AD NUMBER

ADB152500

LIMITATION CHANGES

TO:

Approved for public release; distribution is unlimited.

FROM:

**Distribution authorized to U.S. Gov't. agencies and their contractors;
Administrative/Operational Use; FEB 1991. Other requests shall be referred to Air Force Electronic Systems Division, Hanscom AFB, MA 01731.**

AUTHORITY

UNISYS Ltr via ESD dtd 26 Feb 1991

(2)

TASK: UR2
CDRL: 0108
02/15/9

UR20 — Process Environment Integration

Reusable Graphical Browser Version 0.3.2

UNISYS

STARS-RC-01080/002/00

15 February 1991

TASK: UR20
CDRL: 01080
15 February 1991

USER'S MANUAL

For The

SOFTWARE TECHNOLOGY FOR ADAPTABLE, RELIABLE SYSTEMS (STARS)

Reusable Graphical Browser
Version 0.3.2

STARS-RC-01080/002/00
Publication No. GR-7670-1208(NP)
15 February 1991

Data Type: A005, Informal Technical Data

CONTRACT NO. F19628-88-D-0031
Delivery Order 0002

Prepared for:

Electronic Systems Division
Air Force Systems Command, USAF
Hanscom AFB, MA 01731-5000

Prepared by:

Unisys Defense Systems
Tactical Systems Division
12010 Sunrise Valley Drive
Reston, VA 22091

TASK: UR20
CDRL: 01080
15 February 1991

USER'S MANUAL

For The

SOFTWARE TECHNOLOGY FOR ADAPTABLE, RELIABLE SYSTEMS (STARS)

Reusable Graphical Browser
Version 0.3.2

STARS-RC-01080/002/00
Publication No. GR-7670-1208(NP)
15 February 1991

Data Type: A005, Informal Technical Data

CONTRACT NO. F19628-88-D-0031
Delivery Order 0002

Prepared for:

Electronic Systems Division
Air Force Systems Command, USAF
Hanscom AFB, MA 01731-5000

Prepared by:

Unisys Defense Systems
Tactical Systems Division
12010 Sunrise Valley Drive
Reston, VA 22091

PREFACE

This document was produced by TRW in support of the Unisys STARS Prime contract under the Process/Environment Integration task (UR20). This CDRL, 01080, Volume II is type A005 (Informal Technical Data) and is entitled "Reusable Graphical Browser User's Manual, Version 0.3.2".

Task Manager: Dr. Thomas E. Shields

Reviewed by: Thomas E. Shields
for Teri F. Payton, System Architect

Approved by: Hans W. Polzer
Hans W. Polzer, Program Manager

Contents

List of Figures

1 Introduction

1.1 Purpose

This document is the User's Manual for the Reusable Graphical Browser developed by TRW, under contract to UNISYS, as a subtask of the STARS User Interface Task (UR20). The Reusable Graphical Browser (also referred to as the Reusable Browser) is a reusable software component designed to facilitate the construction of graphical tools for browsing over the contents of various object management systems. More specifically, it is intended to serve as a basis for constructing such tools quickly and easily using the Ada programming language. The purpose of this Manual is to provide guidance to tool builders using the Reusable Graphical Browser to construct specific graphical browser tools.

1.2 Scope

The User's Manual presents both a conceptual description of the Reusable Graphical Browser and detailed instructions for its reuse. Among other information, it contains complete technical specifications for the application (browser tool) interface provided by the Reusable Graphical Browser and a general description of the user (man-machine) interface implemented by the Reusable Graphical Browser. It does not, however, contain instructions regarding the use of tools constructed from the Reusable Graphical Browser. Such instructions are necessarily application-specific.

1.3 Organization

The main body of this manual consists of three sections. Section 1 is this introduction. Section 2 describes the Reusable Graphical Browser at the conceptual level. Section 3 presents detailed instructions for reuse, and illustrates those instructions with an example.

At the end of the manual are four appendices for quick reference. Appendix A presents Ada package specifications for the application interface. Appendix B describes the user interface, and discusses ways in which it may be tailored for a specific application. Appendix C lists current limitations of the Reusable Graphical Browser. Appendix D defines acronyms used in this manual.

2 Overview of The Reusable Browser

The Reusable Graphical Browser is a reusable software component designed to facilitate the construction of graphical tools for browsing over the contents of various object management systems. It is not, in and of itself, a browser tool. Rather, it is intended to serve as a foundation for constructing such tools using the Ada programming language.

In that capacity, the Reusable Graphical Browser accomplishes the following:

- it reduces development costs for graphical browser tools, by providing a graphical user interface that is easily adaptable to a wide variety of browsing applications;
- it improves maintainability and portability of such tools, by insulating them from the underlying graphics systems;
- it reduces user training costs, by promoting a common "look and feel" across all such tools.

2.1 Architecture

Figure 1 illustrates how the Reusable Graphical Browser fits into the overall architecture of a graphical browser tool. The application, a specific graphical browser tool, relies on services provided by the Reusable Graphical Browser for interaction with the user (i.e., the human). The Reusable Graphical Browser, in turn, relies on services provided by an underlying graphics system (in the current implementation, the X window system) for terminal input and output. In addition, the application relies on services provided by an object management system (OMS) to query and/or manipulate application-specific objects. Note that the Reusable Graphical Browser does not interact directly with the OMS; rather, the application must actively transfer information between the two.

The interface between the application and the Reusable Graphical Browser is termed the "Application Interface". It provides a much higher level of abstraction than the X window system interface. Furthermore, it hides virtually all X window system dependencies from the application. As a result of this high degree of abstraction and information hiding, the Reusable Graphical Browser may be ported to another (comparable) window system with minimal impact on application code. Similarly, migration to later releases of the X window system should also be possible with little or no impact on application code.

The Application Interface is generic with respect to the data types used to represent the objects within the OMS and the relationships among them. Recall that the Reusable Graphical Browser does not interact directly with the OMS, and therefore has no immediate knowledge of the OMS schema. Consequently, it must rely on the application to provide any and all information concerning the structure of data within the OMS. For the most part, the Reusable Graphical Browser doesn't need much of this type of information in order to function. Minimally, it needs enough to be able to identify individual objects and relationships (and also collections of objects and relationships, if the application deals with multiple collections). It may also need some information to distinguish between different kinds of objects and/or relationships, if the application wishes them to be treated differently (e.g., displayed differently). And finally, it may need a way to retrieve the attributes of an object or a relationship, if the application wishes them to be automatically integrated into the screen layout. The application is, therefore, expected to provide the following parameters when instantiating the Application Interface:

- an arbitrary data type for values that uniquely identify objects;

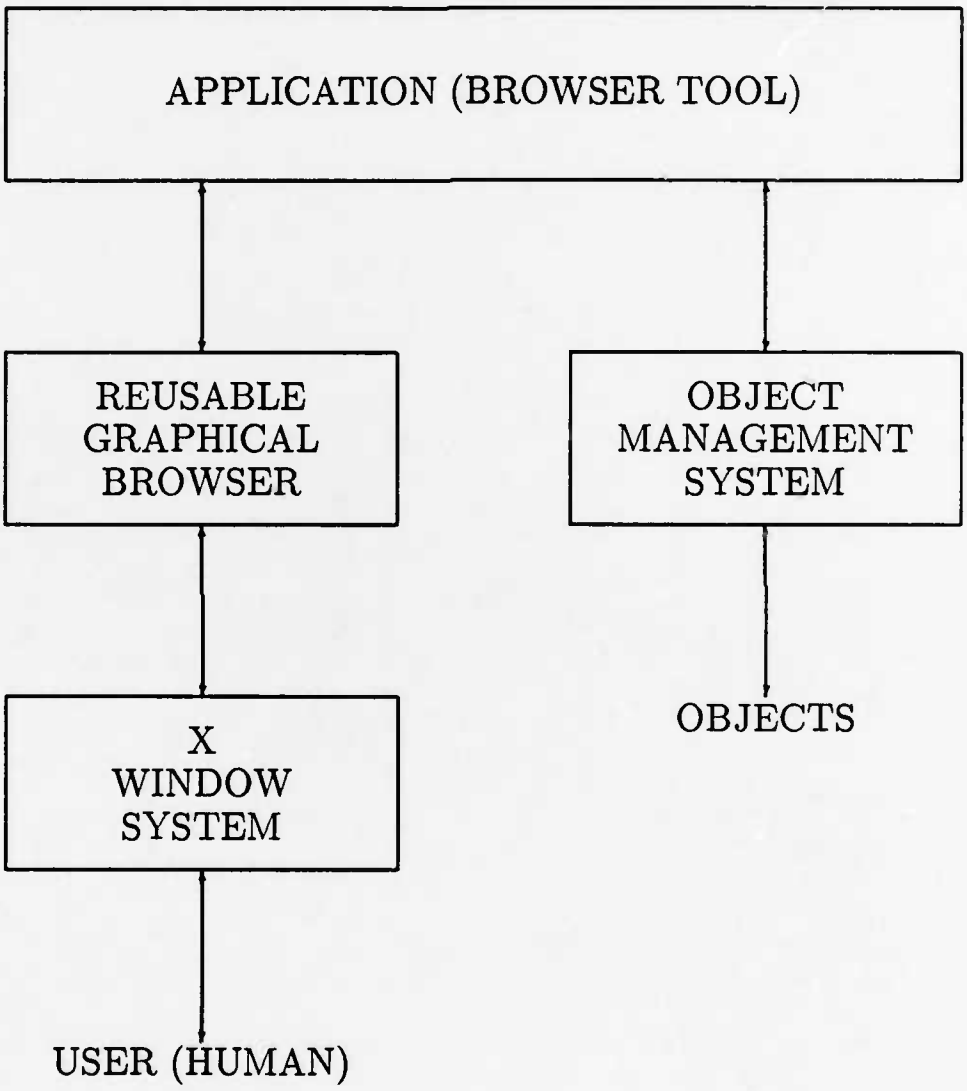

Figure 1: Browser Tool Architecture

- an arbitrary data type for values that uniquely identify relationships;
- an arbitrary data type for values that uniquely identify collections of objects and relationships;
- a scalar data type for values that distinguish different kinds of objects;
- a scalar data type for values that distinguish different kinds of relationships;
- a function for retrieving the attributes of an object;
- a function for retrieving the attributes of a relationship.

