covered by another window. The window becomes visible only if the covering window is moved, or if the stacking order of the two windows is reversed, making the covered window visible.
- An ancestor of the window was not mapped. The window becomes viewable only when all of its ancestors are mapped.
- The window is completely clipped by one of its ancestors. The window becomes visible once the ancestor is resized and includes the window area. Another way is to move the window inside the boundaries of all its ancestors.

2.8 Maintaining Window Contents

When two windows overlap, the contents of the covered window must be preserved, so that they can be restored later. Windows using this technique to preserve their contents are better known as *retained raster* windows. As the name suggests, window contents are saved as a *bitmap*, or *raster* image.

2.8.1 Backing Store

X lays the responsibility of preserving a window's contents on the client using the window. Some X server implementations support retained raster, or *backing store* as it is referred to in X. The backing store feature automatically preserves the contents of a window as it is obscured. As the number of windows increases, memory becomes a scarce resource. To increase efficiency, the X server notifies a client when a window is *exposed*, and relies on the client to redisplay the contents of the window [1].

2.8.2 Save-Unders

Many X servers also use a technique known as *save-unders*. A save-under controls whether the contents of the screen under a certain window should be preserved before the window is mapped, and redisplayed after the window is unmapped. This technique is most useful for pop-up windows, which this dissertation relies on heavily to display different sets of equations.

2.9 Events

An X *event* is a packet of information that the X server generates when certain direct or indirect user actions occur. Events are sent to a client through a queue, which the client reads in a first-in, first-out fashion. The following are some examples of X events:

- A key press on the keyboard.
- Mouse movement.
- A mouse click.

Since events board the queue in a random order, an *event loop* is used to wait for an event to occur, respond to the event, and wait for the next one to happen. The code that forms the event loop consists of an event-getting routine, followed by a C-language *switch* statement. The X event loop is implemented as an infinite *while* loop [2].

2.10 Interfacing to X through Higher Libraries

Although the X server is built at the level of packets and byte-streams, libraries exist that interface to the base window system. One standard interface to X is the C-language, Xlib library. Xlib defines a set of functions that provide the user with complete access and control over the display, windows, and input devices. Identical libraries also exist for LISP and ADA [1].

Application programmers can use Xlib to design user interfaces. However, this library can be difficult to use correctly. Just imagine that to display "hello world" in a window using Xlib takes forty executable statements ! [8]. In an effort to hide the details of programming

with Xlib, higher-level *toolkits* have been designed. These toolkits include InterViews (Stanford University), Andrew (Carnegie-Mellon), Xray (Hewlett-Packard), and CLUE (Texas Instruments). In designing the interface to EDS, I have used the Xt Intrinsics, and the X Widget Set, Xw, contributed to the X community by Hewlett-Packard.

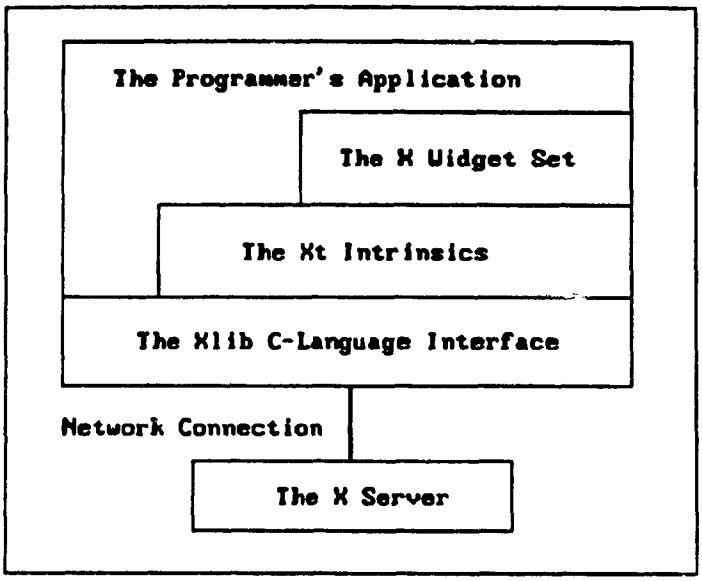


Figure 2.4 A conceptual view of the X Window System.

Using the X Window System, EDS can run as a client application across a network of X host machines. X allows graphics applications to run across networks. An end user may run multiple client applications on different host machines, while graphically interacting with those clients from his/her own terminal.

Before describing the EDS UI, a brief introduction to the Xt and Xw widgets is presented in the following chapter. Both Xt and Xw are built on top of Xlib, and are smoothly integrated with it. This allows programmers to use the functions provided by Xlib, in addition to using the higher level libraries. The Xt Intrinsics and the X Widget Set are both written in C.

CHAPTER 3

The X Toolkit

The Xt Intrinsics provides programmers with an extensible set of user interface components that include *scrollbars, menus, pushbuttons, and dialog boxes*. These components are better known as *widgets*. A widget is a complex data structure that consists of an X window and a set of procedures that act on that window [1]. At this stage, it is important to point out the difference between a widget programmer and an application programmer. A widget programmer is a producer of self-contained reusable components, while an application programmer is a consumer that uses these components to design applications.

3.1 Widget Classes

The Xt Intrinsics use an object-oriented approach that organizes widgets into *classes*. A class is a set of *objects* that possess similar characteristics. On the other hand, an object is an abstraction that combines data and the actions (also referred to as methods) that can be performed on the data. Individual objects are *instances* of a given class. For example, an object-oriented graphics program for drawing circles, polygons, cubes, etc.., might define classes **CIRCLE, POLYGON, CUBE**, etc.., for each of these different objects. This setup may be symbolically represented as follows:

$$CIRCLE = \{x \ni x \text{ is a circle}\}$$

$$POLYGON = \{y \ni y \text{ is a polygon}\}$$

$$CUBE = \{z \ni z \text{ is a cube}\}$$

The class **GRAPHICSOBJECT** is the set of classes formed by combining the above three classes, and is also known as a super class. Each of the given classes might also define such attributes as color, position, and dimension of a graphical object.

$$GRAPHICSOBJECT = \{CIRCLE, POLYGON, CUBE\}$$

In the Xt Intrinsics, instances of a given widget class may be created, and widget attributes (also referred to as widget resources) may be set. The Xt Intrinsics also supports *inheritance*, another useful object-oriented concept that allows a given class to inherit some or all of the characteristics of another class, or super class. The different widget classes provided by the Xt Intrinsics and the X Widget Set are listed in appendix A.

3.2 The Xt Intrinsics Programming Format

The majority of applications using the Xt Intrinsics follow the programming outline shown in figure 3.1. The first step consists of establishing a connection with the X server, and initializing the Xt Intrinsics. Next, widgets may be created and widget resources may be set. In the third step, *event handlers* and *callbacks* may be defined. Finally, all created widgets are realized, and the application enters the event loop.

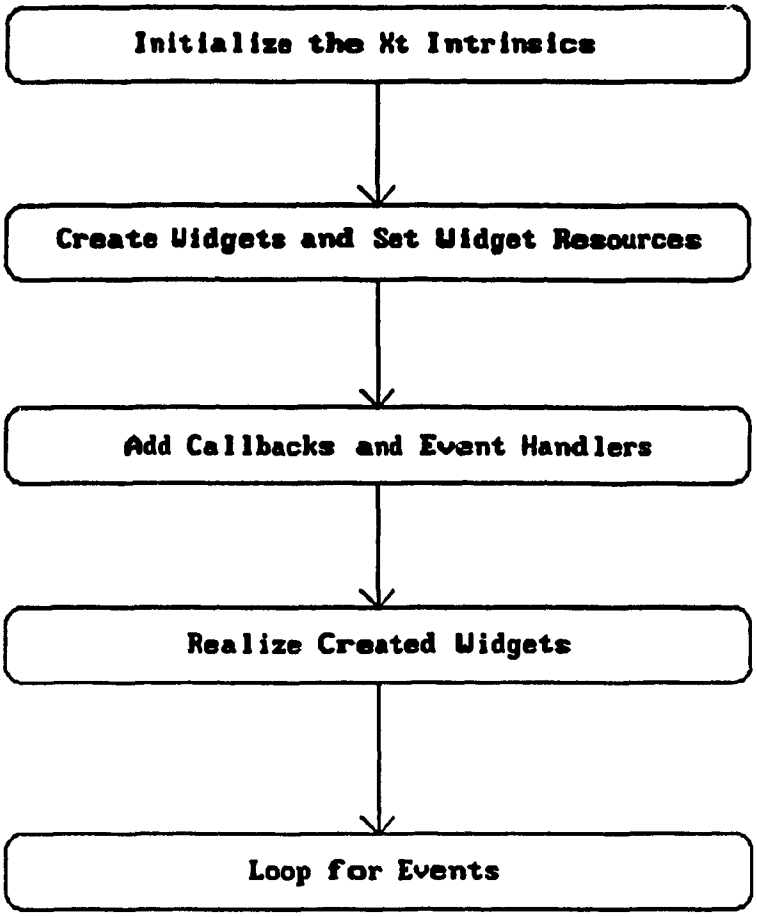


Figure 3.1 The Xt Intrinsics programming chart.

3.2.1 Creating and Managing Widgets

Widgets are organized in a hierarchical structure referred to as a *widget tree*. The root of the widget tree is known as the root window. After a widget is created, it must be *managed* by a parent widget. Basically, a widget's parent *manages* such attributes as the location of the widget, its size, and whether the widget has the input focus. A widget's appearance on the screen is controlled by a set of options and resources that may be specified by the end user.

3.2.2 Handling Events

The Xt Intrinsics provides application programmers with a *listener* mechanism that waits for events to occur in a widget, and automatically invokes *event handling* functions registered by the widget or the application itself for handling each of the events [1]. Finding the proper widget where an event takes place, and invoking the right event handler(s) is referred to as *event dispatching*.

3.2.2.1 Event Handlers and Callback Functions

An *event handler* is a function that gets invoked automatically by the Xt Intrinsics when an X event takes place within a widget. Applications can register multiple event handlers for the same event occurring in the same widget. However, the Xt Intrinsics does not define the order in which multiple event handlers are invoked when the triggering event occurs.

Callbacks are functions that get called when certain widget-specific conditions are met. Callbacks are different from event handlers in that they get invoked by widgets rather than the Intrinsics, and are not necessarily tied to any X event.

3.2.3 Xt Programming: A Brief Example

How hard can it be to display "*Hello World*" in a window using Xt ?. Using Xlib, this takes forty executable statements [8]. The following example program answers the same question from the X Toolkit point of view.

```
/*****************************************************
* hello.c: Display "Hello World" in a StaticText window *
*****************************************************/
#include <X11/Intrinsic.h>
#include <X11/StringDefs.h>
#include <Xw/Xw.h>
#include <Xw/SText.h>

main(argc, argv)
  int  argc;
  char *argv[];
{
  Widget toplevel, helloworld;
  Arg   wargs[1];
  int   i;

  /*
  | Initialize the Xt Intrinsics.
  */
  toplevel = XtInitialize(argv[0], "Hello", NULL, 0,
                &argc, argv);
  /*
  | Create a StaticText widget and display "Hello World"
  | inside it.
  */
  i = 0;
  XtSetArg(wargs[i], XtNstring, "Hello World"); i++;
  helloworld = XtCreateManagedWidget("hw", XwstatictextWidget
                                     Class,toplevel,wargs,i);
  /*
  | Realize the widgets and loop for events.
  */
  XtRealizeWidget(toplevel);
  XtMainLoop();
}
```

The function

XtInitialize(name, class, options, noptions, &argc, argv)

establishes a connection with the X server, and initializes a resource database used by the X resource manager. The *name* parameter is usually the name of the application, while the *class* parameter indicates the general class the application belongs to. The third and fourth parameters specify how the Intrinsics should interpret application-specific command-line arguments, while the last two parameters control common command-line arguments. **XtInitialize()** creates and returns a **TopLevelShell** widget which serves as the root of all other widgets in the calling application [1].

The function

XtCreateManagedWidget(name, class, parent, args, nargs)

creates a managed widget in one call to the Xt Intrinsics. The *name* argument is an arbitrary name of the widget, and *class* is the type of widget to be created. For an overview of the widget classes available in Xw, refer to appendix A. The *parent* argument specifies the widget parent of the widget to be created, while the last two arguments to **XtCreateManagedWidget()** specify the resources used by the widget [1]. Since **XtCreateManagedWidget()** simply allocates and initializes the widget data structures, a call to the function

XtRealizeWidget(widget)

must be made if the widget is to be physically displayed on the screen. The Xt Intrinsics for specifying widget resources is done through the use of the function

XtSetArg(arg, name, value)

where *arg*, *name*, and *value* represent an array element of type **Arg**, the resource name, and the value of the resource respectively. The function **XtSetArg()** can also be used to set or retrieve widget resources after a widget is created.

The event-listening loop is the last statement in the example. It is implemented as an infinite C-language *for* loop as follows:

```
for ( ; ;) {
  XEvent event;
  XtNextEvent(&event);
  XtDispatchEvent(&event);
}
```

The **XtNextEvent()** function extracts the next event from the event queue, and hands it in to **XtDispatchEvent()** which invokes the appropriate event handler. Since this piece of code is always the same in all Xt Intrinsics applications, the Intrinsics provides it as a function:

XtMainLoop()

As may be seen from the above example, the Xt Intrinsics hide the details of programming with Xlib by creating primitives that internally call hundreds of Xlib functions. The Xt Intrinsics, and the X Widget Set provide application programmers with a library of user interface components that simplifies the task of writing applications, while allowing them to access Xlib functions when necessary.

The EDS UI architecture outlined in the next chapter follows the Xt programming format described previously. User interface components such as scrollbars, pull-down menus,

and icons are used to make user interaction with computers easier, and more productive. End users may modify the widget resources of EDS (widget colors, font sizes, ...) by using the X Resource Manager introduced at the end of chapter four.

CHAPTER 4

Interfacing to EDS

4.1 Designing a User Interface for EDS

In order to provide the end user with a way for supplying equations to EDS, as well as equation information, **TextEdit** widgets are created. Each of these widgets is modified to hold a one-line text field, and to adjust its width according to the font type specified in the resource database. Horizontal scrolling is enabled so as to allow the user to enter expressions wider than the width of each of the fields. **TextEdit** widgets support most **EMACS** editing commands, as well as text cutting and pasting. A **TextEdit** field is active when the pointer is inside it.

When the user enters an equation in infix notation in the appropriate **TextEdit** field, a series of events takes place based on the syntax of the equation provided. If the equation provided has the wrong syntax, the following series of events is executed:

- A **StaticRaster** widget is created, and a red cross-mark icon is displayed to suggest that user input cannot be accepted as entered.
- A **StaticText** widget reserved for error messages displays the error message related to the equation provided.
- The **TextEdit** widget cursor points at the location of the error, and enters a blinking state until the error is corrected.

On the other hand, if the syntax of the provided equation is valid, a different series of events is executed:

- A **StaticRaster** widget is created, and a green check-mark icon is displayed to suggest that user input is accepted.
- The equation entered is displayed inside a **TextEdit** widget that keeps a record of the entered equations. The **TextEdit** widget in this case is created as a child of a **ScrolledWindow** widget.
- The pointer moves to the next **TextEdit** field in order to relieve the user from using the mouse to get the same effect.

As humans, our reflexes have been trained to accept green as the "go-ahead" color, and red as the "stop" color. Based on this fact, green is used to color check-mark icons, and red to color cross-mark icons.

In order to label each of the **TextEdit** widgets, **StaticText** widgets are used. The color of a **StaticText** widget is inverted when the corresponding **TextEdit** widget is active. This feature is added to help the user distinguish between active and non-active widgets. Widget labels may be set by the end user in the **.Xdefaults** file (explained at the end of this chapter).

4.1.1 Pull-Down Menus

To provide the user with the ability to display equation flow graphs and binary trees, delete and add equations, create equation subsets, retrieve equation information, and operate on different sets/subsets of equations, pull-down menus are created. By selecting the appropriate menu entry, any of the following actions may be executed:

- Creating and deleting different sets of equations. A set of equations in this case is simply a group of equations supplied by the user.

- Generating flow graphs of one or more sets of equations.
- Converting equations from infix to postfix/prefix.
- Generating binary tree representation graphs of equations.
- Querying flow graph nodes.
- Querying flow graph links.
- Refreshing any of the **WorkSpace** widgets.
- Printing flow graphs and binary tree representation graphs.
- Quitting the application through a dialog box.

WorkSpace widgets are used to display equation flow graphs and binary tree representation graphs. Each **WorkSpace** widget is created as a child of a **ScrolledWindow** widget. The vertical and horizontal scrollbars of the **ScrolledWindow** widget are enabled, allowing large graphs to be drawn into the **WorkSpace**.

4.1.2 Dynamic Creation of PushButton Widgets

When a request is made to delete an equation from a given set of equations, or to form a new set of equations, a pop-up **RowCol** widget is created. This widget holds **PushButton** widgets whose labels represent equations previously entered by the user. These buttons are dynamically created since the number of equations is changing as the user enters or deletes equations. The number of **PushButton** widgets is dependent on the amount of heap space available.

A request made to delete a selected equation is not executed until the user confirms the action by selecting an "Enter" button. A "Cancel" button is also provided, and serves for a last-minute decision change. The same methodology is used when the user is picking equations to form an equation set.

4.1.3 Creating Pop-up Widgets

The first set of equations created by the user is displayed in a static **WorkSpace** widget. Additional equation sets are displayed in pop-up **WorkSpace** widgets that are created as children of **ScrolledWindow** widgets. Similarly, a request to generate the binary tree representation of a selected equation causes a **WorkSpace** widget to be created as a child of a **ScrolledWindow**. All pop-up widgets include menu entries for refreshing these widgets, or for destroying them. Users may pan up, down, left, or right in all **WorkSpace** widgets so as to bring a given flow graph, or binary tree area into focus. As stated previously, the number of pop-up widgets that can be created is totally dependent on the amount of heap space available.

4.2 The EDS Interface Widget Tree

A **BulletinBoard** widget is created as a child of the **TopLevelShell** widget to hold all the non-pop-up widgets stated in the previous sections of this chapter. The widget tree is summarized in figure 4.1, and the corresponding user interface is shown in figure 4.2. End users may easily modify many of the EDS interface resources (such as widget colors, font sizes, field labels, ...) as explained in the following section.

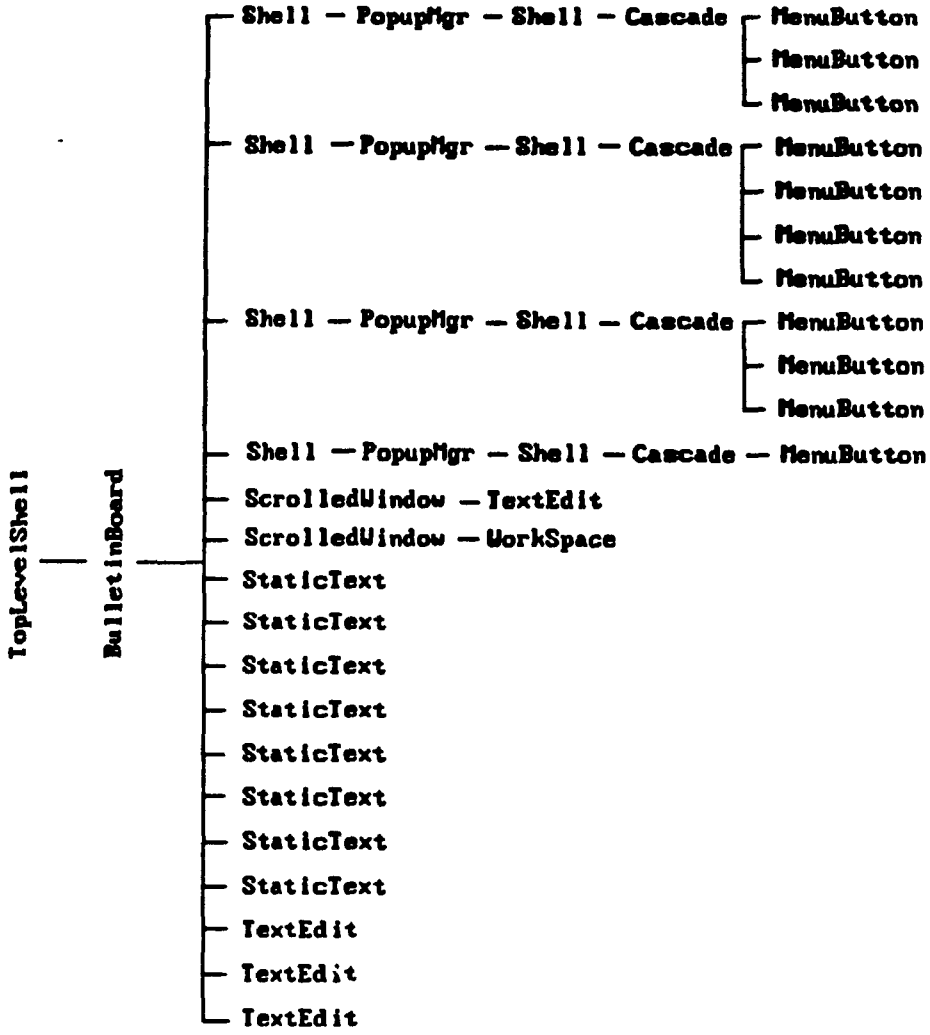


Figure 4.1 The EDS interface widget tree.

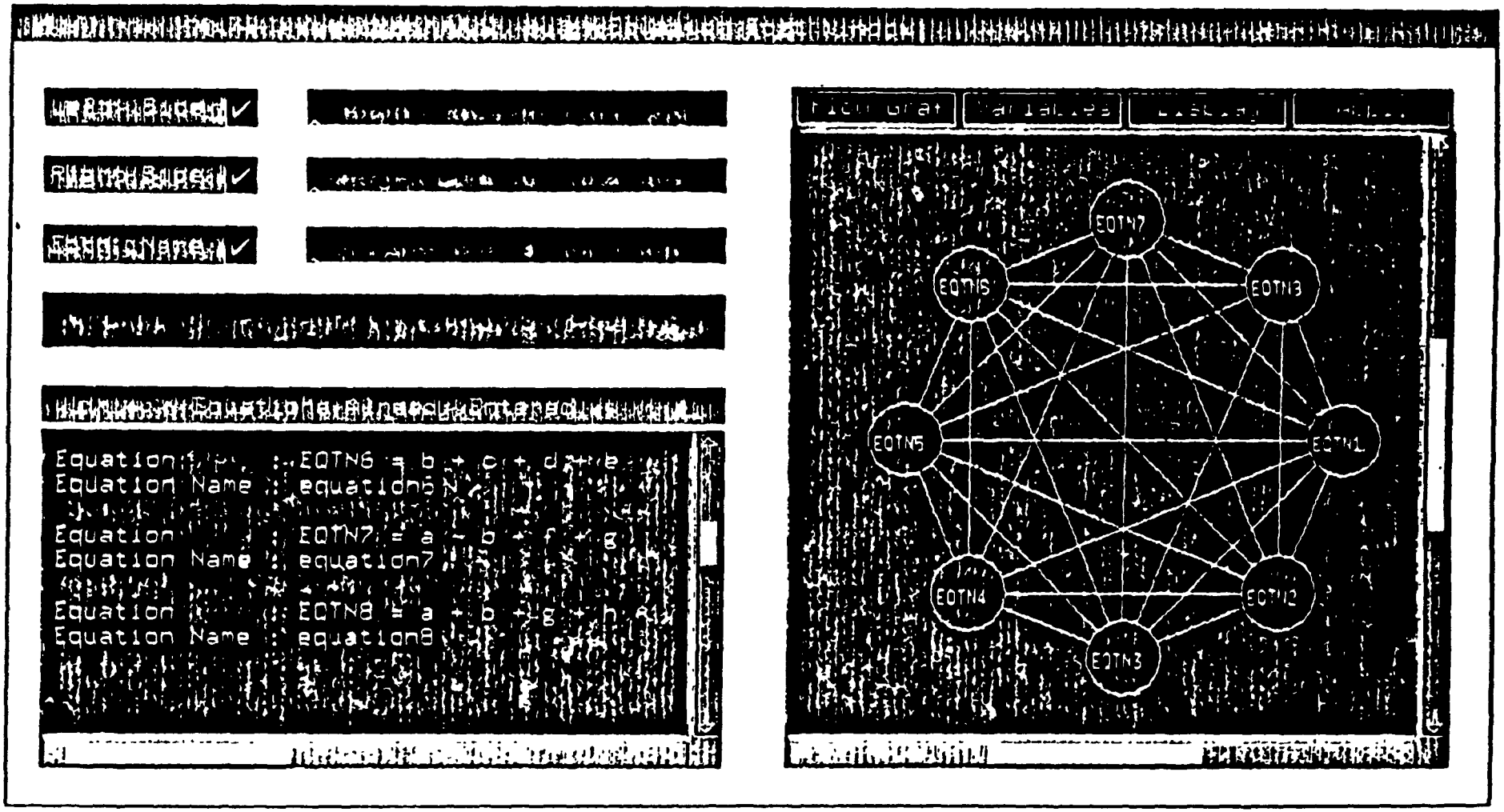


Figure 4.2 A snap shot of the EDS user interface.

4.3 Specifying Resources

Generally speaking, user interfaces are designed to make user interaction with computers easier and more productive. However, no matter how well the application programmer tries to anticipate the needs of the user, there is always someone who wishes to alter the behavior of a program [1]. Therefore, it is important that the user be able to customize an application to his/her own needs and tastes.

In order to provide both the application programmer and the end user with the ability to customize applications, a resource manager facility known as the X Resource Manager was designed by the developers of the X Window System.

To application programmers, X resources are data required by an application. In general, X resources are options that affect the behavior and look of an application. These include window IDs, colors, fonts, sizes, border widths, positions, etc.. . The X Resource Manager allows the user to modify most of these resources.

Every application and resource in X is expected to have a name and a class. A class indicates the general category to which a given entity belongs [1]. Consider the setup shown in figures 4.3 and 4.4. An end user can specify the foreground color of a toggle button named **toggle1** by simply specifying the following line in his/her **.Xdefaults** file residing in the user's home directory:

circuit.frame.work_space.toggles.toggle1.foreground: Green

Specifying the foreground color for **toggle1** consists of traversing the widget tree from top to bottom, and listing all the ancestors of **toggle1**. However, this resource specification rule may be inconvenient when large widget trees are involved. Instead, the X resource

manager provides the user with an asterisk '*' as a wild card character for representing any number of resource names or class names. Therefore, the foreground color of toggle1 may now be specified as follows:

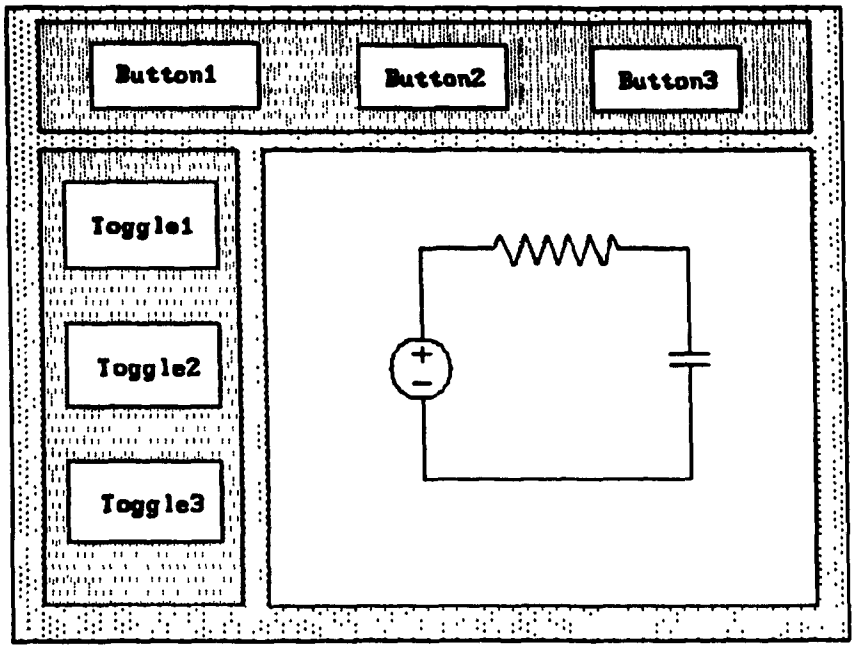


Figure 4.3 A typical example application.

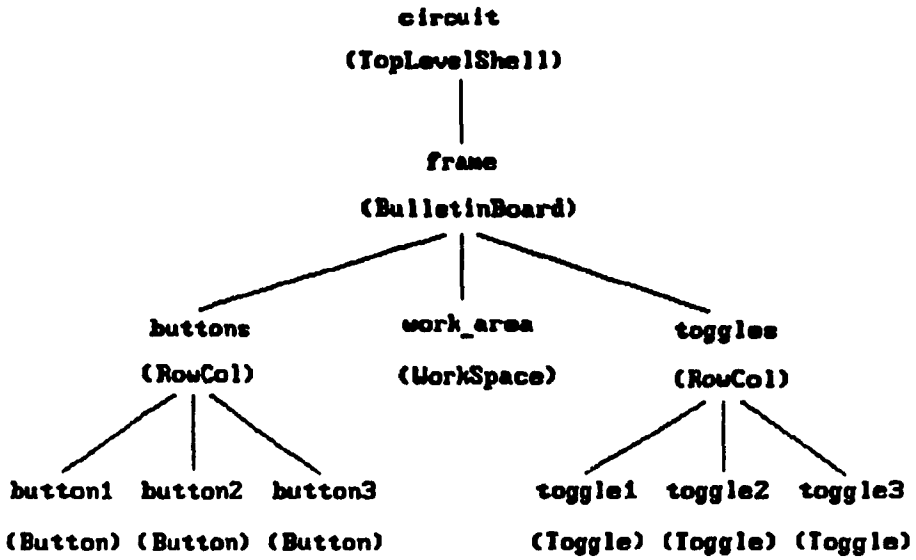


Figure 4.4 Widget tree for the example application of figure 4.3

circuit*toggle1: Green

Resource specification rules may be generalized by using class names instead of widget names:

***RowCol.foreground: Green**

The above specification implies that the foreground color of all widgets belonging to class **RowCol** should be green regardless of the application. The rules governing widget resource specifications are listed below:

- Entries in the **.Xdefaults** file prefixed by a dot ('.') are more specific, and have precedence over those prefixed with an asterisk ('*'). Therefore,

  ***circuit.foreground: Green**

  has precedence over

  ***circuit*foreground: Red**

- Resource names have precedence over class names. Therefore,

  ***toggle1.foreground: Green**

  has precedence over

  ***Toggle.foreground: Red**

- Resource names or class names have precedence over asterisks. Therefore,

  ***BulletinBoard*WorkSpace*Toggle*foreground: Green**

  has precedence over

  ***BulletinBoard*WorkSpace*foreground: Red**

- Entries are evaluated left to right. Items encountered first have precedence over successive items. Therefore,

  circuit*BulletinBoard*foreground: Green

  has precedence over

  circuit*BulletinBoard*foreground: Red

This chapter introduced the X Resource Manager through which widget resources such as color, font type, and widget labels may be specified by end users. The EDS user interface architecture was presented, along with the different widget types that were used. The next chapter discusses parsing of equations supplied by the user through **TextEdit** widgets. Equations are converted to different notations, and their binary tree representations are generated.

CHAPTER 5

Parsing User Input

Recognizing legal programs or expressions and decomposing them into forms suitable for further processing, is better known as *parsing*. Two general approaches are used for parsing, *top-down* and *bottom-up*. Top-down parsers look for a legal expression by first looking for parts of the legal expression, then looking for parts of parts, etc. until the pieces are small enough to match the input directly. Bottom-up parsers on the other hand, keep assembling pieces of the input in a structured way until a legal expression is constructed [10]. Top-down parsers are generally recursive, while bottom-up parsers are iterative. In this dissertation, I have used a bottom-up parser to parse equations entered by the user in **TextEdit** fields. Although I could have used *YACC/LEX* (Unix facilities for building compilers) to achieve the same goal, I have chosen not to, since the X Window System is being ported to PCs (mainly by Interactive Computers of California) running DOS, and where emulated Unix facilities might not be available.

5.1 Expression Grammar and Parse Trees

Before writing a parser program that parses infix equations, infix *grammar* must be defined. A very small subset of infix grammar that involves addition and multiplication is defined below:

[expression] = [term] | [term] + [expression]
*[term] = [factor] | [factor] * [term]*
[factor] = ([expression]) | v

The symbols *(*, *)*, +, and * are known as *terminal* symbols. On the other hand, *[expression]*, *[term]*, and *[factor]* are *non-terminal* symbols, and are internal to the grammar. The symbols =, and | are known as metasymbols, while the symbol *v* stands for any letter or digit. "=" may be read as "*is a*", *and* "|" as "*or*". Therefore, the first line of the infix grammar translates to "*an [expression] is a [term] or a [term] plus an [expression]*" [10].

The following example shows that the parse tree of A * (B + C) complies with the above grammar.

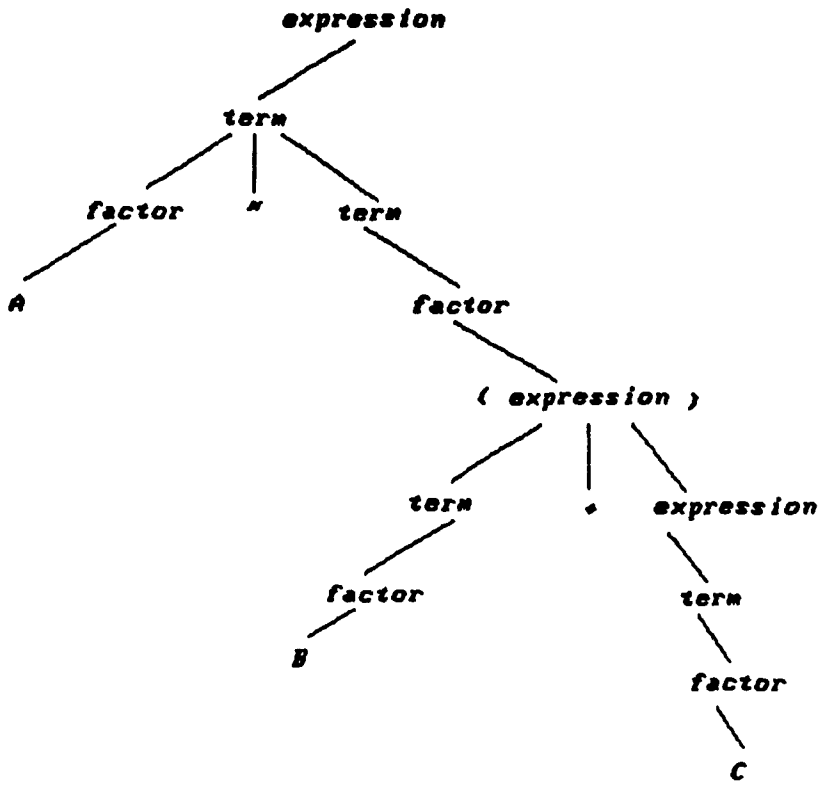


Figure 5.1 Parse tree for A * (B + C).

A parser accepts strings that exist in the language described by the grammar, and discards the rest. Bottom-up parsers do this by starting with the string at the bottom of the parse tree until they reach the nonterminal at the top. Top-down parsers do exactly the opposite, starting at the nonterminal top, and finishing at the bottom of the parse tree [10].