This parameterization allows the Reusable Graphical Browser to accommodate a wide range of OMSs. The Application Interface is presented in detail in Appendix A.

The interface between the application and the human operator (henceforth referred to as "the user"), via the the Reusable Graphical Browser and the underlying window system, is termed the "User Interface". It is primarily a graphical interface, although text input and output are also supported to a limited extent. In general, the user interacts with the application using a pointing device (e.g., a mouse) to select various items (buttons, scrollbars, etc.) depicted on a high-resolution graphics display. In X Window System terminology, these items are referred to as "widgets". The window system provides primitive operations for constructing and manipulating widgets. The Reusable Graphical Browser uses these primitive operations to compose more sophisticated widgets, and provides abstract operations for defining and manipulating them at the Application Interface. Some of these widgets are windows in which text can be displayed, or into which text can be typed using a keyboard; this is how text input and output are supported.

The User Interface is tailorable with respect to both presentation style and content, though that tailoring is constrained in order to promote a common "look and feel" across all graphical browser tools. The term "content" refers to *what* information is presented, whereas the term "presentation style" refers to *how* it is presented. The ability to tailor presentation style is primarily attributable to facilities provided by the underlying window system. The only capability that the Reusable Graphical Browser, itself, provides in this area is the ability to override the default algorithm for laying out (i.e., positioning) objects and relationships on the screen. The ability to tailor content is attributable to facilities provided directly by the Reusable Graphical Browser at the Application Interface. In particular, it is attributable to the abstract operations provided for defining and manipulating items such as menus, command lists and views. The User Interface is presented in detail in Appendix B.

The combination of a generic Application Interface and a tailorable User Interface is what makes the Reusable Graphical Browser reusable. The generic Application Interface allows the tool builder to integrate the Reusable Graphical Browser with almost any OMS. The tailorable User Interface allows the tool builder to customize, in a constrained way, the appearance and behavior of the application as perceived by the user. In short, these features

provide the flexibility that a tool builder needs to adapt the Reusable Graphical Browser to various browsing applications.

Internally, the Reusable Graphical Browser is partitioned into two layers: an application interface layer, and a window interface layer. The application interface layer implements the Application Interface as a collection of abstract data types and utilities to manipulate them (in the manner of reference [1]). It consists of the following components:

- CALLBACKS – abstract data type for application-defined procedures to be invoked in response to user actions;
- MENUS – abstract data type for modal menus;
- CMD_LISTS – abstract data type for non-modal (continually selectable) commands;
- GRAPHS – abstract data type for application-defined graph structures;
- VIEWS – abstract data type for graph depiction information;
- TEXT – abstract data type for text to be displayed;
- VIEW_UTILITIES – utilities for constructing and filtering views;
- LAYOUT_ALGORITHMS – utilities for automatically laying-out (assigning display coordinates to) the nodes and arcs in a view;

The window interface layer implements an interface for graphical interaction with the user that is not specific to any particular window system. It serves to encapsulate the details of the underlying window system, thereby promoting portability and maintainability of the Reusable Graphical Browser itself.

2.2 Operation

In general, a graphical browser tool performs three functions:

1. it acquires information about the objects managed by the OMS and relationships among them;
2. it displays those objects and relationships graphically;
3. it allows the user to interact with those objects and relationships through the graphical display.

The Reusable Graphical Browser is mostly concerned with the latter two functions. It relies on the application to acquire information from the OMS. Then, given that information, the

Reusable Graphical Browser is able to display it graphically and allow the user to interact with it. The user's actions are communicated back to the application, which processes them accordingly. After processing each user action, the application returns control to the Reusable Graphical Browser so that the user may further interact with the display.

2.2.1 The ERA Data Model

In order for the application to communicate information about OMS objects and relationships to the Reusable Graphical Browser, a common data model is required. Essentially, the application must present the information in terms that the Reusable Graphical Browser can understand. This is where the Entity-Relationship-Attribute (ERA) data model comes in.

The Reusable Graphical Browser assumes that the information acquired from the OMS can be represented as a set of entities, a set of relationships between those entities, and possibly some attributes associated with individual entities and/or relationships. The entities need not be all of the same kind; the Reusable Graphical Browser allows the application to distinguish between different kinds of entities. The same is true of relationships. Attributes are treated as arbitrary text strings, the contents of which the Reusable Graphical Browser knows nothing about.

One nicety of the ERA data model is that there is a convenient graphical representation for it. In particular, it can be depicted quite elegantly as a directed graph. This is the approach taken by the Reusable Graphical Browser. The entities are depicted as nodes (vertices) of the graph, and the relationships are depicted as arcs (edges) connecting the nodes. One-way relationships are depicted as unidirectional arcs, whereas two-way relationships are depicted as bidirectional arcs. Different kinds of entities and relationships can be distinguished by marking the nodes or arcs with different icons (symbols). In addition, each node and each arc can be individually labelled with a single line of text. Attributes may be displayed as an integral part of the graph as well, though this requires that all attributes to appear in the graph be retrieved before the graph is displayed. Alternatively, they may be retrieved and displayed individually in response to user interactions with the graph.

2.2.2 Graphs

Graphs are the means by which an application communicates the contents of an OMS to the Reusable Graphical Browser. They provide a mapping between entities and relationships in the application domain and corresponding nodes and arcs in the Reusable Browser domain (note that henceforth the terms "node" and "arc" are used in the context of the Reusable Graphical Browser, whereas the terms "entity" and "relationship" are used in the context of the application). An application may define any number of graphs – though one is normally sufficient for most applications. A graph consist of a set of nodes (entities) and a set of arcs (relationships). Depending on the application, it may include the entire contents of the OMS or only a portion thereof.

The Reusable Graphical Browser provides an abstract data type for graphs, which the application uses to define them. Typically, the application declares an object of the graph data type, invokes a procedure to initialize it, and then invokes procedures to add nodes and arcs to the graph one at a time. For each node and each arc added to the graph, the application must specify a unique identifier mapping it to a corresponding entity or relationship in the application domain. In addition, to allow the Reusable Graphical Browser to distinguish between different kinds of entities and relationships, the application has the option of specifying a node kind or arc kind.

Although graphs contain information about how the arcs and nodes are connected, they do not contain any information about how to depict the individual nodes and arcs or about how the user is supposed to interact with them. That is the purpose of views.

2.2.3 Views

Views are the means by which an application controls the manner of depiction of a graph and the semantics of user interactions with that depiction. They allow the application to control such factors as which nodes and arcs are presented to the user, how they are laid out, how they are labelled, and what actions are taken in response to their selection by the user. An application may construct any number of views for a graph. Each view consists of a subset (not necessarily a proper subset) of the nodes and arcs in the graph, with various depiction and behavior parameters for each one. In addition, each view has a title that is displayed when the view is displayed and a set of commands that may be invoked by the user to operate on the view.

The Reusable Graphical Browser provides an abstract data type for views, which the application may use to construct them. Using the primitive operations provided by the abstract data type to construct a view can be tedious, however. Thus, a view utility is provided to automate this process. All in all, there are two alternative ways to construct a view:

1. using a utility to automatically select nodes and arcs from a graph and insert them into the view;
2. directly inserting nodes and arcs from a graph into the view, one at a time.

Regardless of which method is used, the application must first declare an object of the view data type. Using the first method, the application must also instantiate the appropriate view construction utility, providing functions that this utility requires to determine which nodes and arcs to include in the view and what depiction and behavior to assign to them. The application then applies the utility to a graph to produce the desired view.

Using the second method, the application invokes a procedure to initialize the view and then invokes procedures to insert nodes and arcs from the graph into the view one at a time. For each node and each arc inserted into the view, the application must explicitly specify the desired depiction and behavior.

Once a view has been defined, it may be laid out either by invoking an automatic layout algorithm to process the view or by explicitly assigning positions to each and every node and arc in the view on an individual basis. Once a view is laid out, it may be displayed to the user and the user may be allowed to interact with it. Multiple views may be displayed simultaneously, if desired.