5.2 Converting Infix Equations to Postfix/Prefix

Although EDS converts infix expressions into prefix internally using Lisp, an infix-to-prefix C-routine generally does the conversion faster. The following section outlines an algorithm for converting infix equations to postfix/prefix. The sum of **A** and **B** is represented as **A + B**, where **A** and **B** are known as the operands, and '**+**' as the operator. This representation is referred to as *infix*. The sum of **A** and **B** can also be represented as follows:

+ A B	*Prefix*
A B +	*Postfix*

Prefix is known as *Polish notation*, while postfix is known as *reverse Polish notation*. The names are due to the inventor, the Polish mathematician Jan Lukasiewicz (1878-1956) [11]. Prefix notation can be thought of as a mirror image of postfix. Notice that in going from infix to postfix, parentheses are not required:

(A + B) * C	*Infix*
A B + C *	*Postfix*
A + (B * C)	*Infix*
A B C * +	*Postfix*

The order of the operands in the two previous expressions is the same. The first operand, **A**, of the infix expression **A + B * C**, can be immediately inserted into the postfix expression. the operator '**+**' cannot be inserted until after its second operand. Therefore, it must be stored

away until its proper insertion position is available. When the operand B is encountered, it is inserted directly after A. Now that two operands have been inserted, '+' still cannot be retrieved. This is due to the '*' operator, which follows and has precedence over '+'. In the infix expression (A + B) * C however, the closing parenthesis indicates that the '+' operation should be performed first [11].

5.2.1 Precedence Rules

From the previous example, it is obvious that precedence rules govern the infix-to-postfix conversion. By defining a boolean function, *prcd*(op1, op2), where op1 and op2 are characters representing operators, precedence rules can be set such that *prcd*('*', '+') is true, while *prcd*('+', '*') is false. Generalizing precedence rules to include delimiters, and most arithmetic and unary operators, the following rules emerge:

prcd('(', op2) = false for any op2.

prcd(op1, '(') = false for any op1 other than ')'.

prcd(op1, ')') = true for any op1 other than '('.

prcd(')', '(') = undefined.

prcd('*', op2) = true for any op2 $\in$ {'+', '-', '*', '/'}.

prcd('/', op2) = true for any op2 $\in$ {'+', '-', '*', '/'}.

prcd('+', op2) = true for any op2 $\in$ {'+', '-'}.

prcd('-', op2) = true for any op2 $\in$ {'+', '-'}.

prcd('^', op2) = true for any op2 other than '^'.

prcd(op1, op2) = true for any op2 other than '^', and for

op1 ∈ {'sin','cos','tan','cotan','log','ln','exp'}.

The infix-to-postfix conversion routine is presented below. *opstk* is the operator stack, and is initially empty. Procedures *push()* and *pop()* store and retrieve operators from the operator stack, while procedures *empty() and opnd()* check if the operator stack is empty, or if a token is an operand, respectively. Finally procedure *popandtest()* pops an element from the operator stack, and uses the boolean variable *und* to indicate whether stack underflow has occurred [11].

```
begin {procedure postfix}
    topsymb  := '+';
    opstk.top := 0; {start with an empty stack}
    position := 1;
    outlen   := 0;
    {scan symbols until encountering a blank}
    symb     := infix[position];
    while symb < > ''
     do begin
      if opnd(symb)
       then begin {operand is found}
        outlen     := outlen + 1;
        out[outlen] := symb;
       end
      else begin {if an operator is found}
       popandtest(opstk, topsymb, und)
       while(not und) and (prcd(topsymb, symb))
        do begin
         outlen     := outlen + 1;
         out[outlen] := topsymb;
         popandtest(opstk, topsymb, und)
       end
       if not und
        then push(opstk, topsymb)
       if und or (symb < > ')')
        then push(opstk, symb)
       else topsymb := pop(opstk)
      end
      if position < maxcols
       then begin
        position := position + 1;
        symb     := infix[position];
      end
      else symb := ''
```

```
    end
    while not empty(opstk)
     do begin
       outlen    := outlen + 1;
       out[outlen := pop(opstk)
     end
end
```

When an opening parenthesis is encountered, it is pushed onto the operator stack. This guarantees that an operator appearing after a left parenthesis is pushed onto the stack. On the other hand, when a right parenthesis is encountered, all operators since the opening parenthesis must be popped from the operator stack, and inserted into the postfix expression [11]. The next section discusses *parse tree* construction.

5.3 Binary Tree Representation of Equations

Given an equation in postfix notation, the corresponding binary tree representation, also known as the *parse tree*, may be constructed. The rules for constructing parse trees consist of placing the operator at the root of the tree, and the trees corresponding to the first and second operands at the left and right of the tree. As an example, consider the infix expression **(A + B) * (C + D)**. In postfix notation, this expression is written as **A B + C D + ***, and the corresponding parse tree is shown in figure 5.2.

The routine used for constructing such trees from a postfix expression is listed below. Every tree node has a left and a right link to other nodes. For an operand encountered while scanning a postfix expression, a node is created using the primitive *new*. An operand node has null links. On the other hand, a unary operator such as *log*, is represented as a node with one null link [10].

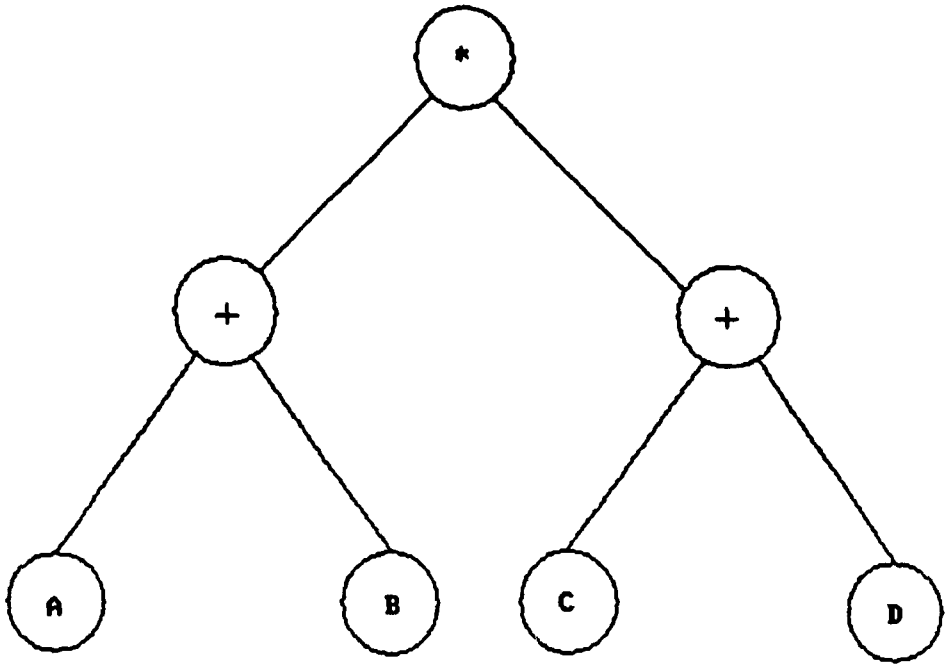


Figure 5.2 Parse tree for (A + B) * (C + D).

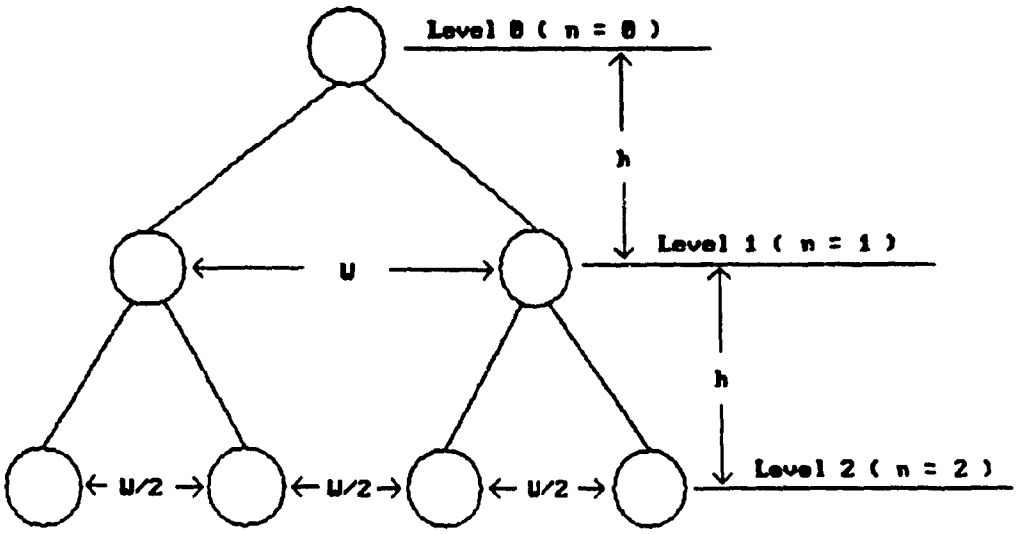
```
type link = ^node;
  node = record  info : char; l, r : link end;
var x, z : link;
      c : char;
begin
stackinit;
new(z); z^.l := z; z^.r := z;
repeat
  repeat read(c) until c< >' ';
  new(x); x^.info := c;
  if (c = '*') or (c = '+') or (c = '-') or (c = '/')
    then begin x^.r := pop; x^.l := pop end
    else if (c = 'log') or (c = 'sin') or .....
    then begin x^.r = pop; x^.l := z end
    else begin x^.r := z; x^.l := z end;
  push(x)
until eoln;
```

The procedures *stackinit*, *push*, and *pop* initialize, push elements onto the stack, and pop elements from the stack. They are defined in appendix C. The next section describes how parse trees are drawn into **WorkSpace** widgets.

5.3.1 Drawing Parse Trees

Once the parse tree of an equation is constructed, it can be drawn into a **WorkSpace** widget. The procedure used to draw a parse tree consists of traversing the postfix tree in level order, where tree nodes are read down from top to bottom, and from left to right. The sequence *, +, +, A, B, C, D refers to the level order traversal of the parse tree shown in figure 5.2 [10]. When a tree node is read, calls to the Xlib primitives **XDrawArc()** and **XDrawString()** are made to draw a circle and a string centered inside it respectively. The string drawn simply represents the contents of a node. At the same time, the left and right links of the node are drawn (assuming that the next level nodes connected to the current links are not null) using the Xlib primitive **XDrawLine()**. The end points of the segments are determined from the fact that as one moves from one level of the tree to the next, the number of null and non-null nodes increases by a factor of 2^n, where n is the level number.

The next chapter discusses equation sets, and the methods used to store and retrieve equation information graphically from flow graphs representing these sets. Dynamic generation of flow graphs representing user chosen equation sets are also discussed.



The number of null and non-null nodes at any level is given by 2^n.

Figure 5.3 Drawing parse trees.

CHAPTER 6

Equation Sets and Equation Flow Graphs

After user supplied equations are parsed, they are used to populate linked lists of data structures that hold detailed information about these equations. Data can then be retrieved from the data structures, and used to generate equation flow graphs representing stored equations. In addition to providing the user with the ability to generate flow graphs for different sets of equations, the package also allows the user to add or delete elements from these sets of equations. The addition or deletion of equations is graphically reflected in the WorkSpace widgets where equation flow graphs are drawn.

6.1 Storage and Retrieval of Equation Information

After the syntax of a supplied equation is validated through the parser, the equation is stored in a C-language data structure. As more equations are entered, a linked list of data structures is formed that represents these equations. When a request to draw the flow graph of a given set of equations is made, the linked list is copied into a dynamically allocated array of structures that has the following format:

```
typedef struct Eqt {
    int     x;          /* x-coordinate of equation node. */
    int     y;          /* y-coordinate of equation node. */
    int     infien;     /* number of tokens in infix eqn. */
    int     postlen;    /* number of tokens in postfix eqn. */
    int     preflen;    /* number of tokens in prefix eqn. */
    int     varlen;     /* number of variables in equation. */
    char    *eqlside;   /* left side of equation. */
    char    *eqrside;   /* right side of equation. */
    char    *eqname;    /* name of equation (optional). */
    char    **infix;    /* equation in infix form. */
```

```
    char      **postfix;        /* equation in postfix form. */
    char      **prefix;         /* equation in prefix form. */
    char      **var;            /* variables in equation. */
    struct Tiedto *tiedto;      /* Equations that may be related to
                                   this equation. */
} Eqt;

typedef struct Tiedto {
    char      *eqlside;         /* Left side of equation. */
    struct Vars  *vars;         /* variables in common. */
    struct Tiedto *next;        /* link to next equation. */
    struct Tiedto *previous;    /* link to previous equation. */
} Tiedto;

typedef struct Vars {
    char      *varname;         /* variable. */
    struct Vars  *next;         /* link to next variable. */
    struct Vars  *previous;     /* link to previous variable. */
} Vars;
```

Before describing the above structures, the model used to represent equations must be defined. Consider the following equations:

$V = I * R$ (*Ohm's Law*)

$P = I^2 * R$ (*Power Dissipation*)

These equations may be graphically represented as two circles (equation nodes), and a line segment (link) connecting them. The setup is shown in figure 6.1.

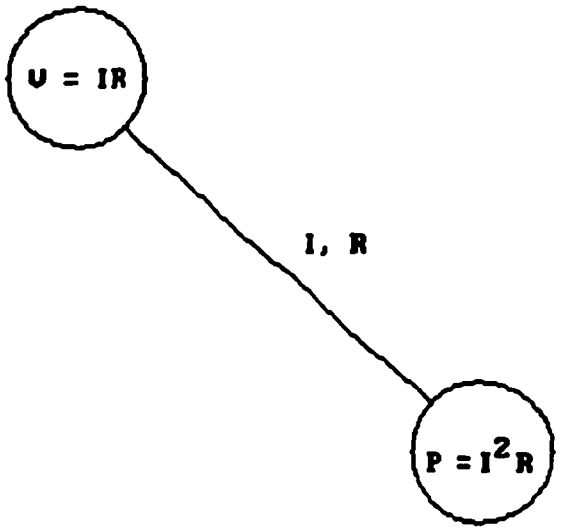


Figure 6.1 Flow graph representation of $V = I * R$ and $P = I^2 * R$.

Since **I** and **R** are common to both equations, a *link* is drawn between the *nodes* representing these equations. A node in this case is simply a circle whose inside holds an equation string. On the other hand, a link shows that a relationship exists between two given equations. The variables common to the two given equations are shown next to the link. The combination of nodes and links is known as a *flowgraph*.

Going back to the data structures presented earlier, **x** and **y** of structure **Eqt** represent the coordinates of the center of an equation node, while **inflen, postlen,** and **preflen** represent the number of tokens in the infix, postfix, and prefix equations respectively. The number of variables in the equation is stored in **varlen**. The left side, right side, and name of the infix equation are pointed at by the structure members **eqlside, eqrside,** and **eqname** respectively. The infix tokens, postfix tokens, prefix tokens, and variables making up the equation are stored

in arrays of strings pointed at by the structure members **infix**, **postfix**, **prefix**, and **var** respectively. Finally, the **tiedto** member forms a linked list of equations tied to the current equation. Variables common to two given equations are stored in the **Vars** linked list.

6.1.1 Extracting Related Equations

When the user requests that a flow graph of a given set of equations be generated, a search through the array of structures representing the set of equations must be conducted internally in order to single out related equations. The search consists of a pattern matching algorithm that tries to match every variable of a given equation with the variables of the remaining equations in the set. Every two equations are compared only once. The search continues until all equations have been scanned. For a set of **n** equations, the number of equation comparisons performed is given by:

$$(n-1)+(n-2)+\ldots+1=(n^2-n)/2$$

To illustrate this algorithm, consider the example equations presented in the previous section. Assume that the set of equations consists of:

$$\{V=I*R,\quad P=I^2*R\}$$

The algorithm starts by picking the first variable (**I**) of the first equation (**V**) in the set, and scans equation **P** in search for the same string. Since **I** is also a variable of equation **P**, the search is successful and the **Eqt** structure member **tiedto** is now initialized to **P**. Next, variable (**R**) of the first equation is picked, and a similar search is conducted. After variables

of the first equation are scanned, the search is complete. By now, the data structures have been filled, and in particular, the structure **tiedto** contains information equivalent to the following:

$$V \cap P = \{ I , R \}$$

The **prefix** member of the **Eqt** data structure presented earlier is used to fill out the *rule* slot of an equation schema in EDS. On the other hand, the **vars** structure member is used to fill out the *has-variables* equation schema slot. At this stage, EDS solves for all variables in the input equation, and stores the corresponding clauses in the *clauses* equation schema slot (assuming that the symbolic solver finds a solution to each of the variables in the input equation) [9].

6.1.2 Generating Flow Graphs

The centers of the equation nodes shown in figure 6.1 occupy the ends of a line segment. For an equation set consisting of three equations, node centers occupy the vertices of a triangle. A four-equation set has its node centers located at the vertices of a square etc.. The algorithm for generating the equation node centers is outlined below:

```
PI = 3.141593;
angle = 2.0 * PI / number_of_equations;
begin i:=1 to number_of_equations do
 x[i] = x_center + radius * cos(i * angle);
 y[i] = y_center + radius * sin(i * angle);
 end;
```

The node centers lie on a circle of center *(x_center, y_center)*, and of radius *radius*, and divide the circle perimeter into *number_of_equations* equal arcs. Each of these arcs has *angle* as its angle, taken from the center of the circle.

The algorithm used for generating equation flow graphs consists of scanning the array of Eqt structures, and of drawing nodes at specified x and y member coordinates. At the same time, the algorithm checks the structure member tiedto to determine if the current equation is related to other equations. Links are drawn between related equations.

Equation flow graphs are drawn inside WorkSpace widgets. The Xlib graphics primitives XDrawString(), XDrawLine(), and XDrawArc() for drawing text, lines, and arcs are used. The contents of WorkSpace widgets are maintained by drawing graphics both inside WorkSpace widgets, and into *pixmaps*. An X pixmap is simply an area of memory similar to a rectangular region on the screen, except that it is stored in *off-screen* memory, and is not visible to the user [1]. Like a screen, a pixmap has a width, height, and depth. The function

XCreatePixmap(display, drawable, width, height, depth)

creates a pixmap of *width* by *height* pixels, having *depth* number of planes.