Regardless of what layout algorithm an application uses, it is possible for the display to become cluttered with too many arcs and nodes. In order to reduce clutter, the views abstraction provides primitive operations that can be used to suppress or unsuppress individual nodes and arcs from a view's display. For convenience, a view utility that supports the filtering of multiple nodes and arcs in a single operation is also provided. This view utility uses application-defined predicates to determine which nodes and arcs of the view to suppress.

To facilitate navigation over the view, the view abstraction provides a mechanism for displaying the topology of a view alongside the main view display. This mechanism consists of an operation that turns the topology display on or off for a specified view. The portion of the topology display that corresponds to the portion of its associated view visible in the main view display is highlighted. The user can reposition a view within its main view display via interactions with its associated topology display.

2.2.4 Node and Arc Depictions

Nodes and arcs in a view are generally depicted by icons (symbols) that indicate the kind of node or arc. Alternatively, they may be depicted by labels indicating the node kind or arc kind. These icons or labels are normally sensitive to mouse events so that the user may interact with them using the mouse. In addition to an icon or label indicating its kind, each node or arc may (optionally) be depicted with a corresponding label that indicates its individual name. Furthermore, some or all of the attributes of each node or arc may also (optionally) be included in its depiction.

The icons or labels to be used for each kind of node and arc in a view are determined by specifications in an application-defined resource file. These resource specifications may also be used to indicate which kinds of nodes and arcs are to be sensitive to mouse events and which particular mouse events they are to be sensitive to. Other presentation parameters, such as fonts for labels, border widths, border patterns and highlighting styles may be specified via the resource file as well. Although these resource specifications cannot be changed during execution of the application, they may be changed between executions without even having to recompile or relink the application. Further information on resource files and how to use them to tailor the appearance of a browser application is provided in Appendix B.

The optional name labels and attributes associated with individual nodes and arcs in a view are determined differently. Rather than being predetermined by specifications in a resource file, they are determined programatically as each node and each arc is inserted into the view. This is necessary because their determination may require retrieval of information from the OMS. Name labels are determined by the application before it calls the node and

arc insertion procedures, in order that they may be passed as parameters. Attributes, on the other hand, are implicitly obtained by the node and arc insertion procedures via calls to the attribute retrieval functions specified when the browser was instantiated.

2.2.5 Layout Algorithms

Layout algorithms are a means of automatically assigning positions (display coordinates) to each node and each arc in a view. They are implemented as procedures. These procedures calculate the positions of the nodes and arcs in a view based upon their connectivity and depiction specifications. Although layout algorithms are intended to produce aesthetically appealing layouts automatically, they often fall short of the mark. One reason for their shortcomings is that the definition of what is "aesthetically appealing" may vary considerably from one application to another. Another reason is that most of the algorithms are computationally very expensive (in fact, in its most general form, the problem of how to layout a directed graph is NP-complete); for practical applications, tradeoffs often have to be made between aesthetics and performance.

The Reusable Graphical Browser provides several predefined layout procedures. These procedures are designed primarily for simplicity and speed, rather than for quality of layout. They are all based on a common algorithm, which is presented in reference [5], but they have improved upon that algorithm somewhat. These layout procedures also make use of a topological sort algorithm based on the one presented in reference [2]. An application may either use the default layout procedures provided by the Reusable Graphical Browser or define layout procedures of its own. It is not required to use the predefined layout procedures.

2.2.6 Command Lists

Command lists are the means by which an application specifies operations to be associated with a view. Each command list consists of a list of commands and subcommands that may be selected by the user at any time while the view is displayed. One command list is associated with each view when the view is created. The command list associated with a view is displayed when the view is displayed, and remains displayed as long as the view remains displayed.

The Reusable Graphical Browser provides an abstract data type for command lists, which the application uses to define them. Typically, the application declares an object of the command list data type, invokes procedures to initialize it, and then invokes procedures to define individual commands and subcommands one at a time. When initializing a command list, the application must specify the number of commands in the command list and the number of subcommands (if any) for each command. The effects of selecting a particular command or subcommand are determined by an application-defined callback procedure that is invoked when the selection is made. There is only one such procedure for each command list. The application must define this procedure after initializing the command list, but before displaying a view with which the command list is associated. If it fails to do so, the

user's selections from the command list will have no effect.

2.2.7 Menus

Menus are the means by which an application specifies operations to be associated with individual nodes and arcs in a view. In addition, they provide a means by which applications may input parameters for commands or subcommands selected from a command list. Each menu consists of a list of items, any one of which may be selected by the user while the menu is displayed. A menu may be displayed by the application at any time – although the most logical time to display a menu is immediately after the user has selected a node, arc, command or subcommand. The Reusable Graphical Browser supports three different ways of displaying a menu: one associates the menu with a specified node; another associates the menu with a specified arc; the third does not associate the menu with anything. When a menu appears, the user must select an item from it immediately; any other selection is ignored and causes the menu to disappear.

The Reusable Graphical Browser provides an abstract data type for menus, which the application uses to define them. Typically, the application declares an object of the menu data type, invokes a procedure to initialize it, and then invokes procedures to define the individual items in the menu one at a time. When initializing a menu, the application must specify the title of the menu and the number of items in the menu. The effects of selecting a particular item from the menu are determined by an application-defined callback procedure that is invoked when the selection is made. There is only one such procedure for each menu. The application must define this procedure after initializing the menu, but before displaying the menu. If it fails to do so, the user's selections from the menu will have no effect.

2.2.8 Text Displays

Text displays are the means by which an application displays textual information to the user. These are pop-up displays that the application may create as necessary. They are read-only; their contents cannot be modified directly by the user. An application may use these displays to present the values of attributes, the contents of various files, alert messages, or any other information that makes sense to present as text.

The Reusable Graphical Browser supports two kinds of text displays. One kind of text display, which is intended to display arbitrary text for an indefinite period of time, consists of a scrollable text window with window manager decorations and a "QUIT" button. The window manager decorations allow the user to move, resize, raise, lower or iconify the text display. The "QUIT" button allows the user to erase it altogether. The second kind of text display, which is intended to display alert messages, consists of a scrollable text window with a confirmation button (marked "OK") but no window manager decorations. The user must acknowledge the alert message immediately by selecting the confirmation button. Until the message is acknowledged, all other selections are disabled. Once acknowledged, the message disappears and other selections are reenabled.

The source for the first kind of text display may be either an in-memory buffer or a text file. The Reusable Graphical Browser provides an abstract data type for in-memory text buffers, which the application uses to define them. An abstract data type for text files is already provided by the standard Ada Text I/O facilities. The application may either create text files itself, using these facilities, or may use text files created by other programs (e.g., text editors). The source for the second kind of text display is an ordinary Ada text string.

Changes to the source of a text display are not immediately reflected in the display. Rather, the text display must be erased and redisplayed in order for the changes to become visible.

2.2.9 Dialog Boxes

Dialog boxes are the means by which an application obtains input from the user via the keyboard. These are pop-up displays consisting of a prompt string, a box into which the user may type input, and a pair of buttons for confirming or cancelling the input. Whenever a dialog box is displayed, the user must respond to it immediately; all other selections are disabled until the user has either confirmed or cancelled the input. The user supplies input for a dialog box by typing at the keyboard. When the user is satisfied with the input, he/she selects the confirmation button to transmit the input to the application. If the user decides to cancel the input, he/she selects the cancel button instead; this erases the dialog box without transmitting any input to the application.

The Reusable Graphical Browser provides an operation associated with text displays that prompts the user for input. This is the operation that displays a dialog box. The keyboard input supplied by the user is transmitted to the application via the event that is generated when the user selects the confirmation button.

2.2.10 Events

Events are the means by which user interactions are communicated to an application. Whenever the user makes a selection, the Reusable Graphical Browser is informed of that selection. It handles some kinds of selections itself (e.g., scrollbar selections). Others are reported to the application. This reporting is accomplished via events.

Events are records containing information that describes the user's selection. Different kinds of events contain different information. For example, the event generated when a user selects the confirmation button on a dialog box contains the input string and its length, whereas the event generated when a user selects a node contains information identifying which node of which view was selected.

An application may choose to receive events in one of two different ways:

1. as input parameters to callback procedures;

2. as output parameters from the browser mainloop.

Using the first method, the application predefines callback procedures to be invoked for every possible event. These callback procedures are then invoked automatically by the Reusable Graphical Browser whenever corresponding events occur. The second method is assumed by default for any events for which no callback has been defined. In this case, the Reusable Graphical Browser returns control to the application via a return from the "browse" procedure. One drawback to the second method is that the application must explicitly reinvoke the "browse" procedure in order to enable subsequent user inputs. Using the first method, user inputs are implicitly reenabled upon exiting the callback procedure.

2.2.11 Callbacks

Callbacks are the means by which an application specifies actions to be taken in response to various user-generated events. A callback is a value that designates a procedure (referred to as the "callback procedure"). This value can be passed between subprograms and stored in data structures just like any other value. Each displayed object that can be selected by a user has a callback value associated with it. That callback value designates the procedure that is to be invoked when the user selects the object.

The Reusable Graphical Browser provides an abstract data type for callbacks, which the application uses to define them. In order to define a callback, the application must instantiate a generic package that imports (as a generic parameter) the name of the callback procedure. That generic package, in turn, exports a function which returns a callback value designating the imported callback procedure.