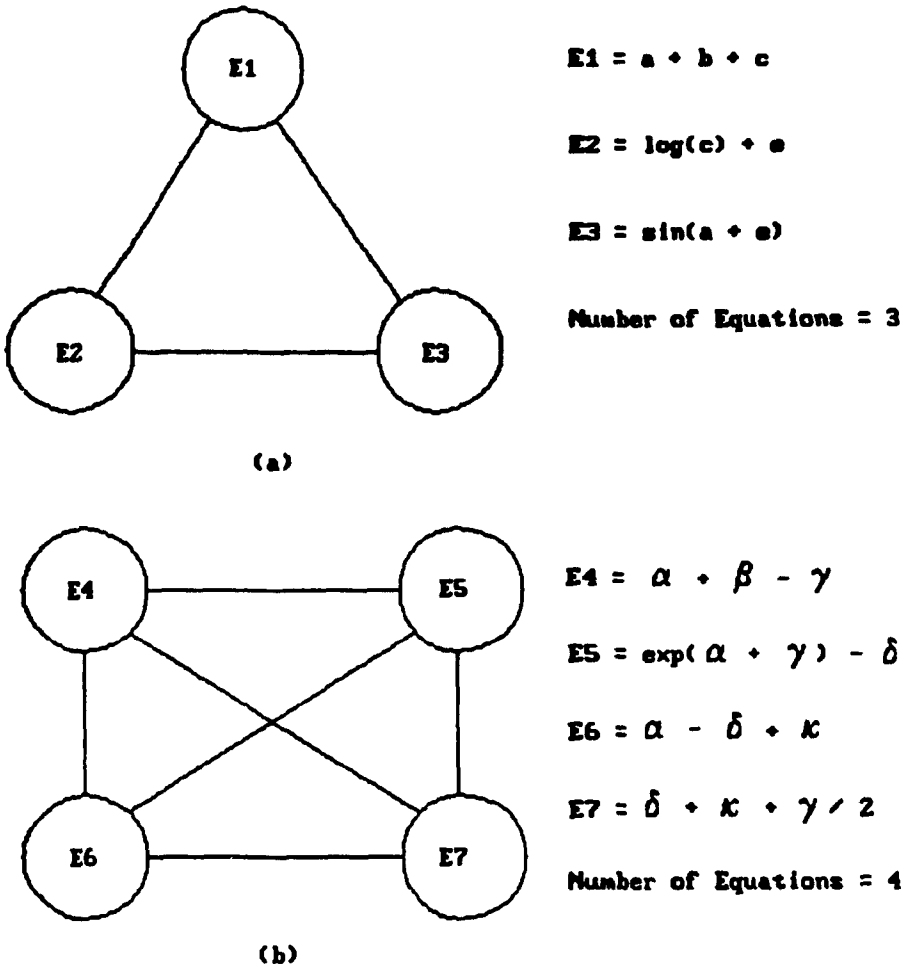


Figure 6.2 Example flow graphs.

For a large set of equations where each equation is related to all of its counterparts in the set, the flowgraph becomes crowded with intersecting node links. This problem is solved by generating flowgraphs for subsets of the original equations set. Flowgraphs of equation subsets are displayed in pop-up widgets.

6.1.3 Retrieving Equation Information

At any time, the user can query a flow graph for equation information. One level of hypertext is provided; a pointer event inside any of the flow graph nodes causes a **StaticText** pop-up widget to appear. This widget holds detailed information about the node equation. The information displayed includes:

- The name of the equation.
- The equation in infix form.
- The equation in postfix form.
- The variables that form the equation.
- Equations that this equation is related to.
- Variables common to this, and each of the related equations.

The location of every pointer event taking place inside a **WorkSpace** widget is checked to determine if the event happened inside one of the flow graph nodes. A routine that scans the **Eqt** structure array is used to determine if the distance from the point where the event occurs, to the center of each of the equation nodes, is less than the radius of each of the nodes.

Pointer events occurring near a link cause a **StaticText** text widget pop-up to be displayed. The pop-up widget displays the variables common to the two equations whose nodes are connected through the link. The following section describes the procedure used for determining whether a pointer event occurs inside any of the thin rectangles surrounding equation links.

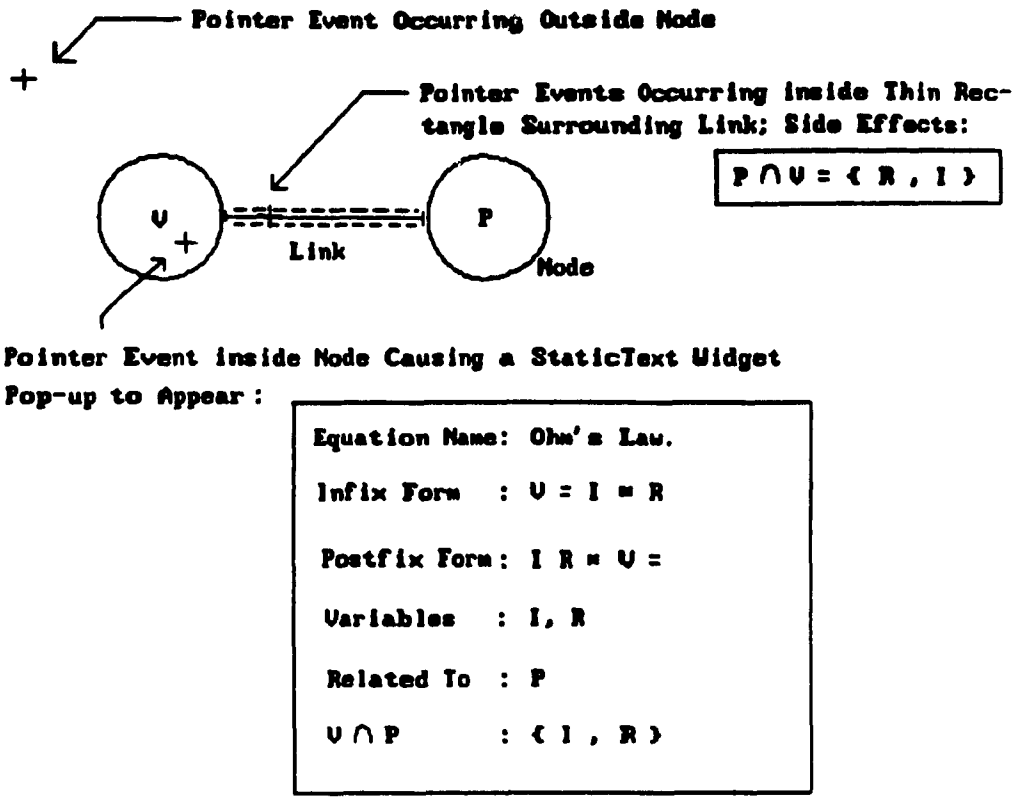


Figure 6.3 Side effects of pointer events.

6.1.4 Inclusion in a Polygon

Given a pointer location and a polygon (in this case a rectangle) surrounding a flow graph link, a search must be conducted to determine if the pointer location is inside or outside the polygon. A solution to this natural problem consists of drawing a long line segment from the pointer location in any direction, such that the endpoint of the segment is guaranteed to

be outside the polygon. If the number of intersections of the polygon with the line segment is odd, the pointer location must be inside the polygon. If it is even, the pointer location is outside the polygon.

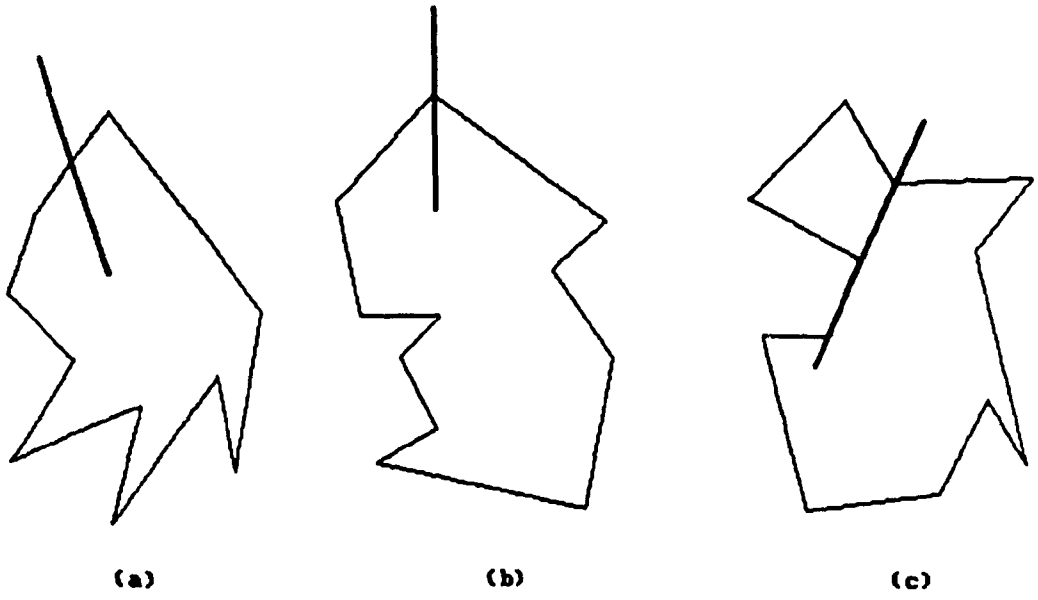


Figure 6.4 Different point-in-polygon cases.

However, the situation is not so simple because some intersections might occur at the vertices of the polygon as shown in figure 6.4 (b). The line segment might also align with one of the edges of the polygon as shown in figure 6.4 (c). Therefore, the need to handle all these cases must be addressed.

An algorithm that covers all the particular cases stated above consists of traveling around the polygon, and incrementing an intersection counter whenever the test line is crossed from one side to the other. Points that fall on the test line are ignored [10].

```
function inside(t:point):boolean;
 var count,i,j:integer;
    lt,lp:line;
 begin
 count:=0; j:=0;
 p[0]:=p[N]; p[N+1]:=p[1];
 lt.p1:=t; lt.p2:=t; lt.p2.x:=maxint;
 for i:=1 to N do
   begin
   lp.p1:=p[i]; lp.p2:=p[i];
   if not intersect(lp,lt) then
     begin
     lp.p2:=p[j]; j:=i;
     if intersect(lp,lt) then count:=count+1;
     end;
   end
 inside:=((count mod 2)=1);
 end;
```

Polygon vertices are stored in the *p[1..N]* array. The *intersect* function simply checks if two line segments intersect, and is listed in appendix C. The variable *j* is maintained as the index of the last point on the polygon known not to lie on the test line. The algorithm assumes that *p[1]* is the point with the smallest x-coordinate among all the points with the smallest y-coordinate [10].

6.2 Memory Management

At any time, equations may be added or deleted from any set of equations already entered. Flow graphs are informed of the changes in equation sets through **Callbacks**. Therefore, flow graphs always reflect these changes. When adding new equations, heap space must be allocated to accomodate the newly formed data structures. This is done using the C-language library calls **calloc()**, and **malloc()**. When an equation, or a whole set of equations is deleted, the associated heap space must be freed. Also, when a pop-up widget is destroyed,

it must be first removed from the linked list of pop-up widgets, and then freed. Routines for freeing the structures outlined at the beginning of this chapter are designed to search for all the allocated heap space inside these structures, and to release it.

6.3 Printing Flow Graphs and Parse Trees

Using the **xwd** client application provided with the X Window System core distribution, a snap shot of any window on the screen may be taken. The output of the command consists of a bitmap representing a selected window, and may be directed to a file. Using the **xwd** picture format, picture files are converted to the following formats:

* HP Laser Jet Series II.
* HP Paint Jet.
* QMS Postscript.

The converted files can then be queued to any of the above printers. Sample pictures of the interface were generated on an HP Paint Jet printer, and are shown in chapter seven.

In this chapter, the data structures used in the interface program were presented, and a method for generating flow graphs of equations was developed. Subsets of equations may be created, and their corresponding flow graphs may be shown in pop-up widgets. Equations may be added or deleted from any set of equations. The next chapter presents a summary of the work done in this dissertation. Conclusions, as well as suggestions for further enhancements are also included.

CHAPTER 7

Conclusions

7.1 Thesis Summary

The EDS user interface provides users with a mechanism for graphically representing algebraic models in terms of flow graphs. Flow graphs of different sets of equations may be generated, and equations may be added to, or deleted from equation sets. An equation set is represented by a pop-up widget that contains a flow graph depicting the set. The number of equation sets that may be created is limited by the amount of heap space available on the host machine, since pop-up widgets are created dynamically.

The EDS user interface supports one level of hypertext; users may query flow graph links in search of information about variables common to related equations. Similarly, flow graph nodes may be queried for equation relationships, variables in the equations, as well as different equation notations. Querying is triggered by clicking on the flow graph object in question (node or link). Object information is displayed inside pop-up widgets.

As the number of related equations in a set increases, the flow graph representing the given set becomes densely populated with equation links. Flow graphs of subsets of the given equation set may therefore be generated.

Figure 7.1 is a picture of the EDS user interface in which four mutually related equations are entered through the **TextEdit** widget fields. The flow graph of all four equations is displayed in the **WorkSpace** widget area below the pull-down menus. Equation links are shown in red, while equation nodes are drawn in green. Figure 7.2 shows pop-up widgets holding variables

common to equations connected by each of the links. Equation information is also displayed in pop-up widgets as shown in figure 7.3. Popup widgets holding link or node information may be collapsed by simply clicking inside them.

In figure 7.4, all pop-up widgets are collapsed, and new equations are added to the main equation set. In figures 7.5 and 7.6, an equation is selected from the main set of equations, and a request to generate the binary tree is made. The resulting binary tree for the selected equation is displayed in a pop-up widget as shown in figure 7.7.

As the number of equations increases, a 'crowding' of equation links takes place, especially when equations in a given set are densely interrelated. This may be obvious from figure 7.4. In this case, equation subsets may be formed as shown in figures 7.8 through 7.11. Equation subsets are displayed in pop-up **WorkSpace** widgets.

Equations may be added or deleted from a given set of equations, as shown in figures 7.12 and 7.13, where equation5 is deleted from the main set of equations, and replaced by equation9.

Text cutting and pasting may be used between the different **TextEdit** widget fields that the interface uses. Vertical and horizontal scrolling is implemented in all **WorkSpace** widgets, allowing for top, bottom, left, and right panning. Additional input and output **TextEdit** widget fields can be easily added to the interface data structures, therefore allowing it to display more equation information when needed.

The actual interfacing of EDS to the EDS UI was not possible, due to the fact that the **Knowledge Craft** shell that EDS uses to run is no longer available (*Centre de Recherche Informatique de Montréal* sold its copy of KC which EDS has been accessing from the *Mcgill CADLab*).

The EDS user interface is portable, and runs on a variety of X11.R2 platforms. It was tested on Apollo and Sun workstations running SR10.1 and SunOS 4.0 respectively. After all, portability is one of the major advantages of using the X Window System. It is interesting to note however, that simple graphical operations such as panning in **WorkSpace** widgets, or moving up or down within pull-down menus cause extensive paging (memory to hard disk mapping) on most workstations running X, and lead to slow performing applications. This is a major drawback of X, especially when applications are run across networks that are subject to frequent delays.

Another advantage of using the X Window System is the user-controlled, X Resource Manager. Colors, menu labels, as well as other widget resources may be easily customized by end users (**.Xdefaults** file in the user's home directory). The software written for this dissertation is available as public domain, and is located in directory *thesis* on *noujeim@dwight.ee.mcgill.ca*.

7.2 Suggestions for Future Work

Although the X Toolkit provides application programmers with a higher level library built on top of Xlib, it still involves a fair amount of detail. A programming tool that hides many of these details, without restricting access to the Xlib library, is bound to increase the productivity of application programmers.

A well-suited language that simplifies X Toolkit programming considerably is Lisp (although Smalltalk might be another potential candidate). By creating Lisp functions that internally interface to X Toolkit routines, higher-level abstractions are created that simplify

the development task, and increase programming efficiency. Consider the following lines of code that create a StaticText widget, set its location, size, and border color by directly using the X Toolkit:

```
.
.
Arg    wargs[5];
Widget toplevel, w;
int    i = 0;

toplevel = XtInitialize(argv[0], "Example", NULL, 0,
                &argc, argv);
/* Set the StaticText widget attributes. */
XtSetArg(wargs[i], XtNx, 100);          i++;
XtSetArg(wargs[i], XtNy, 200);          i++;
XtSetArg(wargs[i], XtNwidth, 120);      i++;
XtSetArg(wargs[i], XtNheight,150);      i++;
XtSetArg(wargs[i], XtNborderColor, "Red"); i++;
w = XtCreateManagedWidget("static", XwstaticTextWidgetClass,
                toplevel, wargs, i);
.
.
```

The equivalent code implemented in Lisp might look as follows:

```
(setq shell (initializeXt "Example"))
(createStaticTextWidget shell "static" 100 200 120 150 "Red")
```

Every Lisp function used to create a widget contains the class name of the widget, preceded by the word 'create'. Therefore, the function **createStaticTextWidget** simply creates a **StaticText** widget. **StaticText** widget resources are specified as arguments to the **createStaticTextWidget** function. The parent id argument is specified first, followed by the name of the widget, its location, size, color, etc.. . The X Window System convention of first specifying the location of a window, followed by its size, and of specifying x-coordinates before y-coordinates is maintained.