Once a callback has been defined, it may be associated with objects (e.g., command lists, menus, nodes and arcs) to be displayed. The Reusable Graphical Browser provides procedures to set the callback value for each object.

3 Reusing The Reusable Browser

Recall, from the preceding section, that the Reusable Graphical Browser is intended to serve as a foundation for constructing a wide variety of graphical browser tools using the Ada programming language. Tool builders must, therefore, be provided with a description of how to reuse this software component. This section provides such a description. It begins with a general description of the intended model of reuse for the Reusable Graphical Browser, and concludes with step-by-step instructions describing how to use it to construct graphical browser tools.

3.1 Model of Reuse

The process of adapting the Reusable Graphical Browser to a particular browsing application is a two-part process. One part of the process is to integrate it with a particular OMS. The other part of the process is to tailor the user interface to the application.

The Reusable Graphical Browser takes the form of an Ada generic package. An application integrates it with a particular OMS by instantiating it with OMS-specific data types and subprograms. In particular, the application is required to provide the following parameters at the time of instantiation (though not necessarily in this order):

- a data type for values that uniquely identify entities;
- a data type for values that uniquely identify relationships;
- a data type for values that uniquely identify collections of entities and relationships;
- a data type for values that distinguish different kinds of entities;
- a data type for values that distinguish different kinds of relationships;
- a function for retrieving the attributes of an entity;
- a function for retrieving the attributes of a relationship;
- a function for hashing entity identifiers;
- a function for hashing relationship identifiers.

The following additional parameters may also be required, depending on the datatype :

- a function for determining whether two values identify the same entity;
- a function for determining whether two values identify the same relationship;
- a function for determining whether two values designate the same collection of entities and relationships.

Some of these parameters may not make sense for all applications. In order to get around this problem, applications may specify arbitrary data types or functions for such parameters. For example, an application that has no concept of multiple collections of entities and relationships may specify a data type such as the following for the identifier of a collection:

```
type collection_id is (anonymous);
```

(of course, the application can then define only one collection – i.e, only one graph). Similarly, an application that has no concept of attributes of a relationship may specify a function that always returns a null attribute string in place of the function for retrieving the attributes of a relationship. Also, an application that has no need to optimize the translation between OMS entities and relationships and browser nodes and arcs (respectively) may specify functions that always return a value of one (1) for the hash functions.

Once the Reusable Graphical Browser has been instantiated, the application uses facilities provided by the browser instance to display the contents of the OMS and to allow the user to interact with the display. The application first defines one or more graphs using primitive operations provided by the graphs abstraction. Next, it defines views of those graphs using either view construction utilities or primitive operations provided by the views abstraction. In the process of constructing a view, it defines a command list for the view and callbacks for the individual nodes and arcs that make up the view. The command list is defined via primitive operations provided by the command lists abstraction. In the process of defining the command list, the application defines a callback for the command list as well. The callbacks are defined by instantiating a generic callback package provided by the callback abstraction. Once the views have been defined, the application lays them out using either an automatic layout utility or primitive operations provided by the views abstraction. The application may then display a view via another operation provided by the views abstraction. Once a view has been displayed, the application invokes a browse procedure to turn control over to the window system so that the user can interact with the display. The window system notifies the browser instance when the user makes a selection. The browser instance, in turn, notifies the application of the user's selection. If a callback procedure has been defined for the selection, the application is notified by invoking that callback procedure and passing it an event describing the selection. The callback procedure may take any action at all in response to the selection, including actions that modify the display (e.g., displaying a menu). Upon completion of the callback procedure, control is again returned to the window system so that the user may interact further with the display. If no callback procedure has been defined for the selection, the application is notified by returning the event as an "out" parameter from the browse procedure instead. In this case, the application must reinvoke the browse procedure after processing the selection, in order to allow the user to interact further with the display.

The facilities provided by the browser instance give the application extensive control over the dynamic behavior of the user interface. The ability to define the contents of graphs, views, command lists, menus and the like allows the application to specify the information content of the display. The ability to specify the layout and depiction of individual nodes and arcs in a view allows the application to tailor the aesthetic appearance of the display. And lastly, the ability to specify the actions to be taken in response to user selections allows the application to control the sequencing of user interactions.

3.2 Step-By-Step Instructions

Detailed instructions for the tool builder, describing how to use the Reusable Graphical Browser to construct graphical browsing tools, are presented below. These instructions describe only how to design and code such tools – not how to compile and link them (compiling and linking are discussed in the Version Description Document). The instructions are illustrated with code fragments taken from a simple browser tool (a UNIX[1] file system browser) that was developed as a demonstration of the Reusable Graphical Browser. Complete source code for this demonstration program is distributed along with the source code for the Reusable Graphical Browser.

Note that the instructions make frequent reference to items declared in package *BROWSER*. For convenience, a listing of the specification of that package is included in Appendix A of this manual. The reader is directed to that listing for more complete descriptions of the referenced items.

3.2.1 Step 1: Define Data Types for Instantiation

The first step in constructing a browser tool is to define data types for instantiating the *BROWSER* package. As indicated in Subsection 3.1, the application must supply five data types when instantiating this package:

- a data type for values that uniquely identify entities;
- a data type for values that uniquely identify relationships;
- a data type for values that uniquely identify collections of entities and relationships;
- a data type for values that distinguish different kinds of entities;
- a data type for values that distinguish different kinds of relationships.

These data types must match the specifications stated in the generic formal part of package *BROWSER*. Predefined data types may suffice for some of these, though probably not for all of them. Even so, you may still wish to declare subtypes of the predefined types (for example, to make the application code more understandable) rather than use them directly.

The data types specified for the unique identifiers may be anything you desire (scalar types, arrays, records, whatever), provided that they have a sufficient range of values to guarantee uniqueness. Normally, the best idea is to use the same data types that the OMS uses for these identifiers, since the application can then use them to directly query and manipulate the contents of the OMS. There is, however, a performance consideration that you should be aware of concerning the choice of these data types: namely, that the amount of memory (in particular, heap space) used by the application depends somewhat on the amount of memory

[1] UNIX is a registered trademark of AT&T

used to store each identifier. This is a consequence of the fact that the Reusable Graphical Browser typically has to store a large number of these identifiers.

The data types specified for distinguishing different kinds of entities and relationships must be scalar types. Enumerated types are typically the best choice for these, although you are free to use integer or character types if you so desire.

If any of these required data types does not have meaning for your particular application, you may supply some arbitrary type when instantiating the browser – subject to the constraints described above. The predefined data types are often handy for these situations.

An example, showing how the UNIX file system browser defines data types for instantiating the *BROWSER* package, is given below:

```
1  -- Excerpt from application/browser_params.a:
2
3  subtype file_id is file_system.infoptr;  -- Identified by file system info
4
5  type link_id is  -- Identified by source and destination file IDs
6    record
7      source : file_id;
8      dest   : file_id;
9    end record;
10
11 type file_kind is (file, directory, other); -- Kinds of UNIX file system
12                                             -- entities
13
14 type link_kind is (structural); -- Kinds of UNIX file system relationships
15                                 -- (only one)
```

The UNIX file system browser declares the type "file_id" to uniquely identify objects maintained by the UNIX file system. The identifier, in this case, is an access value that designates a record containing information about the object. That information is obtained from the UNIX file system when the browser tool is activated. This browser tool also declares the type "file_kind" to distinguish between different kinds of objects maintained by the UNIX file system. Similarly, the type "link_id" is declared to uniquely identify relationships between these objects and the type "link_kind" is declared to distinguish between different kinds of relationships. As you can see, this browser tool defines only one kind of relationship; so all relationships that the tool deals with must be of this one kind. It uses a predefined type for the other data type required to instantiate the *BROWSER* package.

3.2.2 Step 2: Define Subprograms for Instantiation

The second step in constructing a browser tool is to define subprograms that are required for instantiation of the *BROWSER* package. As indicated in Subsection 3.1, the application

must supply four such subprograms:

- a function for retrieving the attributes of an entity;
- a function for retrieving the attributes of a relationship;
- a function for hashing entity identifiers;
- a function for hashing relationship identifiers.

If any of the identified datatypes is an access type or a private type, additional functions must be defined to compare values of these types :

- a function for determining whether two values identify the same entity;
- a function for determining whether two values identify the same relationship;
- a function for determining whether two values designate the same collection of entities and relationships.

These subprograms must match the specifications stated in the generic formal part of package *BROWSER*.

The functions for retrieving attributes are only necessary if the Reusable Graphical Browser is to automatically display the attributes for nodes and arcs when displaying a view. If this is not the case for your application, simply provide functions that always return a null attribute string. If, on the other hand, your application does require that the attributes for nodes and arcs be automatically displayed along with the view, you must provide functions that retrieve those attributes from the OMS and format them into a single text string. That string may contain ASCII linefeed (LF) characters to separate individual lines of text.

Similarly, the functions for hashing identifiers are not necessary for all applications. In particular, they are only necessary for applications that deal with a large number of entities and/or relationships. If your application falls into that category, you must provide hashing functions that transform the identifiers into integers between one and the number of hash buckets (inclusive). If not, simply provide functions that always return a value of one (1).

An example, showing how the UNIX file system browser defines subprograms for instantiating the *BROWSER* package, is given below:

```
1  -- Excerpts from application/browser_params_b.a:
2
3  function get_attributes (                    -- Return attributes of node
4      the_node:   file_id                      -- Node ID
5  ) return string is
6
```

```
7  begin
8
9  ----
10 --  Just return nothing
11 ----
12
13     return "";
14
15 end get_attributes;
16 ------------------------------------------------------------------------------
17 function hash (                          -- Hash node ID to a bucket number
18     the_node:   file_id;                 -- Node ID
19     number_of_buckets:  positive         -- Number of buckets
20 ) return positive is
21
22 begin
23
24 ----
25 --  Return the inode value modulo the number of buckets
26 ----
27
28     return ( positive ( the_node.sbuf.st_ino ) mod number_of_buckets ) + 1;
29
30 end hash;
31 ------------------------------------------------------------------------------
32 function get_attributes (          -- Return attributes of arc
33     the_arc:           link_id     -- Arc ID
34 ) return string is
35
36 begin
37
38 ----
39 --  Just return nothing
40 ----
41
42     return "";
43
44 end get_attributes;
45 ------------------------------------------------------------------------------
46 function hash (                    -- Hash arc ID to a bucket number
47     the_arc:           link_id;    -- Arc ID
48     number_of_buckets:  positive   -- Number of buckets
49 ) return positive is
50
51 begin
```

```
52
53  ----
54  --  Return the inode value of the destination file modulo the number of buckets
55  ----
56
57      return ( positive ( the_arc.dest.sbuf.st_ino ) mod number_of_buckets ) + 1;
58
59  end hash;
```

The functions for retrieving attributes are not necessary for this application. Therefore, functions that return null attribute string have been provided. The functions for hashing identifiers were easily constructed by applying modulo arithmetic to the unique identifiers that UNIX uses to identify the files.

3.2.3 Step 3: Instantiate The *BROWSER* Package

The third step in constructing a browser tool is to instantiate the *BROWSER* package. This is done by declaring an instance of the generic *BROWSER* package and supplying the required generic parameters. The *BROWSER* package has seven formal generic parameters that are required for instantiation and three that are optional. These parameters are matched by data types and functions that were defined in steps 1 and 2 – provided that you have performed those steps correctly.

An example, showing how the UNIX file system browser instantiates the generic *BROWSER* package, is given below:

```
1   -- Excerpt from application/browser_instance.a:
2
3   with browser;
4   with browser_params;
5
6   package browser_instance is new browser (
7       graph_id =>              integer,                    -- Graph ID
8   --  designates_same_graph => standard."=",               -- Equality operator
9       node_id =>               browser_params.file_id,     -- Node ID
10       node_kind =>             browser_params.file_kind,   -- Node kind
11       designates_same_node => browser_params."=",          -- Equality operator
12       hash_of_node =>          browser_params.hash,        -- Hash of node ID
13       arc_id =>                browser_params.link_id,     -- Arc ID
14       arc_kind =>              browser_params.link_kind,   -- Arc kind
15       hash_of_arc =>           browser_params.hash,        -- Hash of arc ID
16       designates_same_arc =>   browser_params."="          -- Equality operator
17  );
```

Note that the predefined data type, integer, is specified as the data type for unique identifiers of collections of entities and relationships (graphs). This choice of data types is arbitrary. The UNIX file system browser, having no need to communicate with the OMS (in this case UNIX) concerning collections of entities and relationships, simply uses a predefined type out of convenience. The other data types specified for instantiation of the *BROWSER* package were defined in step 1. The specified functions were defined in step 2.

3.2.4 Step 4: Define Graphs

The fourth step in constructing a browser tool is to communicate the contents of the OMS to the Reusable Graphical Browser. This is accomplished by defining one or more graphs representing collections of entities and relationships contained in (i.e., managed by) the OMS. First, declare the graphs as objects of type <browser_instance>.graphs.graph_type (where <browser_instance> denotes the name that you have given to your instance of the *BROWSER* package). These are the objects that you pass to subprograms that operate on graphs. Next, initialize these graph objects by calling the <browser_instance>.graphs.create_graph procedure for each one. The "estimated_nodes" and "estimated_arcs" parameters for this procedure are optional; they are only useful for applications that instantiate the *BROWSER* package with actual hash functions. After initializing the graphs, add individual entities (nodes) and relationships (arcs) to them by calling the <browser_instance>.graphs.add_node and <browser_instance>.graphs.add_arc procedures, respectively.

The following example illustrates this process by showing how the UNIX file system browser defines a graph for the contents of the directory hierarchy rooted at the current working directory. Before defining the graph, the UNIX file system browser first preserves the contents of that directory hierarchy. It does so by querying the UNIX file system and recording the information in an internal data structure. Once the contents of the directory hierarchy have been recorded in the internal data structure, a function is invoked to traverse that data structure and use the information contained therein to define a graph. That function is the "make_graph" function, listed below. The "make_graph" function makes use of the "addanode" function, also listed below, to add individual nodes and arcs to the graph.

```
1  -- Excerpts from application/utilities_b.a:
2
3  procedure addanode (
4      graph:      in out browser_instance.graphs.graph_type;
5      node:       in out browser_instance.graphs.node_type;
6      parent:     in     file_system.infoptr
7  ) is
8
9      curr:       file_system.infoptr;            -- Child object of interest
10      kid:        browser_instance.graphs.node_type;      -- Created node
11      arc:        browser_instance.graphs.arc_type;       -- Created arc
12
```

```
13  begin
14
15  ----
16  --  Walk through the children
17  ----
18
19      curr := parent.child;
20      while curr /= NULL
21      loop
22
23  ----
24  --  Create the child node
25  ----
26
27          browser_instance.graphs.add_node (
28              the_node =>     kid,
29              with_id =>      curr,
30              with_name =>    browser_instance.graphs.node_name ( curr.name ),
31              of_kind =>      to_file_kind (curr.sbuf.st_mode.s_ifmt),
32              to_graph =>     graph );
33
34  ----
35  --  Add an arc between the parent and this node
36  ----
37
38          browser_instance.graphs.add_arc (
39              the_arc =>      arc,
40              with_id =>      browser_params.link_id'(parent, curr),
41              from_node =>    node,
42              to_node =>      kid,
43              to_graph =>     graph );
44
45          arc_count := arc_count - 1;
46
47  ----
48  --  Recurse if this is a directory
49  ----
50
51          if file_system."/="(curr.child, NULL) then
52              addanode ( graph, kid, curr );
53          end if;
54
55          curr := curr.next;
56      end loop;
57
```

```
58  end addanode;
59  ------------------------------------------------------------------------
60  function make_graph (
61      root:       in      file_system.infoptr
62  ) return browser_instance.graphs.graph_type is
63
64      graph:      browser_instance.graphs.graph_type;     -- Graph object created
65      node:       browser_instance.graphs.node_type;      -- Node object created
66
67  begin
68
69  ---- Set initial arc count for assigning unique arc IDs:
70
71      arc_count := file_system.arc_count;
72
73  ----
74  --  Create the graph object
75  ----
76
77      browser_instance.graphs.create_graph (
78          the_graph =>            graph,
79          with_id =>              1,
80          estimated_arcs =>       positive ( file_system.arc_count + 1 ),
81          estimated_nodes =>      positive ( file_system.node_count ) );
82
83  ----
84  --  Now create the root node
85  ----
86
87      browser_instance.graphs.add_node (
88          the_node =>             node,
89          with_id =>              root,
90          with_name =>            browser_instance.graphs.node_name ( root.name ),
91          of_kind =>              to_file_kind(root.sbuf.st_mode.s_ifmt),
92          to_graph =>             graph );
93
94  ----
95  --  Create the rest of the graph from this node's children
96  ----
97
98      addanode ( graph, node, root );
99
100      return graph;
101
102  end make_graph;
```

3.2.5 Step 5: Define Views of Each Graph

The fifth step in constructing a browser tool is to define one or more views of each graph. Each view delineates a subset of the graph's nodes and arcs to be displayed, and indicates the manner in which individual nodes and arcs are to be depicted. Each view also specifies, via callbacks, the behavior of the application in response to selection of these nodes and arcs by the user. Furthermore, each view provides a list of commands that the user may select to perform operations on the view.

As mentioned in the preceding section, there are two alternative methods that can be used to define views.

1. use a utility to automatically select nodes and arcs from a graph and insert them into the view;
2. directly insert nodes and arcs from a graph into the view, one at a time.

Before you can define a view, you must decide which of these methods to use. Depending on the application, one method may be more appropriate than another.

Regardless of which method you use, you will have to declare an object of type <browser_instance>.views.view_type (where <browser_instance> denotes the name that you've given to your instance of the *BROWSER* package) to represent each view. These are the objects that you pass to subprograms that operate on views.

If you use one of the utilities provided by the <browser_instance>.view_utilities package (i.e., the first method) to construct the view, you will have to instantiate the utility before using it. This is necessary because the view construction utilities are generic. The utility for constructing a view from a graph requires eight generic parameters:

- a function that determines whether or not to include a given node from a graph;
- a function that determines the label for a given node;
- a function that determines the attributes for a given node;
- a function that determines the callback value for a given node;
- a function that determines whether or not to include a given arc from a graph;
- a function that determines the label for a given arc;
- a function that determines the attributes for a given arc;
- a function that determines the callback value for a given arc.