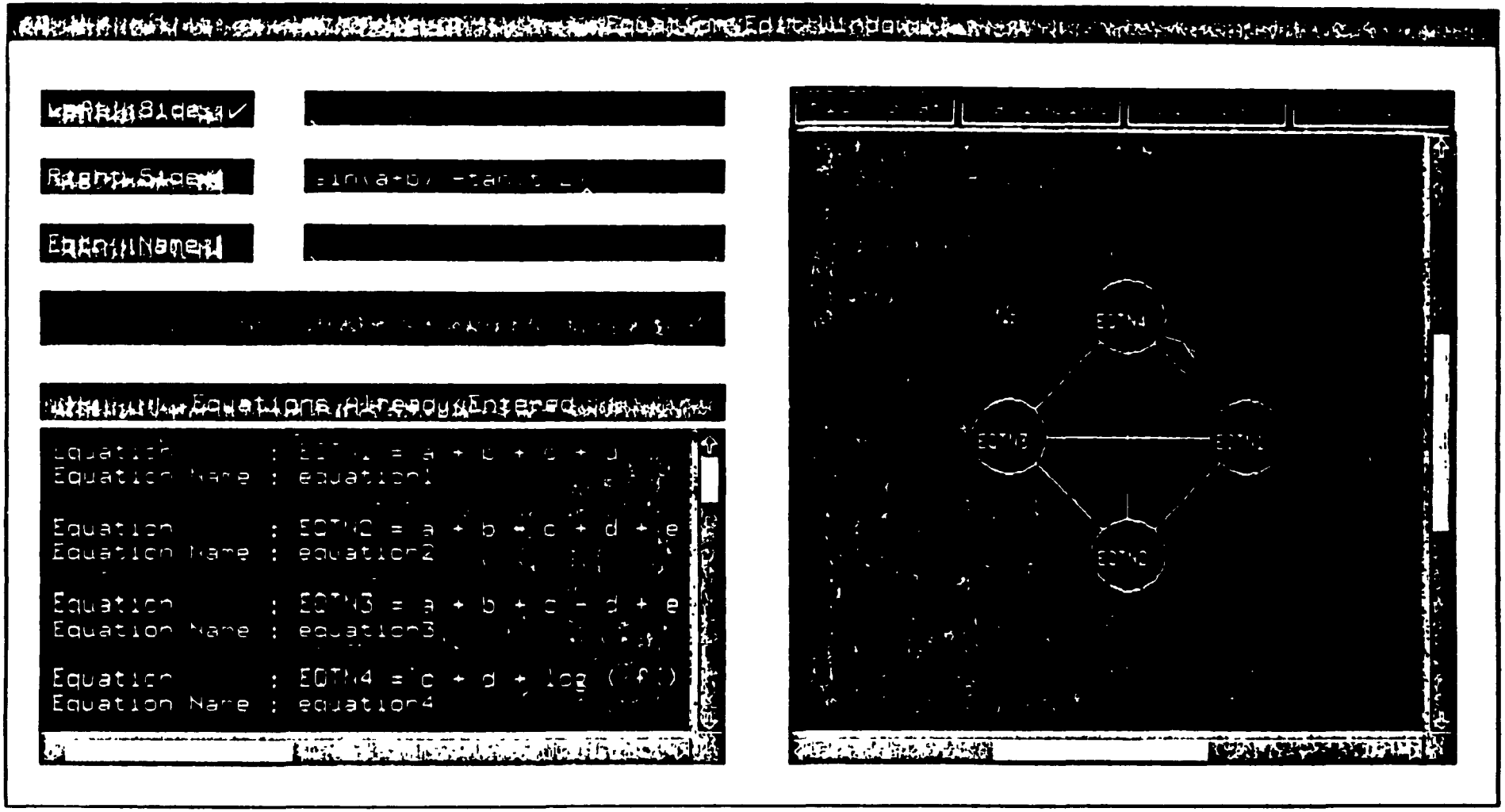


Figure 7.1 A typical four-equation flow graph generated by the EDS user interface.

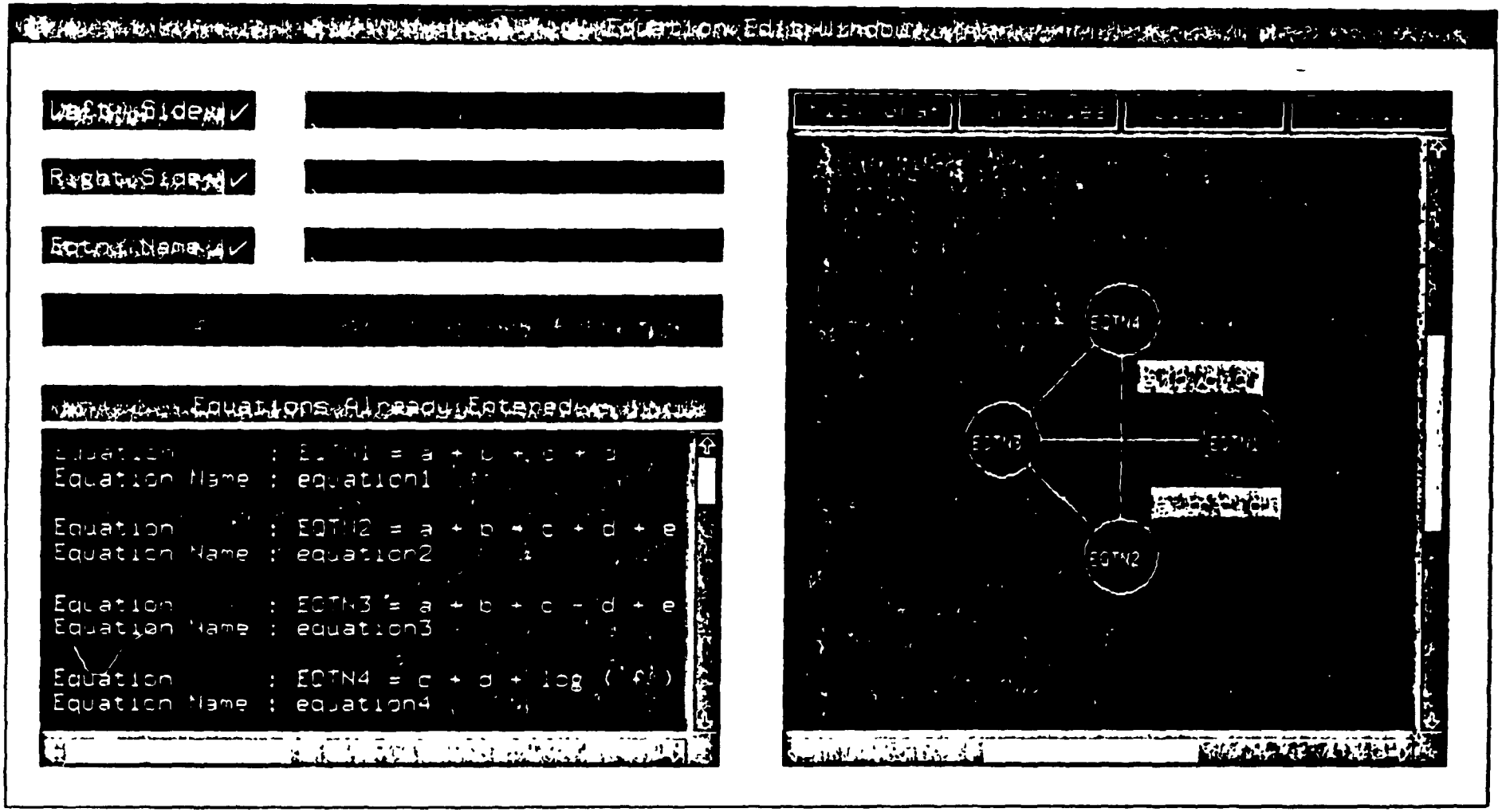


Figure 7.2 **Querying flow graph links. Variables common to related equations are displayed in pop-up StaticText widgets.**

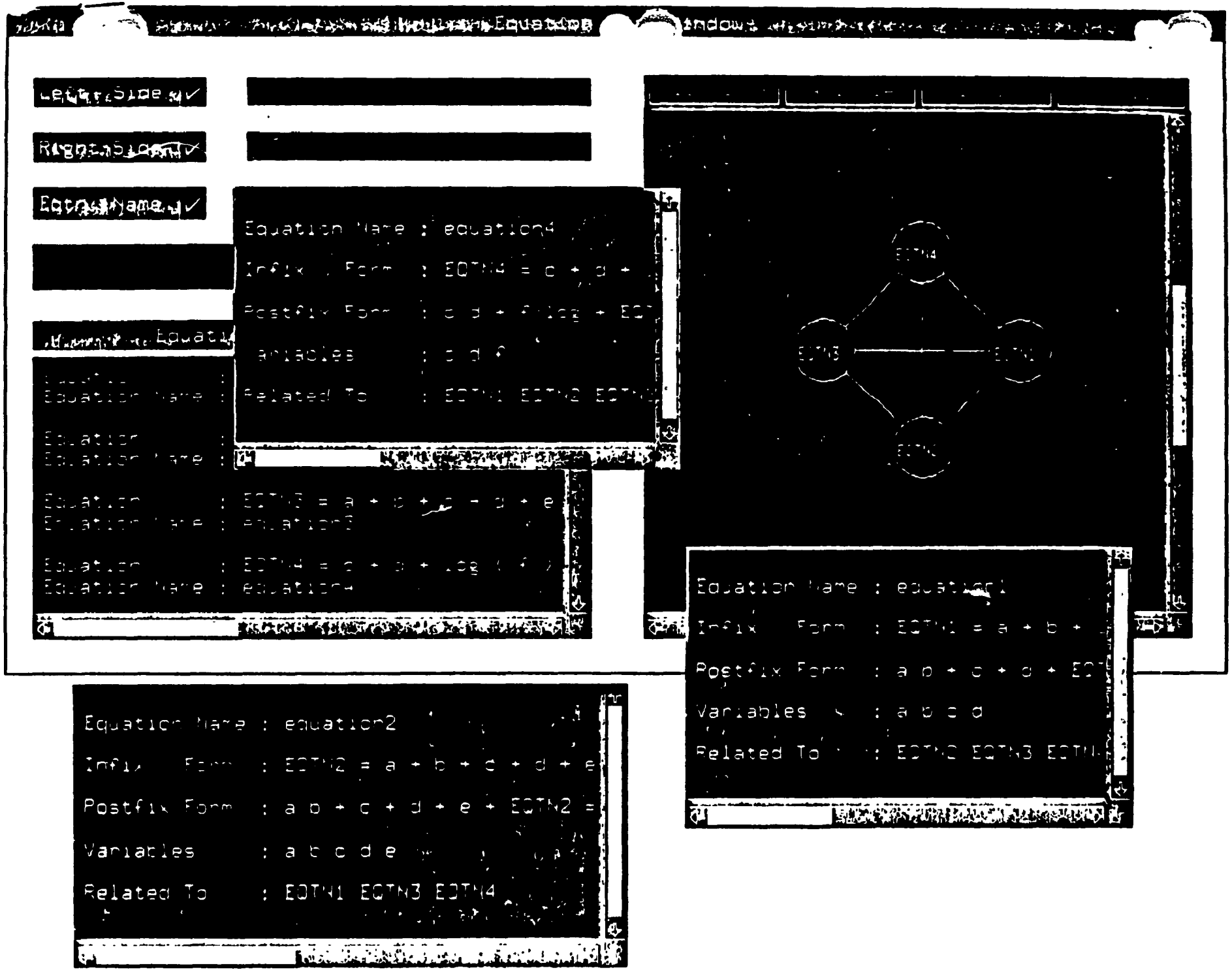


Figure 7.3 Querying flow graph nodes. Equation information is displayed in pop-up TextEdit widgets.

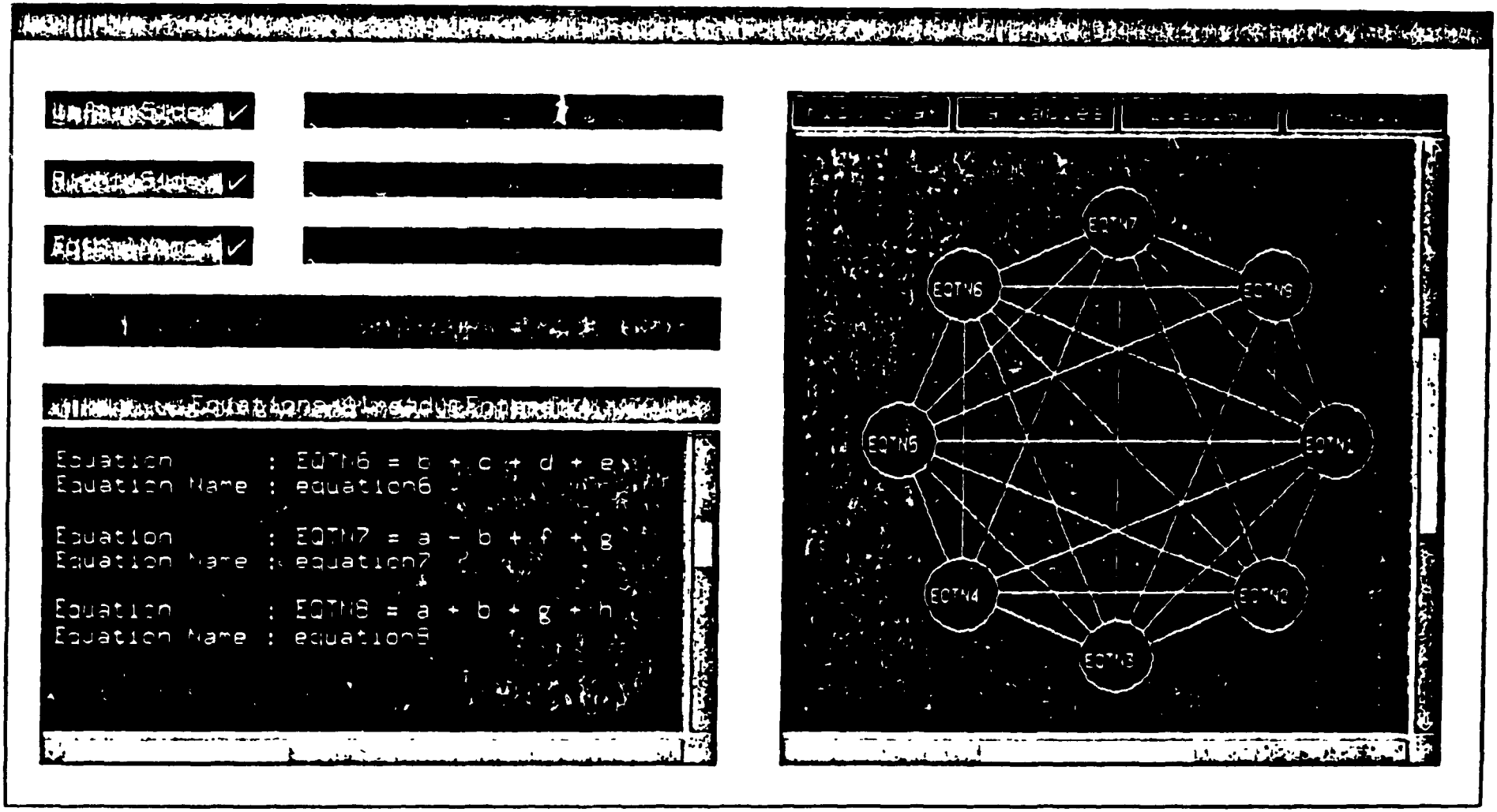


Figure 7.4 **The equation set of figure 7.1 is expanded to include more equations. The corresponding flow graph shows an increase in link 'crowding'.**

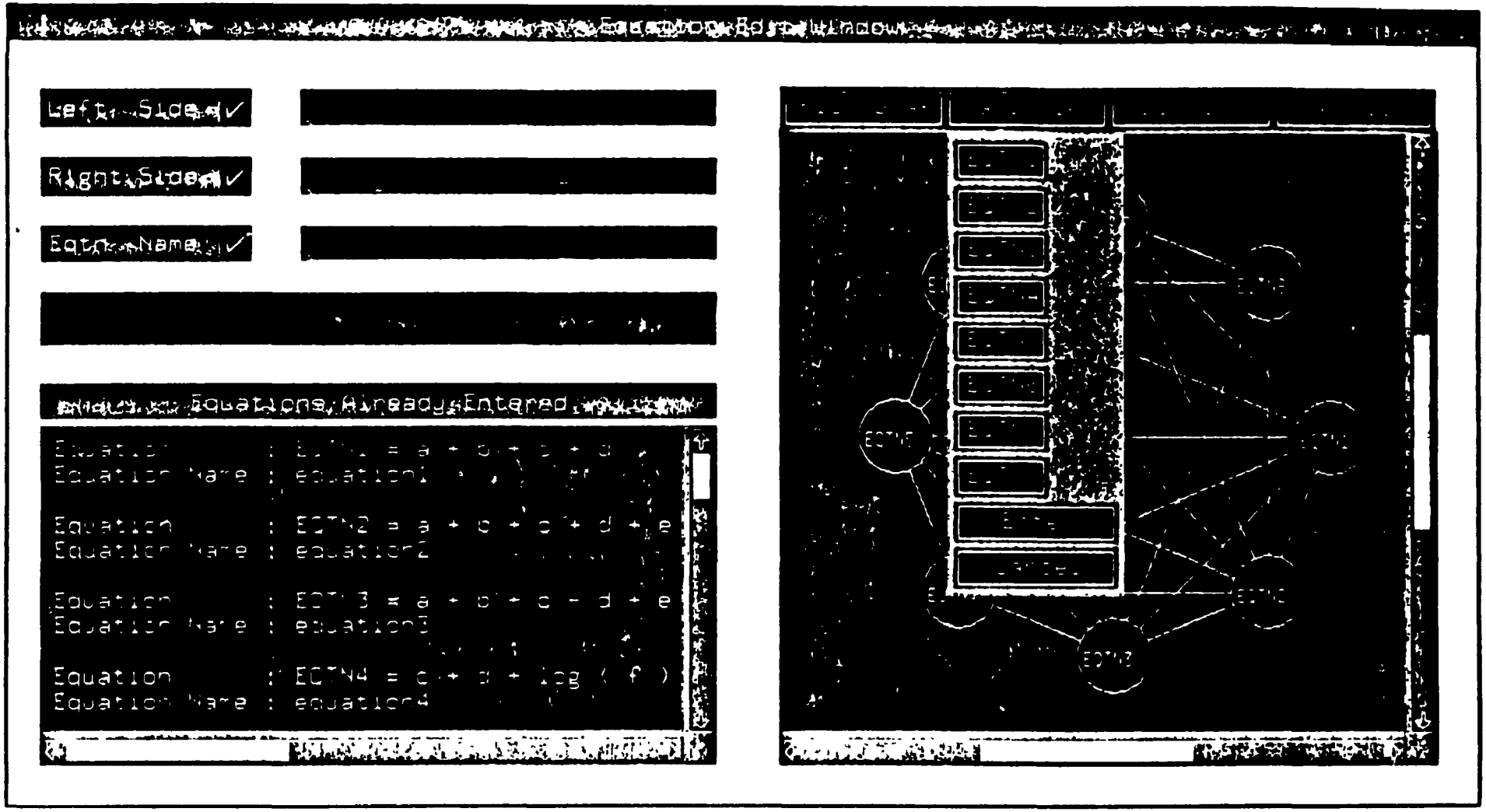


Figure 7.5 **A set of PushButton widgets representing each of the equations in the current set. Equation sets are implemented as RowCol widgets holding PushButtons.**