These functions must be defined before the view utility can be instantiated.

In order to define functions that determine callback values for the nodes and arcs of a view, you first have to define all of the callback procedures that may be invoked in response to selection of the nodes and arcs by the user. These procedures must either be declared as Ada library units or as subprograms in the outermost scope of a package that is itself a library unit. Once you have declared the callback procedures, you have to declare an instance of the <browser_instance>.callbacks.callback package for each of them. The callback value designating a particular callback procedure can then be obtained by invoking the function exported by the corresponding instance of the callback package.

You will also have to define a command list before defining the view. Every view has a command list associated with it. To define a command list, first declare an object of type <browser_instance>.cmd_lists.cmd_list_type.
Then, call the <browser_instance>.cmd_lists.create_cmd_list procedure to initialize the command list. Also, if there are to be subcommands for any of the commands in the command list, call the <browser_instance>.cmd_lists.create_subcmd_list procedure to initialize the subcommand list for each command. Then, call the <browser_instance>.cmd_lists.set_cmd procedure and the <browser_instance>.cmd_lists.set_subcmd procedure to define the individual commands and subcommands, respectively.

To define a view using the first method, first declare an object of type <browser_instance>.views.view_type. Then, invoke an appropriate instance of the <browser_instance>.view_utilities.construct_view procedure to construct the desired view of a graph. When invoking this procedure, you must specify four parameters:

- the view object for the view to be constructed;
- the graph object for the graph to use as input;
- the command list object for the command list to be associated with the view;
- a title for the view.

There are also two optional parameters that may be provided to improve performance in conjunction with a hashing function:

- the estimated number of arcs in the view;
- the estimated number of nodes in the view.

These are only estimates; they need not be exact. They are used to determine the number of hash buckets for the view.

To define a view using the second method, also declare an object of type <browser_instance>.views.view_type. Then, instead of invoking a view utility, invoke the <browser_instance>.

views.create_view procedure to initialize the view object. When invoking this procedure, you must specify the graph that the view is to be a view of, the command list to be associated with the view and a title for the view. You may also (optionally) specify the estimated number of arcs and nodes in the view, in order to improve performance. You may then insert individual nodes and arcs from the graph into the view by calling the <browser_instance>.views.insert_node procedure to insert each node and the <browser_instance>.views.insert_arc procedure to insert each arc.

The following example illustrates the first method described above, by showing how the UNIX file system browser defines two of its views. Before defining the view, the UNIX file system browser defines functions for instantiating the view construction utilities. But before even defining these functions, it must first define all of the callbacks for the nodes and arcs of the view. The callbacks for nodes and arcs are defined as follows:

```
1  -- Excerpts from application/callbacks_b.a
2
3  procedure process_node_selection (
4      to_process_event:       in browser_instance.event_ptr
5  );
6
7  procedure process_arc_selection (
8      to_process_event:       in browser_instance.event_ptr
9  );
10
11  package node_selection is new browser_instance.callbacks.callback (
12      the_procedure =>        process_node_selection
13  );
14
15  package arc_selection is new browser_instance.callbacks.callback (
16      the_procedure =>        process_arc_selection
17  );
```

The functions for instantiating the view construction utilities are defined as shown below:

```
1  -- Excerpts from application/utilities_b.a:
2
3  function always_true (
4      the_node:               browser_instance.graphs.node_type
5  ) return boolean is
6
7  begin
8      return true;
9  end always_true;
10  ------------------------------------------------------------------------
11  function file_name (
```

```
12      the_node:               browser_instance.graphs.node_type
13  ) return browser_instance.views.node_label is
14
15      leaf:                       file_system.infoptr;
16
17  begin
18      leaf := browser_instance.graphs.id_of ( the_node );
19
20      case leaf.sbuf.st_mode.s_ifmt is
21
22      when stat.s_unknown =>
23          return browser_instance.views.node_label ( leaf.name & '?' );
24
25      when stat.s_ififo =>
26          return browser_instance.views.node_label ( leaf.name );
27
28      when stat.s_ifchr =>
29          return browser_instance.views.node_label ( leaf.name );
30
31      when stat.s_ifdir =>
32          return browser_instance.views.node_label ( leaf.name & '/' );
33
34      when stat.s_ifblk =>
35          return browser_instance.views.node_label ( leaf.name );
36
37      when stat.s_ifreg =>
38          if leaf.sbuf.st_mode.s_iexecu or else
39              leaf.sbuf.st_mode.s_iexecg or else
40              leaf.sbuf.st_mode.s_iexeco then
41              return browser_instance.views.node_label ( leaf.name & '*' );
42          else
43              return browser_instance.views.node_label ( leaf.name );
44          end if;
45
46      when stat.s_iflnk =>
47          return browser_instance.views.node_label ( leaf.name & '@' );
48
49      when stat.s_ifsock =>
50          return browser_instance.views.node_label ( leaf.name & '=' );
51
52      when others =>
53          return browser_instance.views.node_label ( leaf.name );
54
55      end case;
56
```

```
57  end file_name;
58  ------------------------------------------------------------------------
59  function no_attributes (
60      the_node:               browser_instance.graphs.node_type
61  ) return browser_instance.views.attributes is
62  begin
63      return "";
64  end no_attributes;
65  ------------------------------------------------------------------------
66  function node_action (
67      the_node:               browser_instance.graphs.node_type
68  ) return browser_instance.callbacks.callback_type is
69
70  begin
71      return callbacks.node_callback;
72  end node_action;
73  ------------------------------------------------------------------------
74  function always_true (
75      the_arc:                browser_instance.graphs.arc_type
76  ) return boolean is
77
78  begin
79      return true;
80  end always_true;
81  ------------------------------------------------------------------------
82  function no_label (
83      the_arc:                browser_instance.graphs.arc_type
84  ) return browser_instance.views.arc_label is
85
86  begin
87      return "";
88  end no_label;
89  ------------------------------------------------------------------------
90  function arc_attributes (
91      the_arc:                browser_instance.graphs.arc_type
92  ) return browser_instance.views.attributes is
93    arc_id : browser_params.link_id
94           := browser_instance.graphs.id_of (the_arc);
95  begin
96      return browser_instance.views.attributes(
97                      "Source: " & arc_id.source.name & ascii.lf &
98                      "Dest:   " & arc_id.dest.name);
99  end arc_attributes;
100 ------------------------------------------------------------------------
101 function arc_action(
```

```
102      the_arc:                browser_instance.graphs.arc_type
103  ) return browser_instance.callbacks.callback_type is
104
105  begin
106      return callbacks.arc_callback;
107  end arc_action;
108  ------------------------------------------------------------------------
109  function is_directory (
110      the_node:               browser_instance.graphs.node_type
111  ) return boolean is
112  begin
113      return browser_instance.graphs.kind_of(the_node) = directory;
114  end is_directory;
115  ------------------------------------------------------------------------
116  function dest_is_directory (
117      the_arc:                browser_instance.graphs.arc_type
118  ) return boolean is
119  begin
120      return browser_instance.graphs.kind_of(
121        browser_instance.graphs.destination_of(the_arc)) = directory;
122  end dest_is_directory;
```

The view construction utilities are then instantiated as follows:

```
1  -- Excerpts from application/utilities_b.a:
2
3  procedure construct_view_of_all_files is
4    new browser_instance.view_utilities.construct_view (
5      include_node =>         always_true,
6      label_for_node =>       file_name,
7      attributes_for_node =>  no_attributes,
8      action_for_node =>      node_action,
9      include_arc =>          always_true,
10      label_for_arc =>        no_label,
11      attributes_for_arc =>   arc_attributes,
12      action_for_arc =>       arc_action
13    );
14  ------------------------------------------------------------------------
15  procedure construct_view_of_directories is
16    new browser_instance.view_utilities.construct_view (
17    new browser_instance.view_utilities.construct_view (
18      include_node =>         is_directory,
19      label_for_node =>       file_name,
20      attributes_for_node =>  no_attributes,
21      action_for_node =>      node_action,
```

```
22      include_arc =>          dest_is_directory,
23      label_for_arc =>        no_label,
24      attributes_for_arc =>   arc_attributes,
25      action_for_arc =>       arc_action
26    );
```

Finally, the views are constructed by invoking the view construction utilities like so:

```
1  -- Excerpt from application/utilities_b.a:
2
3  procedure make_full_view (      -- Make a view showing all files/directories
4      view :      in out  browser_instance.views.view_type;   -- The view to make
5      graph:      in      browser_instance.graphs.graph_type; -- To make it from
6      cmd_list:   in      browser_instance.cmd_lists.cmd_list_type -- For the view
7  ) is
8  begin
9
10     construct_view_of_all_files (the_view => view,
11                                  of_graph => graph,
12                                  with_cmd_list => cmd_list,
13                                  with_title => "All Files",
14                                  estimated_arcs => file_system.arc_count,
15                                  estimated_nodes => file_system.node_count);
16
17 end make_full_view;
18
19 procedure make_dir_view (       -- Make a view showing directories only
20     view :      in out  browser_instance.views.view_type;   -- The view to make
21     graph:      in      browser_instance.graphs.graph_type; -- To make it from
22     cmd_list:   in      browser_instance.cmd_lists.cmd_list_type -- For the view
23 ) is
24 begin
25
26     construct_view_of_directories (the_view => view,
27                                    of_graph => graph,
28                                    with_cmd_list => cmd_list,
29                                    with_title => "Directories Only",
30                                    estimated_arcs => file_system.arc_count,
31                                    estimated_nodes => file_system.node_count);
32
33 end make_dir_view;
34
35 -- Excerpt from application/main.a
36
37     -- Create a view of all files (and directories, etc.).
```

```
38
39      utilities.make_full_view (
40          view =>     globals.full_vista,
41          graph =>    graph,
42          cmd_list =>    static_cmds.full_view_cmd_list);
43
44      -- Create a view of only the directories.
45
46      utilities.make_dir_view (
47          view =>     globals.dir_vista,
48          graph =>    graph,
49          cmd_list =>    static_cmds.dir_view_cmd_list);
```

The command lists that are specified in the calls to the view construction utilities are static, so the UNIX file system browser has predefined them during elaboration of the static_cmds package. This is done by the following sequence of statements in the package body:

```
1  -- Excerpts from application/static_cmds_b.a
2
3  with browser_instance;
4  with callbacks;
5  pragma elaborate (callbacks);
6  package body static_cmds is
7
8    package cmd_lists renames browser_instance.cmd_lists;
9
10 begin
11
12   -- Initialize all static command lists
13
14   init_initial_cmd_list:
15   declare
16     the_cmd_list : cmd_lists.cmd_list_type renames initial_cmd_list;
17     the_callback : constant browser_instance.callbacks.callback_type
18                  := browser_instance.callbacks.no_callback;
19     n_cmds       : constant := 2;
20     item_1       : constant cmd_lists.cmd_item := "Select View";
21     item_2       : constant cmd_lists.cmd_item := "Quit";
22   begin
23     -- Create the cmd_list object.
24     cmd_lists.create_cmd_list(the_cmd_list, n_cmds);
25     -- Set up the cmd_list callback.
26     cmd_lists.set_action(the_cmd_list, the_callback);
27     -- Set the commands.
28     cmd_lists.set_cmd(in_cmd_list => the_cmd_list, the_cmd => 1,
```

```
29                          to_value => item_1);
30      cmd_lists.set_cmd(in_cmd_list => the_cmd_list, the_cmd => 2,
31                          to_value => item_2);
32    end init_initial_cmd_list;
33
34    init_full_view_cmd_list:
35    declare
36      the_cmd_list : cmd_lists.cmd_list_type renames full_view_cmd_list;
37      the_callback : constant browser_instance.callbacks.callback_type
38                   := callbacks.full_view_cmd_list_callback;
39      n_cmds       : constant := 3;
40      item_1       : constant cmd_lists.cmd_item := "Filter View";
41      item_2       : constant cmd_lists.cmd_item := "Topology";
42      item_3       : constant cmd_lists.cmd_item := "Quit";
43    begin
44      -- Create the cmd_list object.
45      cmd_lists.create_cmd_list(the_cmd_list, n_cmds);
46      -- Set up the cmd_list callback.
47      cmd_lists.set_action(the_cmd_list, the_callback);
48      -- Set the commands.
49      cmd_lists.set_cmd(in_cmd_list => the_cmd_list, the_cmd => 1,
50                          to_value => item_1);
51      cmd_lists.set_cmd(in_cmd_list => the_cmd_list, the_cmd => 2,
52                          to_value => item_2);
53      cmd_lists.set_cmd(in_cmd_list => the_cmd_list, the_cmd => 3,
54                          to_value => item_3);
55      -- Initialize subcommand lists.
56      declare
57        cmd_index : constant := 1; -- Filter View
58        n_subcmds : constant := 3;
59        subitem_1 : constant cmd_lists.cmd_item := "Suppress Files";
60        subitem_2 : constant cmd_lists.cmd_item := "Unsuppress Files";
61        subitem_3 : constant cmd_lists.cmd_item := "Unsuppress All";
62      begin
63        -- Create the subcommand list.
64        cmd_lists.create_subcmd_list(the_cmd_list, cmd_index, n_subcmds);
65        -- Set the subcommands.
66        cmd_lists.set_subcmd(in_cmd_list => the_cmd_list, the_cmd => cmd_index,
67                             the_subcmd => 1,
68                             to_value => subitem_1);
69        cmd_lists.set_subcmd(in_cmd_list => the_cmd_list, the_cmd => cmd_index,
70                             the_subcmd => 2,
71                             to_value => subitem_2);
72        cmd_lists.set_subcmd(in_cmd_list => the_cmd_list, the_cmd => cmd_index,
73                             the_subcmd => 3,
```

```
74                              to_value => subitem_3);
75      end;
76    end init_full_view_cmd_list;
77
78    init_dir_view_cmd_list:
79    declare
80      the_cmd_list : cmd_lists.cmd_list_type renames dir_view_cmd_list;
81      the_callback : constant browser_instance.callbacks.callback_type
82                   := callbacks.dir_view_cmd_list_callback;
83      n_cmds       : constant := 3;
84      item_1       : constant cmd_lists.cmd_item := "Filter View";
85      item_2       : constant cmd_lists.cmd_item := "Topology";
86      item_3       : constant cmd_lists.cmd_item := "Quit";
87    begin
88      -- Create the cmd_list object.
89      cmd_lists.create_cmd_list(the_cmd_list, n_cmds);
90      -- Set up the cmd_list callback.
91      cmd_lists.set_action(the_cmd_list, the_callback);
92      -- Set the commands.
93      cmd_lists.set_cmd(in_cmd_list => the_cmd_list, the_cmd => 1,
94                        to_value => item_1);
95      cmd_lists.set_cmd(in_cmd_list => the_cmd_list, the_cmd => 2,
96                        to_value => item_2);
97      cmd_lists.set_cmd(in_cmd_list => the_cmd_list, the_cmd => 3,
98                        to_value => item_3);
99      -- Initialize subcommand lists.
100      declare
101        cmd_index : constant := 1; -- Filter View
102        n_subcmds : constant := 1;
103        subitem_1 : constant cmd_lists.cmd_item := "Unsuppress All";
104      begin
105        -- Create the subcommand list.
106        cmd_lists.create_subcmd_list(the_cmd_list, cmd_index, n_subcmds);
107        -- Set the subcommands.
108        cmd_lists.set_subcmd(in_cmd_list => the_cmd_list, the_cmd => cmd_index,
109                             the_subcmd => 1,
110                             to_value => subitem_1);
111      end;
112    end init_dir_view_cmd_list;
113
114  end static_cmds;
```

3.2.6 Step 6: Lay Out Each View

The sixth step in constructing a browser tool is to lay out each view in preparation for displaying it. The process of laying out a view consists of setting its orientation and assigning positions (actually display coordinates) to each of the nodes and arcs in the view. This is typically accomplished by applying an automatic layout algorithm to the view. The Reusable Graphical Browser predefines utilities that implement four different variations of one particular layout algorithm. If these predefined layout utilities are suitable for your needs, you may use them. If not, you can define your own layout utilities using primitive operations provided by the <browser_instance>.views package.