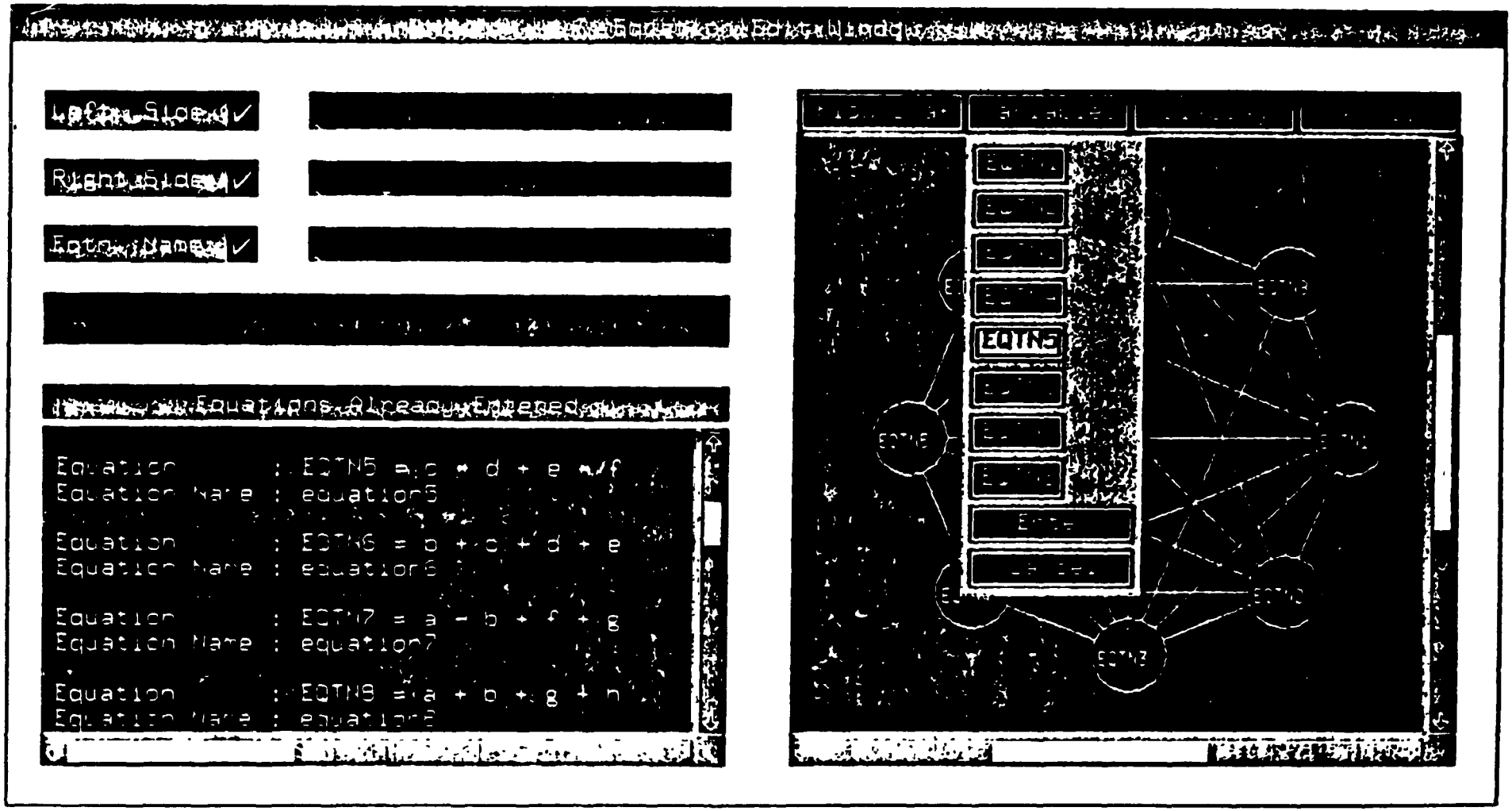


Figure 7.6 **Highlighting is used to show selected equations in a set. The operation to be performed on a selected equation(s) must be chosen first, before any set of PushButtons can be displayed. The set of equations shown was displayed after the user picked the sub-menu entry 'Binary Tree'(not shown), from the 'Variables' pull-down menu.**

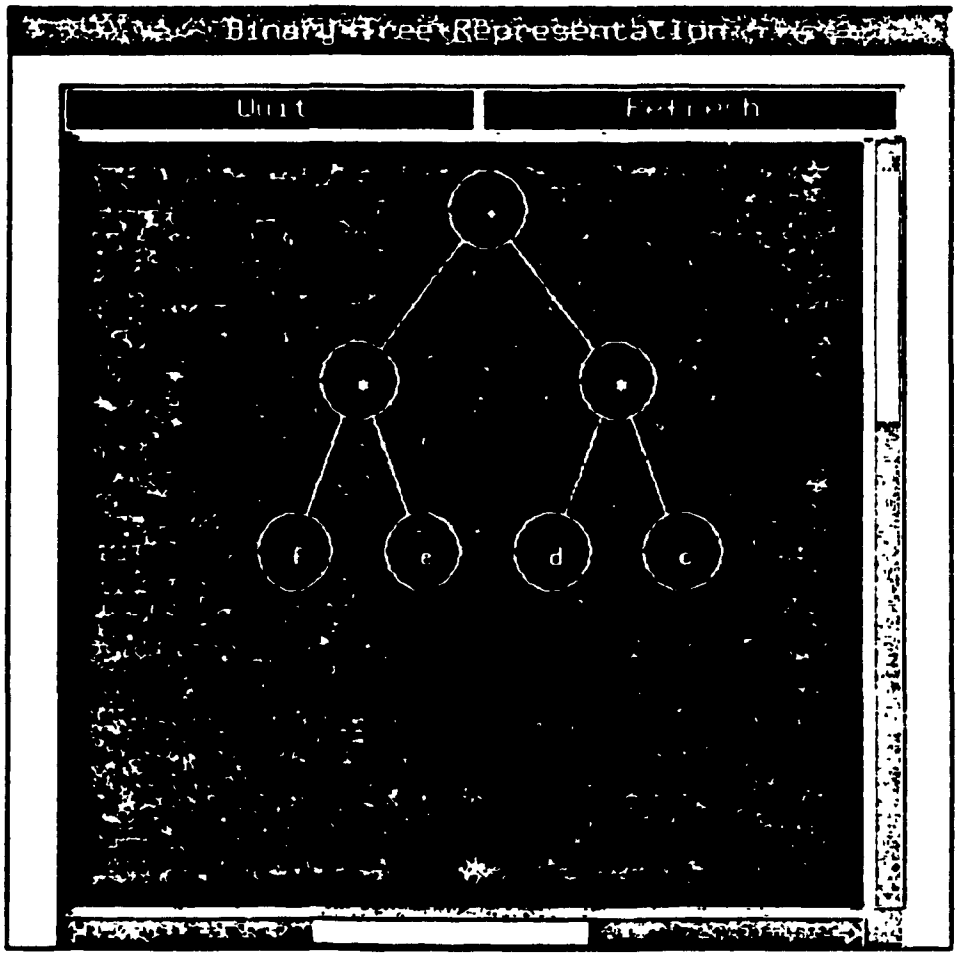


Figure 7.7 **The binary tree representation of equation5. Binary trees are shown in pop-up WorkSpace widgets.**

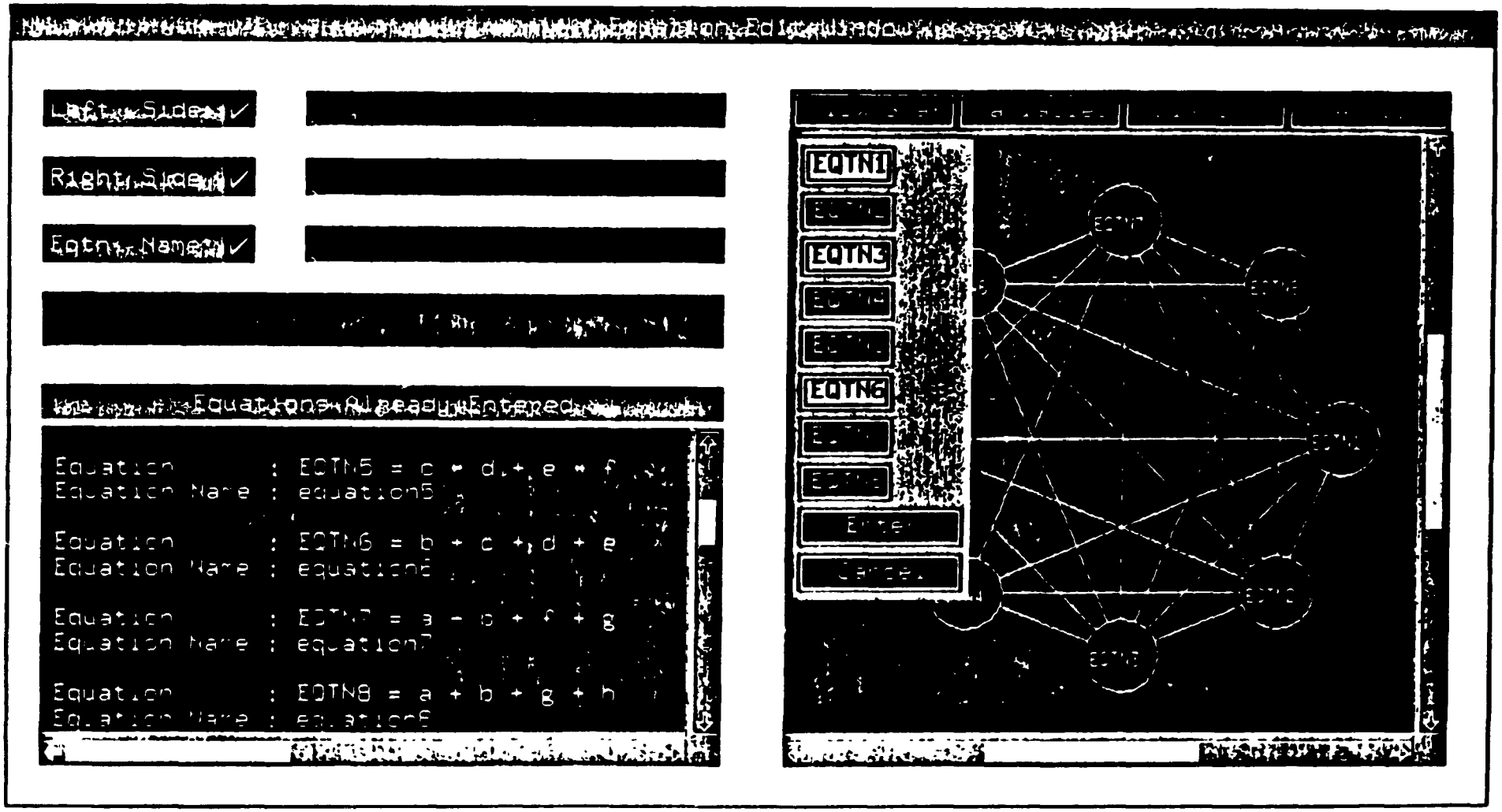


Figure 7.8 Forming a subset of equations. Equation subsets are useful in reducing 'link crowding' in large interrelated equation sets.

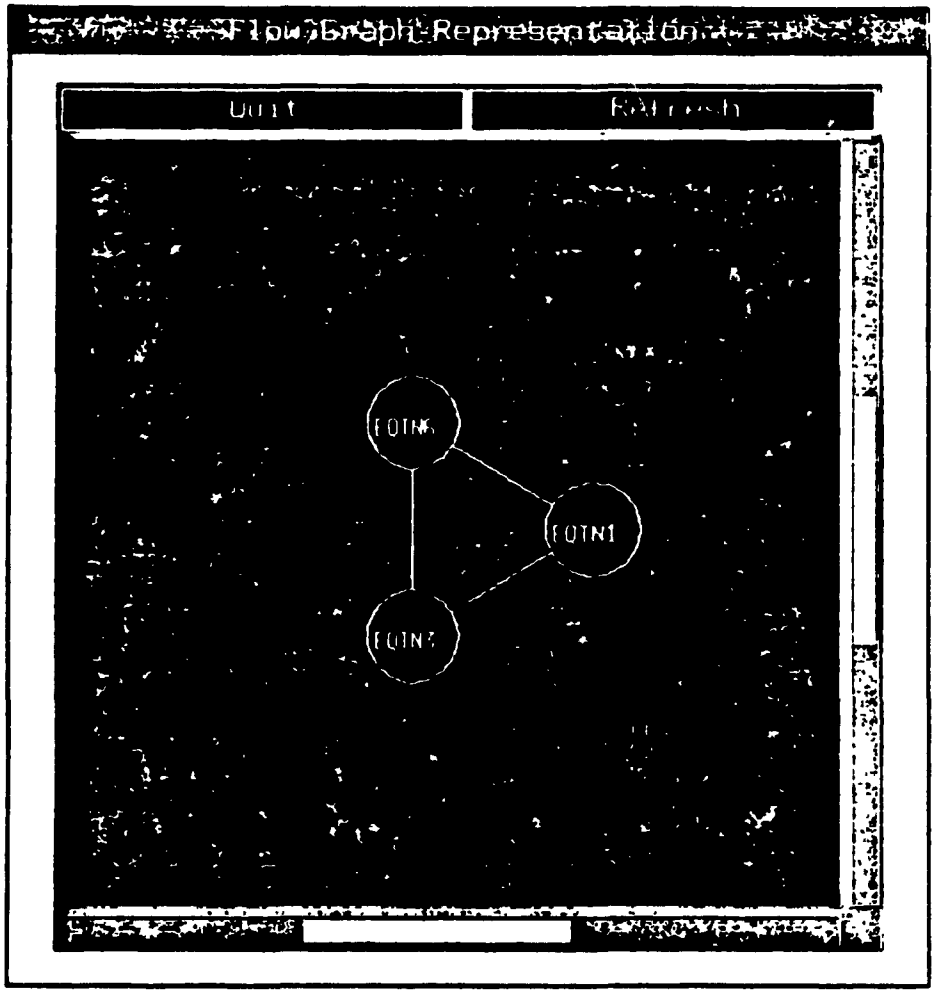


Figure 7.9 The flow graph representation of the equation subset formed in figure 7.8. Flow graphs of equation subsets are displayed in pop-up WorkSpace widgets.

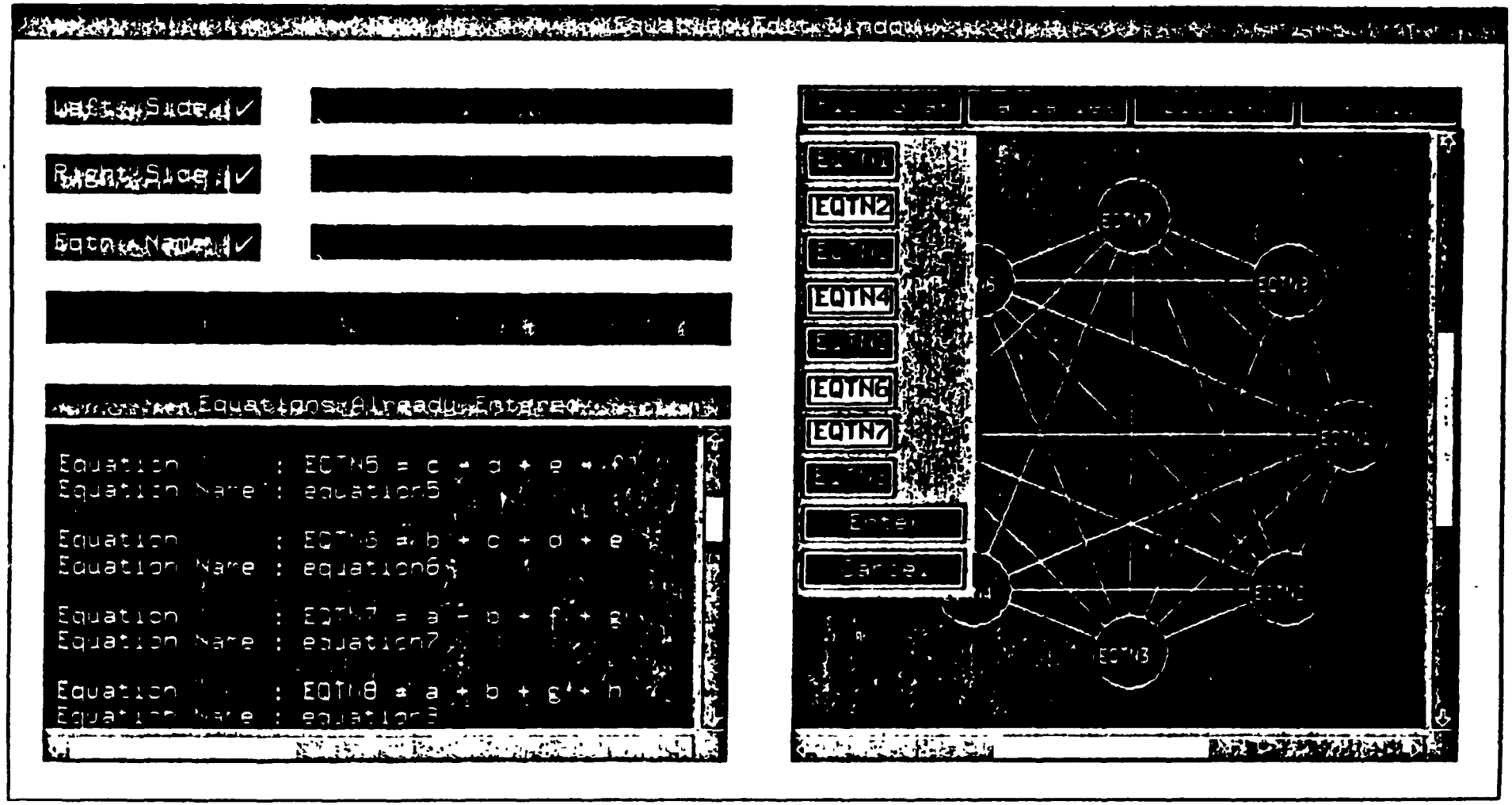


Figure 7.10 Creating another subset of equations.

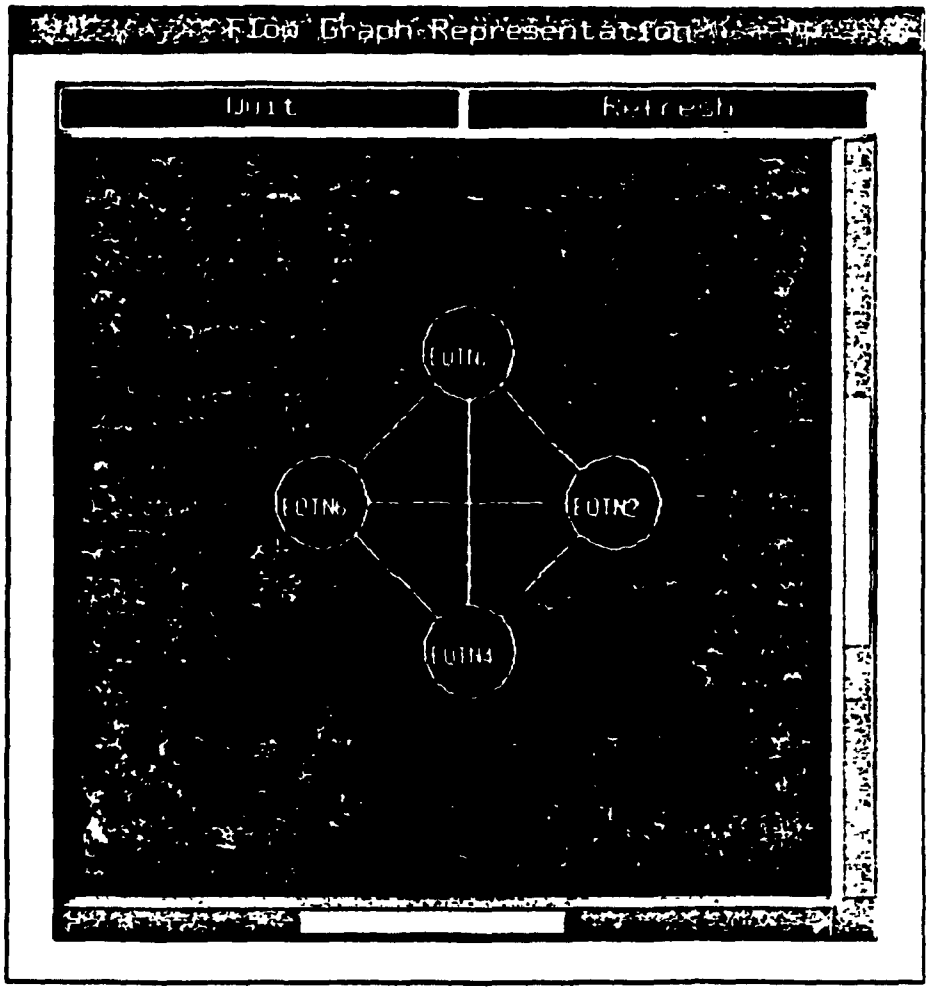


Figure 7.11 The flow graph representation of the equation subset formed in figure 7.10.

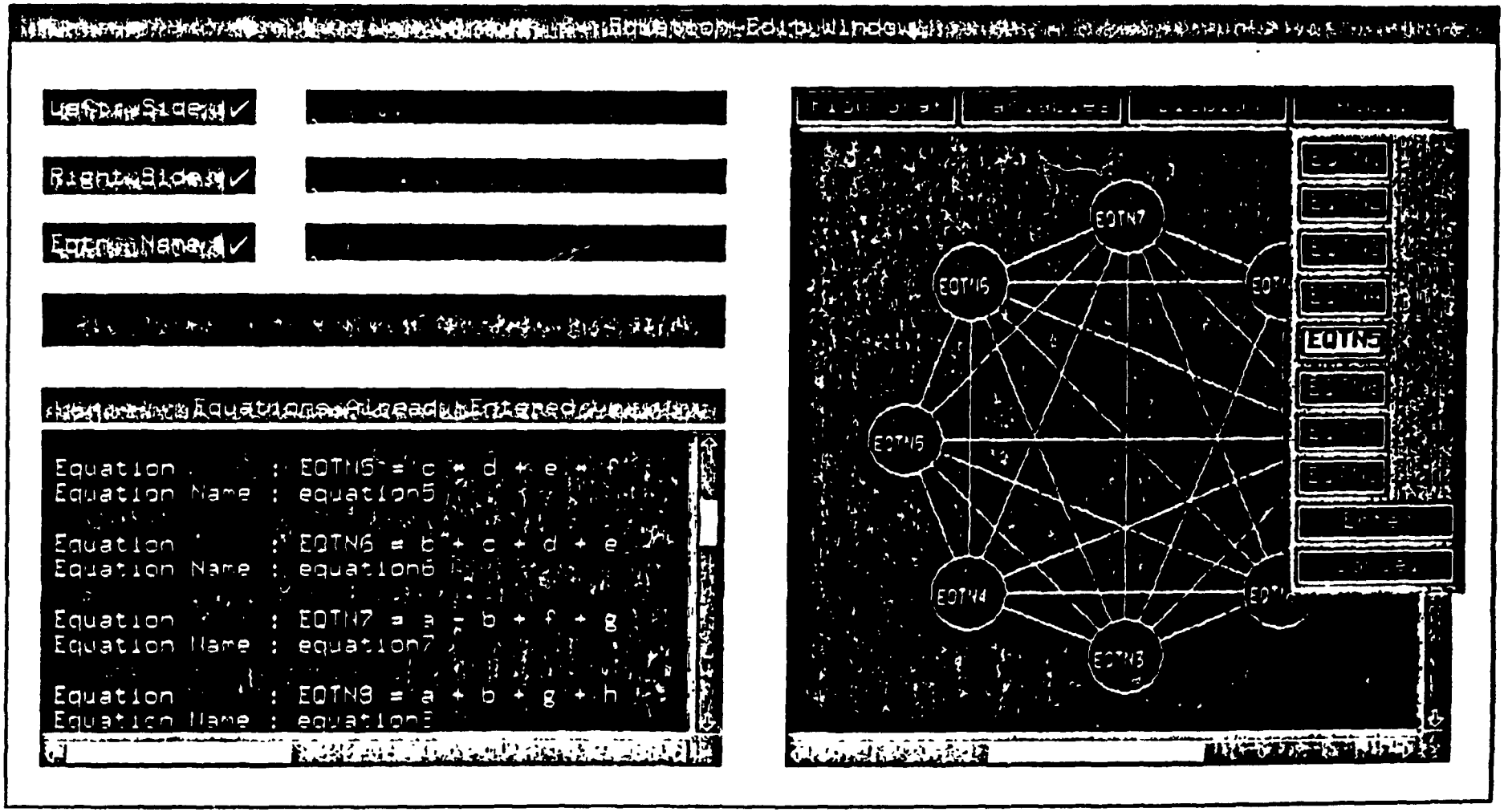


Figure 7.12 Selecting equations to be deleted from the current set of equations.

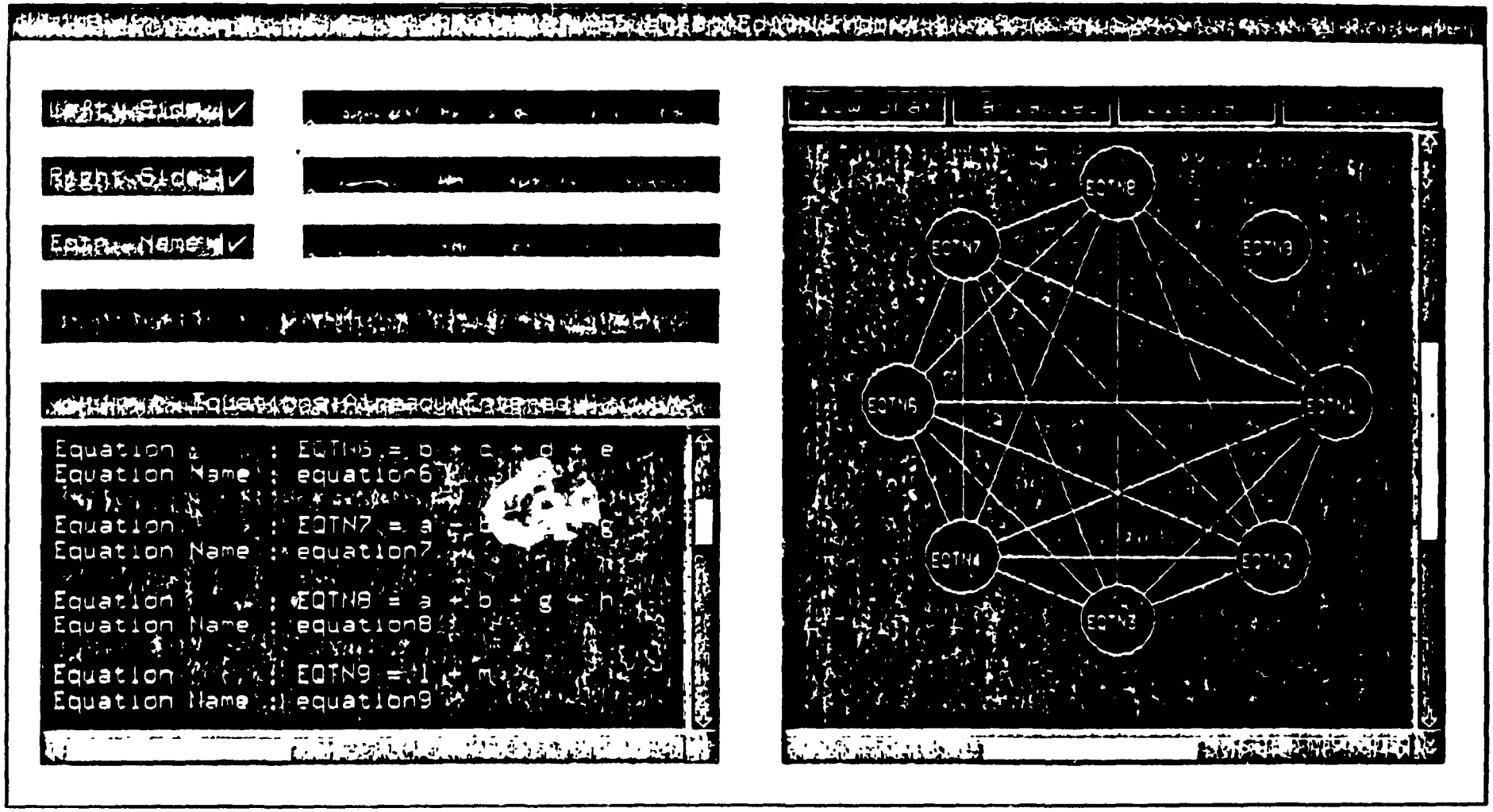


Figure 7.13 **Equation5 was removed from the current equation set and replaced by equation9. Equations may be added to, or deleted from any set of equations.**

The Lisp function **initializeXt** establishes a connection with the X server. The first argument to **initializeXt** is the name of the **Shell** widget. Arguments that affect command line parsing may also be specified following the widget name.

By 'loading' the previous Lisp statements at the Lisp-interperter prompt, a **StaticText** widget is created at location (100, 200), having a size of 120x150 pixels, and a red border color. To application programmers, this example implies that:

- X Toolkit programming details are reduced to a bare minimum.
- Lisp code interpeted at the Lisp-prompt produces graphics instantaneously on the screen. This is a great advantage since X-Toolkit C programs had to be compiled before they could be run.
- It takes only a few lines of Lisp code to create a sophisticated looking user interface.

A disadvantage of Lisp is that the newly created layer sitting on top of the existing Xt Intrinsics and the X Widget Set layers, leads to slow-performing applications. However, performance may be improved by compiling Lisp programs.

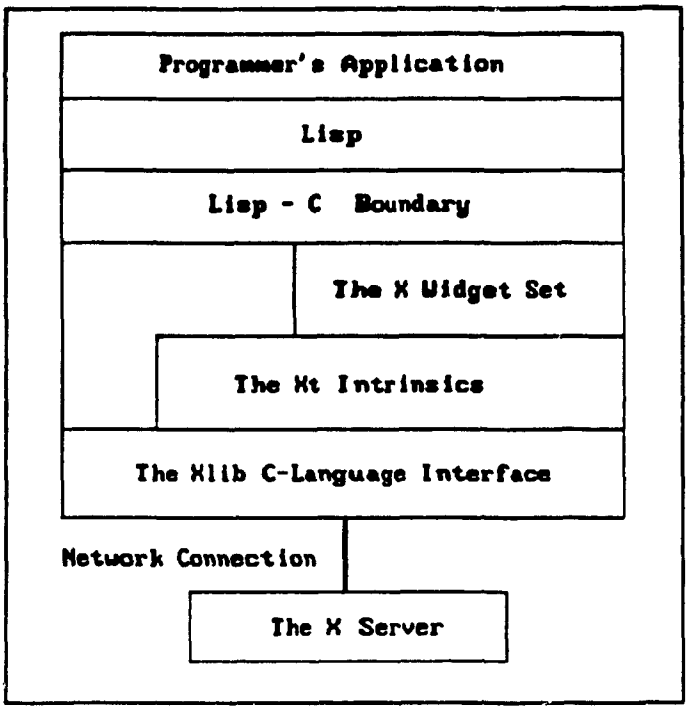


Figure 7.14 A conceptual view of the layers leading to Lisp.

7.2.1 Rules and Methods

It is important that application programmers be provided with *rules* that control user interface components such as menus, buttons, and scrollbars. A rule is simply a condition imposed on an interface component. As an example, consider the pull-down menu shown in figure 7.15. The second menu entry in the pull-down menu is disabled when all other menu entries are active.

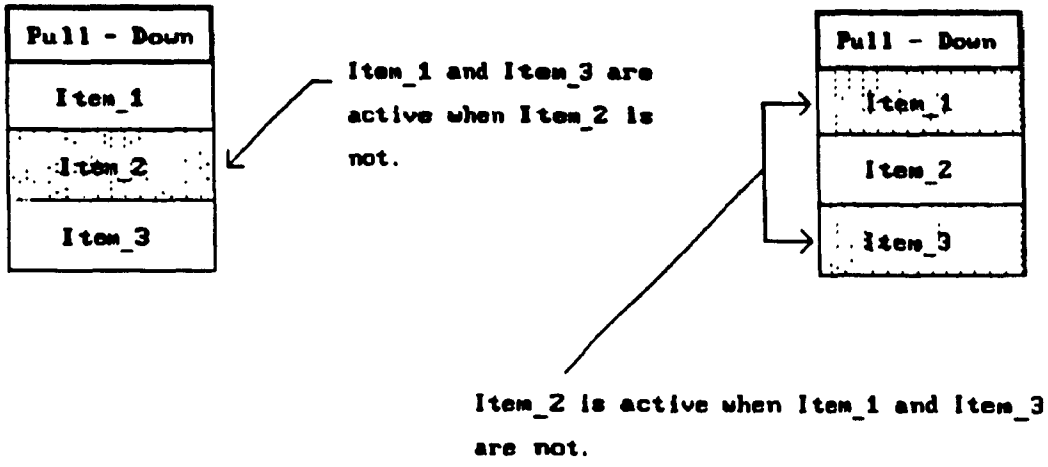


Figure 7.15 Rules governing menu entries.

In programming terms, the second entry is created and rule-governed as follows:

```
.
.
(createMenuButtonWidget RowCol "Pull-Down" "item 2"
 '(ActiveOthersNot))
.
.
```

The rule is outlined at the end of the Lisp function. It simply states that **item_2** must be active when **item_1**, and **item_3** are not, and vice-versa. Therefore, the activation status of menu entries is completely controlled by rules that application programmers can specify.

User interface component rules can also be combined using logical operations as outlined in the following examples:

(rule1 & rule2 & rule3 ...)

(rule1 | rule2 | rule3 ...)

(~rule1 & rule2 & rule3 ...)

(rule1 | ~rule2 | rule3 ...)

The '&' operator performs an 'anding' operation on a set of user interface component rules, while the '|' operator 'ors' a set of rules inclusively. The '~' operator is used for rule negation.

7.2.2 Expanding the EDS UI

The EDS user interface supports one level of hypertext only. A multi-level hypertext mechanism for querying equation nodes and links is necessary as EDS is interfaced to a package that uses interval mathematics to impose bounds on variables [9][30]. This allows the user to move between different levels of equation information ranging from variable names to bounds imposed on each of those variables.

The EDS user interface needs to be expanded so as to access a package for tool integration. The package uses a black box architecture [31] to gather information about different design tools (EDS, finite-element-based field simulation tools [32]), and to trigger those tools based on a schedule determined from the information gathered.

Appendix A

Widget Classes

This appendix briefly lists the X widget class tree, and serves as a quick reference to the Xt Intrinsics and the X Widget Set widget classes.

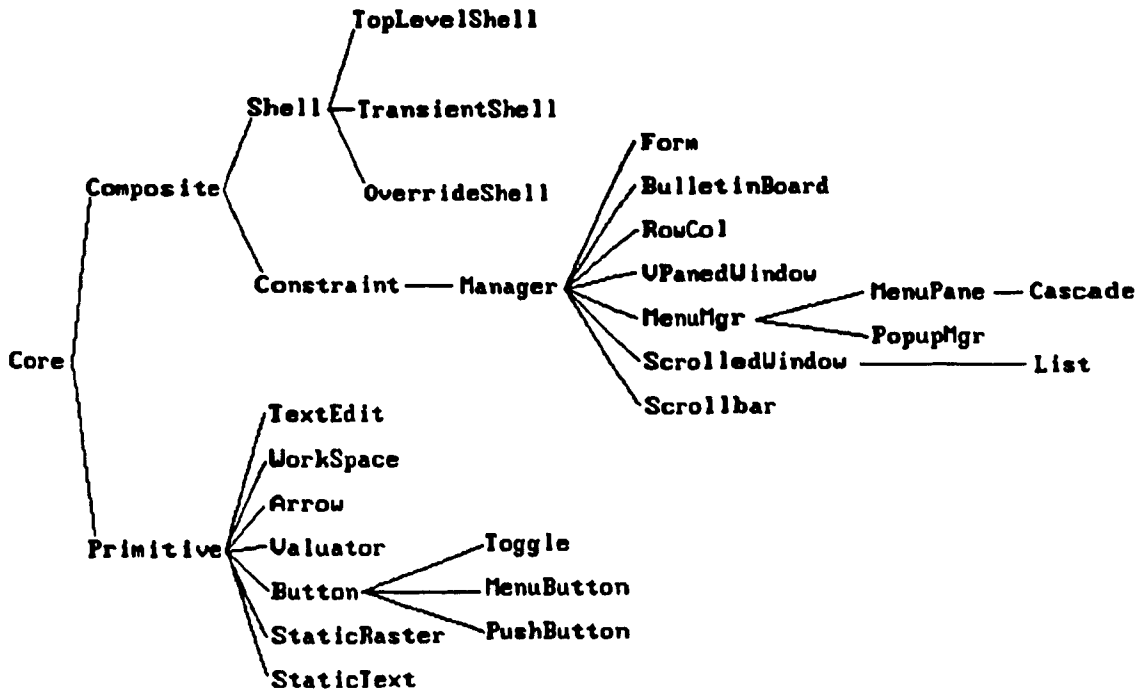


Figure A.1 The X Widget class tree [1].

The Core Intrinsics Widget Classes

COMPOSITE

Class:	**compositeWidgetClass**
Class Name:	**Composite**
Superclasses:	**Core**

The **Composite** widget class is a meta-class used as a container of other widgets.

CONSTRAINT

Class:	**constraintWidgetClass**
Class Name:	**Constraint**
Superclasses:	**Core, Composite**

The **Constraint** widget is a meta-class defined by the Xt Intrinsics. It attaches additional resources to its children, and uses these constraints to manage the geometry of its children.

CORE

Class:	**WidgetClass**
Class Name:	**Core**
Superclasses:	None
Callback List:	**XtNdestroy**

The **Core** widget class is an Xt Intrinsics widget class. It is never instantiated as a widget, and its sole purpose is simply a supporting super class to other widget classes. It provides resources required by all widgets.

SHELL

Class:	**shellWidgetClass**
Class Name:	**Shell**
Superclasses:	**Core, Composite**

The **Shell** widget class is defined by the Xt Intrinsics. It provides an interface between applications and the window manager.

<u>The X Widget Set Classes</u>

ARROW

Class:	**XwarrowWidgetClass**
Class Name:	**Arrow**
Superclasses:	**Core, Primitive**
Callback List:	**XtNselect, XtNrelease**
	XtNenter, XtNleave
	XtNselect, XtNunselect

The **Arrow** widget supports drawing of an arrow within the bounds of its window. The arrow can be drawn in the up, down, left, and right directions.

BULLETINBOARD

Class:	**XwBulletinWidgetClass**
Class Name:	**BulletinBoard**
Superclasses:	**Core, Composite, Constraint, Manager**

The **BulletinBoard** widget is a composite widget that does not enforce any ordering on its children. Applications must specify the location of **BulletinBoard** widget children.

BUTTON

Class:	**XwbuttonWidgetClass**
Class Name:	**Button**
Superclasses:	**Core, Primitive**

The **Button** widget is an X Widget meta-class. It is never instantiated as a widget, and provides a set of resources needed by other widgets (**XwtoggleWidgetClass** and **XwpushButtonWidgetClass**).

CASCADE

Class:	**XwcascadeWidgetClass**
Class Name:	**Cascade**
Superclasses:	**MenuPane**
Callback List:	**XtNselect, XtNleave XtNvisible, XtNunmap**

The **Cascade** widget is a composite widget that application programmers use for creating menus. It always displays its children in a single column, and attempts to resize itself to the smallest possible size.

FORM

Class:	**XwformWidgetClass**
Class Name:	**Form**
Superclasses:	**Core, Composite, Constraint, Manager**

The **Form** widget is a constraint widget based manager that establishes spatial relationships between its children.

LIST

Class:	**XwlistWidgetClass**
Class Name:	**List**

Superclasses:	**Core, Composite, Constraint, Manager, ScrolledWindow**
Callback List:	**XtNselect, XtNdoubleClick**

The **List** widget allows a two-dimensional set of widgets to be displayed in a row/column fashion. It provides management and layout functions for its elements.

MANAGER

Class:	**XwmanagerWidgetClass**
Class Name:	**Manager**
Superclasses:	**Core, Composite, Constraint**

The **Manager** class is an X Widget meta class. It is never instantiated as a widget. It is mainly used as a supporting superclass for other widget classes.

MENU

Class:	**XwmenumgrWidgetClass**
Class Name:	**MenuMgr**
Superclasses:	**Core, Composite, Constraint, Manager**

The **Menu Manager** class is an X Widget meta class. It is never instantiated as a widget. Its main purpose is to serve as a supporting superclass for other menu manager classes.

MENUBUTTON

Class:	**XwmenubuttonWidgetClass**
Class Name:	**MenuButton**
Superclasses:	**Core, Primitive, Button**
Callback List:	**XtNselect, XtNcascadeSelect, XtNcascadeUnselect**

The **Menu Button** widget is commonly used with **Menu Pane** and **Menu Manager** widgets to build menus.

MENUPANE

Class:	**XwmenupaneWidgetClass**
Class Name:	**MenuPane**
Superclasses:	**Core, Composite, Constraint, Manager, MenuMgr**

The **Menu Pane** class is an X Widget meta class. It is never instantiated as a widget, and its main purpose is to serve as a supporting superclass for other **Menu Pane** widget classes.

POPUP MENU MANAGER

Class:	**XwpopupmgrWidgetClass**
Class Name:	**PopupMgr**
Superclasses:	**Core, Composite, Constraint, Manager, MenuMgr**

The **Popup Menu Manager** widget is a composite widget that manages a collection of **Menu Pane** widgets.

PRIMITIVE

Class:	**XwprimitiveWidgetClass**
Class Name:	**Primitive**
Superclasses:	**Core**

The **Primitive** class is an X Widget metaclass. It is never instantiated as a widget, and it is mainly used as a supporting class for other widget classes.

PUSH BUTTON

Class:	**XwpushButtonWidgetClass**
Class Name:	**PushButton**
Superclasses:	**Core, Primitive, Button**
Callback List:	**XtNselect, XtNrelease**

The **Push Button** widget consists of a text label surrounded by a button border. By default, the interior of the button is inverted when the button is in the down state. The interior of the button is reinverted when the button is released.

ROWCOL

Class:	**XwrowColWidgetClass**
Class Name:	**RowCol**
Superclasses:	**Core, Composite, Constraint, Manager**

A **Row Column** widget arranges its children into rows and columns.

SCROLL BAR

Class:	**XwscrollbarWidgetClass**
Class Name:	**ScrollBar**
Superclasses:	**Core, Composite, Constraint, Manager**
Callback List:	**XtNareaSelected, XtNsliderMoved** XtNgranularity

The **Scrollbar** widget combines the **Valuator and Arrow** widgets to form a horizontal or a vertical scrollbar.

SCROLLED WINDOW

Class:	**XwswindowWidgetClass**
Class Name:	**ScrolledWindow**
Superclasses:	**Core, Composite, Constraint, Manager, ScrolledWindow**
Callback List:	**XtNvScrollEvent, XtNhScrollEvent**

The **Scrolled Window** widget combines the **Scrollbar and Bulletin Board** widgets to implement a visible window onto a larger data display.

STATIC RASTER

Class:	**XwstaticrasterWidgetClass**
Class Name:	**StaticRaster**
Superclasses:	**Core, Primitive**

STATIC TEXT

Class:	**XwstatictextWidgetClass**
Class Name:	**StaticText**
Superclasses:	**Core, Primitive**
Callback List:	**XtNselect, XtNrelease**

The **Static Raster** widget displays an uneditable raster image. By default, the image is placed in a window that has the exact size of the raster.

TEXT EDIT

Class:	**XwtexteditWidgetClass**
Class Name:	**TextEdit**
Superclasses:	**Core, Primitive**
Callback List:	**XtNmotionVerification, XtNmodifyVerification, XtNleaveVerification, XtNexecute**

The **Text Edit** widget provides a mutli-line text editor which has a customizable user interface.

VALUATOR

Class:	**XwvaluatorWidgetClass**
Class Name:	**Valuator**
Superclasses:	**Core, Primitive**

The **Valuator** widget implements a horizontal or vertical scrolling widget as a rectangular bar containing a sliding box.

VERTICAL PANED WINDOW

Class:	**XwvPanedWidgetClass**
Class Name:	**VPanedWindow**
Superclasses:	**Core, Composite, Constraint, Manager**

The **Vertical Paned Manager** is a composite widget which lays out its children in a vertically tiled format.

WORKSPACE

Class:	**XwworkspaceWidgetClass**
Class Name:	**WorkSpace**
Superclasses:	**Core, Primitive**
Callback List:	**XtNexpose**
	XtNresize
	XtNkeyDown

The **WorkSpace** widget provides the application programmer with an empty primitive widget, that can be used for drawing graphics.

Appendix B

X Event Masks

The following is a listing of X Event masks and the associated event types [2]:

Event Mask	Event Type
ButtonMotionMask	MotionNotify
Button1MotionMask	MotionNotify
Button2MotionMask	MotionNotify
Button3MotionMask	MotionNotify
Button4MotionMask	MotionNotify
Button5MotionMask	MotionNotify
ButtonPressMask	ButtonPress
ButtonReleaseMask	ButtonRelease
ColormapChangeMask	ColormapNotify
EnterWindowMask	EnterNotify
LeaveWindowMask	LeaveNotify
ExposureMask	Expose
GCGraphicsExposures	GraphicsExpose
	NoExpose
FocusChangeMask	FocusIn
	FocusOut
KeymapStateMask	KeymapNotify
KeyPressMask	KeyPress

KeyReleaseMask	KeyRelease
OwnerGrabButtonMask	Not Applicable
PointerMotionMask	MotionNotify
PointerMotionHintMask	Not Applicable
PropretyChangeMask	PropretyNotify
ResizeRedirectMask	ResizeRequest
StructureNotifyMask	CirculateNotify
	ConfigureNotify
	DestroyNotify
	GravityNotify
	MapNotify
	ReparentNotify
	UnmapNotify
SubstructureNotifyMask	CirculateNotify
	ConfigureNotify
	CreateNotify
	DestroyNotify
	GravityNotify
	MapNotify
	ReparentNotify
	UnmapNotify
SubstructureRedirectMask	CirculateRequest
	ConfigureRequest
	MapRequest

Not Applicable	ClientMessage
Not Applicable	MappingNotify
Not Applicable	SelectionNotify
Not Applicable	SelectionClear
Not Applicable	SelectionRequest
VisibilityChangeMask	VisibilityNotify

Appendix C

Stack Procedures and Line Segment Algorithms

The following procedures are for initializing a stack, pushing elements onto a stack, and popping elements from a stack:

```
type link = ^node;
   node = record key: integer; next : link end;
var head, z : link;
procedure stackinit;
 begin
 new(head); new(z);
 head^.next := z; z^.next := z
 end;
procedure push(v : integer);
 var t : link;
 begin
 new(t);
 t^.key := v; t^.next := head^.next;
 head^.next := t
 end
function pop : integer;
 var t : link;
 begin
 t := head^.next;
 pop := t^.key;
 head^.next := t^.next;
 dispose(t)
 end;
function stackempty : boolean;
 begin stackempty := (head^.next = z) end;
```

C.1 Line Segment Intersection

Given two line segments, a straight forward way to determine if they intersect consists of finding the intersection point of the lines defined by the segments, and then checking

whether this intersection point falls between the endpoints of both segments. An easier method is based on the following. Given three points A, B, and C, a check is made to determine if A, B, and C are stored in clockwise or counterclockwise direction (Assuming that we travel from A to B to C). The procedure outlined below checks for this property.

```
function ccw(p0,p1,p2):integer;
 var dx1,dx2,dy1,dy2:integer
 begin
 dx1:=p1.x-p0.x; dy1:=p1.y-p0.y;
 dx2:=p2.x-p0.x; dy2:=p2.y-p0.y;
 if dx1*dy2>dy1*dx2 then ccw:=1;
 if dx1*dy2<dy1*dx2 then ccw:=-1;
 if dx1*dy2=dy1*dx2 then
   begin
   if(dx1*dx2<0) or (dy1*dy2<0) then ccw:=-1 else
   if(dx1*dx1+dy1*dy1)>=(dx2*dx2+dy2*dy2) then ccw:=0 else ccw:=-1;
   end;
 end;
```

First, suppose that the quantities *dx1*, *dx2*, *dy1*, *dy2* are positive. Then, the slope of the line connecting *p0* to *p1* is *dy1/dx1*, and the slope connecting *p0* and *p2* is *dy2/dx2*. If the slope of the second line is greater than that of the first, a counterclockwise turn is required to go from *p0* to *p1* to *p2*. If the slope is less, a clockwise turn is required. However, if the three points align, the following rules are used to set the value of *ccw*:

* *ccw* = 1 if *p1* is between *p0* and *p2*.

* *ccw* = 0 if *p2* is between *p0* and *p1*.

* *ccw* = -1 if *p0* is between *p1* and *p2*.

This immediately suggests a solution to the two-segment intersection problem. If both endpoints of each line segment are on different sides (different *ccw* values) of the other, then the line segments must intersect [10]:

```
function intersect(l1,l2:line):boolean;
 begin
 intersect:=((ccw(l1.p1,l1.p2,l2.p1)*
        ccw(l1.p1,l1.p2,l2.p2))<=0) and
        ((ccw(l2.p1,l2.p2,l1.p1)*
        ccw(l2.p1,l2.p2,l1.p2))<=0);
 end;
```

References

[1] Young D. A., "X Window Systems Programming and Applications with Xt", Prentice-Hall, Inc., 1989.

[2] Nye A., "The Xlib Programming Manual", O'Reilly & Associates, Inc., 1988.

[3] Nye A., "The Xlib Reference Manual", O'Reilly and Associates, Inc., 1988.

[4] O'Reilly T., "The Toolkits (and Politics) of X Windows", Unix World, vol.6, no.2, pp. 66-73, February 1989.

[5] McCormack J., Asente P., "Using the X Toolkit or How to Write a Widget", Proceedings of the Summer, 1988 USENIX Conference, pp. 1-13.

[6] McCormack J., Asente P., "An Overview of the X Toolkit", Proceedings of the ACM SIGGRAPH Symposium on User Interface Software, pp. 46-55, October 1988.

[7] Rosenthal D. S., "Going for Baroque", UNIX Review, vol 6, no. 6, pp. 71-79.

[8] Rosenthal D. S., "A Simple X11 Client Program, or, How Hard can it Really Be to Write 'Hello World' ?", Proceedings of the Winter, 1988 USENIX Conference, pp. 229-235.

[9] Saldanha C., "Electromagnetic Design System", Master's Thesis, McGill University, 1987, pp. 1-31, 78-81.

[10] Sedgewick R., "Algorithms", 2nd edition, Addison-Wesley Publishing Company, 1988.

[11] Tenenbaum A., Augenstein M., "Data Structures Using Pascal", 2nd edition, Prentice-Hall, 1984.

[12] Schiefler R., Gettys J., "The X Window System", ACM Transactions on Graphics, vol.5, no. 2, pp. 79-109, April 1986.

[13] Schiefler R., Gettys J., Newman R., "X Window System", DEC Press, 1988.

[14] Shneiderman B., "Designing the User Interface", Addison-Wesley Publishing Company, 1987.

[15] Smith W., "Using Computer Color Effectively", Prentice-Hall, 1989.

[16] Mayer B., "Object-Oriented Software Construction", Prentice-Hall, 1988.

[17] Foley J. D., Van Dam A., "Fundamentals of Interactive Computer Graphics", Addison-Wesley Publishing Company, 1983.

[18] Kernighan B. W., Ritchie D. M., "The C Programming Language", Prentice-Hall, 1978.

[19] Rochkind M., "Advanced Unix Programming", Prentice-Hall, 1985.

[20] Schildt H., "C: the Complete Reference", McGraw-Hill, 1987.

[21] Anderson P., Anderson G., " Advanced C: Tips and Techniques", Howard Sams Publishing Company, 1988.

[22] Knuth D. E., "The Art of Computer Programming. Volume 3: Sorting and Searching", second printing, Addison-Wesley, Reading, MA, 1975.

[23] Knuth D. E., "The Art of Computer Programming. Volume 1: Fundemental Algorithms,", second edition, Addison-Wesley, Reading, MA, 1973.

[24] OSF/Motif User's Guide, Prentice-Hall, Englewood Cliffs, NJ, 1990

[25] Holstein, D., 'Decision Tables: A Technique for Minimizing Routine, Repetitive Design', Machine Design, August 1962, pp. 76-79.

[26] Davis, E., 'Constraint Propagation with Interval Labels', Artificial Intelligence, volume 32, 1987, pp. 281-331.

[27] Preiss, K., 'Data Frame Model for the Engineering Design Process', Design Studies, volume 1, no. 5, April 1980.

[28] 'Knowledge Craft Reference Manual', Carnegie Group Inc, Pittsburgh, Pennsylvania, 1986.

[29] Steele Jr., G. L., 'Common Lisp: The Language', Digital Press, Hanover, Massachussetts, 1984.

[30] Brett, C., 'An Interval Mathematics Package For Computer-Aided Design In Electromagnetics', Master's Thesis, McGill University, 1990.

[31] Sassine, R., Lowther D. A., 'Integrating Computer Based Electromagnetic Device Design Tools To Solve Coupled Problems', To appear in the 8th Conference on the Computation of Electromagnetic Fields, July 7-11, 1991 Sorrento-Italy.

[32] Lowther, D. A., and Silvester, P. P., 'Computer-Aided Design in Magnetics', Springer-Verlag, New York, 1985.