The following is an implementation of the original layout algorithm on which the predefined layout utilities for the Reusable Graphical Browser are based. It is a good example of how to use the facilities provided by the <browser_instance>.views package to define a layout utility. This implementation is not suitable as a layout utility itself, however, because it does not lay out arcs.

```
1       procedure isi_grapher (the_view : in out views.view_type) is
2
3         -- An implementation of the original layout algorithm on which the
4         -- Reusable Graphical Browser's predefined layout utilities are based.
5         -- This is a literal implementation of the algorithm stated in a
6         -- research report by Gabriel Robins entitled, "The ISI Grapher:
7         -- A Portable Tool for Displaying Graphs Pictorially," reprinted from
8         -- Proceedings of Symboliikka '87, held in Helsinki, Finland, August
9         -- 17-18, 1987.  It is available from Information Sciences Institute
10         -- (ISI), 4676 Admiralty Way, Marina del Rey, CA 90292-6695, as
11         -- Report No. ISI/RS-87-196.
12
13         -- Data Objects:
14
15        x_pad : constant := 30; -- amount of horizontal spacing between nodes
16        last_y : natural := 0;
17        orientation : constant views.orientation_type := views.horizontal;
18
19        -- Specs for procedures needed to instantiate generics:
20
21        procedure zero_y_and_continue (the_node : in views.node_type;
22                                       continue : out boolean);
23
24        procedure zero_x_and_continue (the_node : in views.node_type;
25                                       continue : out boolean);
26
27        procedure layout_a_child (the_arc : in views.arc_type;
28                                  continue : out boolean);
```

```
29
30        procedure layout_a_parent (the_arc : in views.arc_type;
31                                   continue : out boolean);
32
33        procedure layout_y_if_root (the_node : in views.node_type;
34                                    continue : out boolean);
35
36        procedure layout_x_if_leaf (the_node : in views.node_type;
37                                    continue : out boolean);
38
39        procedure layout_y (the_node : in views.node_type);
40
41        procedure layout_x (the_node : in views.node_type);
42
43        -- Generic Instantiations:
44
45        procedure zero_y_for_all_nodes is
46          new views.iterate_nodes(visit => zero_y_and_continue);
47
48        procedure zero_x_for_all_nodes is
49          new views.iterate_nodes(visit => zero_x_and_continue);
50
51        procedure layout_y_for_children_of_node is
52          new views.iterate_arcs_from (visit => layout_a_child);
53
54        procedure layout_x_for_parents_of_node is
55          new views.iterate_arcs_to (visit => layout_a_parent);
56
57        procedure layout_y_for_all_roots is
58          new views.iterate_nodes (visit => layout_y_if_root);
59
60        procedure layout_x_for_all_leaves is
61          new views.iterate_nodes (visit => layout_x_if_leaf);
62
63        -- Procedure and Function Bodies:
64
65        procedure zero_y_and_continue (the_node : in views.node_type;
66                                       continue : out boolean) is
67        begin
68          views.set_position (of_node => the_node,
69            to_position => (views.get_position(the_node).x_coordinate,0));
70          continue := true;
71        end zero_y_and_continue;
72
73        procedure zero_x_and_continue (the_node : in views.node_type;
```

```
74                                      continue : out boolean) is
75      begin
76        views.set_position (of_node => the_node,
77          to_position => (0, views.get_position(the_node).y_coordinate));
78        continue := true;
79      end zero_x_and_continue;
80
81      function is_a_root (the_node : views.node_type) return boolean is
82      begin
83        return views.number_of_arcs_to(the_node) = 0;
84      end is_a_root;
85
86      function is_a_leaf (the_node : views.node_type) return boolean is
87      begin
88        return views.number_of_arcs_from(the_node) = 0;
89      end is_a_leaf;
90
91      procedure layout_y_if_root (the_node : in views.node_type;
92                                  continue : out boolean) is
93      begin
94        if is_a_root(the_node) then
95          layout_y(the_node);
96        end if;
97        continue := true;
98      end layout_y_if_root;
99
100     procedure layout_x_if_leaf (the_node : in views.node_type;
101                                 continue : out boolean) is
102     begin
103       if is_a_leaf(the_node) then
104         layout_x(the_node);
105       end if;
106       continue := true;
107     end layout_x_if_leaf;
108
109     procedure layout_a_child (the_arc : in views.arc_type;
110                               continue : out boolean) is
111       child : views.node_type := views.destination_of (the_arc);
112     begin
113       layout_y (the_node => child);
114       continue := true;
115     end layout_a_child;
116
117     procedure layout_a_parent (the_arc : in views.arc_type;
118                                continue : out boolean) is
```

```
119         parent : views.node_type := views.source_of(the_arc);
120       begin
121         layout_x (the_node => parent);
122         continue := true;
123       end layout_a_parent;
124
125       function has_any_unlayed_out_children(the_node : views.node_type)
126                                             return boolean is
127         result : boolean := false;
128         procedure check_child (the_arc : in views.arc_type;
129                                continue : out boolean) is
130           child : views.node_type := views.destination_of (the_arc);
131         begin
132           if views.get_position(child).y_coordinate = 0 then
133             result := true;
134             continue := false;
135           else
136             continue := true;
137           end if;
138         end check_child;
139         procedure scan_children is
140           new views.iterate_arcs_from (visit => check_child);
141       begin
142         scan_children (the_node => the_node);
143         return result;
144       end has_any_unlayed_out_children;
145
146       function average_y_for_children_of_node (the_node : views.node_type)
147                                             return natural is
148         number_of_children : natural := 0;
149         total_y : natural := 0;
150         procedure check_child (the_arc : in views.arc_type;
151                                continue : out boolean) is
152           child : views.node_type := views.destination_of (the_arc);
153         begin
154           total_y := total_y + views.get_position(child).y_coordinate;
155           number_of_children := number_of_children + 1;
156           continue := true;
157         end check_child;
158         procedure scan_children is
159           new views.iterate_arcs_from (visit => check_child);
160       begin
161         scan_children (the_node => the_node);
162         return total_y / number_of_children;
163       end average_y_for_children_of_node;
```

```
164
165       procedure layout_y (the_node : in views.node_type) is
166       begin
167         if views.get_position(the_node).y_coordinate = 0 then
168           if has_any_unlayed_out_children(the_node) then
169             layout_y_for_children_of_node (the_node);
170             views.set_position(of_node => the_node,
171                       to_position => (views.get_position(the_node).x_coordinate,
172                                       average_y_for_children_of_node(the_node)));
173           else
174             views.set_position(of_node => the_node,
175               to_position => (views.get_position(the_node).x_coordinate,
176                 last_y + views.get_dimensions(the_node,orientation).height));
177             last_y := views.get_position(the_node).y_coordinate;
178           end if;
179         end if;
180       end layout_y;
181
182       function max_x_plus_width_of_parents (the_node : views.node_type)
183                                             return natural is
184         max : natural := 0;
185         temp : natural := 0;
186         procedure check_parent (the_arc : in views.arc_type;
187                                 continue : out boolean) is
188           parent : views.node_type := views.source_of(the_arc);
189         begin
190           temp := views.get_position(parent).x_coordinate +
191             views.get_dimensions(parent,orientation).width;
192           if temp > max then
193             max := temp;
194           end if;
195           continue := true;
196         end check_parent;
197         procedure scan_parents is
198           new views.iterate_arcs_to (visit => check_parent);
199       begin
200         scan_parents (the_node => the_node);
201         return max;
202       end max_x_plus_width_of_parents;
203
204       function has_parents (the_node : in views.node_type) return boolean is
205       begin
206         return views.number_of_arcs_to(the_node) /= 0;
207       end has_parents;
208
```

```
209         procedure layout_x (the_node : in views.node_type) is
210         begin
211           if views.get_position(the_node).x_coordinate = 0 then
212             if has_parents(the_node) then
213               layout_x_for_parents_of_node (the_node);
214               views.set_position(of_node => the_node,
215                     to_position => (max_x_plus_width_of_parents(the_node) + x_pad,
216                                     views.get_position(the_node).y_coordinate));
217             end if;
218           end if;
219         end layout_x;
220
221       begin
222
223         -- Pass 1:  Layout all Y coordinates, beginning at each root, and
224         -- traversing the graph depth-first.
225
226         zero_y_for_all_nodes (of_the_view => the_view);
227         last_y := 0;
228         layout_y_for_all_roots (of_the_view => the_view);
229
230         -- Pass 2:  Layout all X coordinates, beginning at each leaf, and
231         -- traversing the graph in the other direction (towards the roots)
232         -- depth-first.
233
234         zero_x_for_all_nodes (of_the_view => the_view);
235         layout_x_for_all_leaves (of_the_view => the_view);
236
237         -- Indicate orientation of layout.
238
239         views.set_orientation (of_view => the_view,
240                                to_value => orientation);
241
242
243       end isi_grapher;
```

In order to lay out a view, simply invoke the desired layout algorithm. For example, the UNIX file system browser lays out three views, using a different layout algorithm for each. This is done like so:

```
1   -- Excerpts from application/main.a:
2
3       -- Lay out the initial (empty) view.
4       -- A layout algorithm need not be invoked for an empty view, but its
5       -- orientation must still be set before the display routine can be called.
```

```
6
7       browser_instance.views.set_orientation (of_view => globals.init_vista,
8           to_value => browser_instance.views.horizontal);
9
10      -- Lay out the full view (all files, directories, etc.).
11
12      browser_instance.layout_algorithms.cyclic_horizontal (
13          the_view => globals.full_vista );
14
15      -- Lay out the directory view (directories only).
16
17      browser_instance.layout_algorithms.acyclic_vertical (
18          the_view => globals.dir_vista );
```

Note that the very least that a layout utility must do is to set the view orientation. This is required even for an empty view. If the view contains any nodes and/or arcs, the layout utility must also assign coordinates to each and every one of them.

3.2.7 Step 7: Display A View

The seventh step in constructing a browser tool is to display a view. Displaying a view is a simple matter of invoking the <browser_instance>.views.display_view procedure and specifying the desired view. The view title and the command list associated with the view are automatically displayed, and the individual nodes and arcs of the view are automatically displayed at their assigned positions and using their specified depictions.

The UNIX file system browser, for example, invokes the display_view procedure as follows:

```
1  -- Excerpt from application/main.a:
2
3       browser_instance.views.display_view (the_view => globals.init_vista);
```

It is possible to display multiple views simultaneously by issuing such a call for each view to be displayed.

3.2.8 Step 8: Allow User Interaction With The Display

The eighth step in constructing a browser tool is to allow the user to interact with the displayed view(s). To do so, simply invoke the <browser_instance>.browse procedure.

If callbacks have been predefined for all possible user selections, this procedure never exits. It just continues processing user selections indefinitely. Each time the user makes a selection, the browse procedure invokes the appropriate callback procedure to process the selection.

If on the other hand there is some selection for which no callback procedure is defined, the <browser_instance>.browse procedure will exit when the user makes that selection. In that case, the application must be prepared to process the selection following the return from the <browser_instance>.browse procedure. Furthermore, it must reinvoke this procedure in order to subsequently reenable user interaction.

The UNIX file system browser invokes the browse procedure like so:

```
1  -- Excerpt from application/main.a:
2
3      loop
4        browser_instance.browse ( exit_event );
5        if browser_instance.views."="(exit_event.view, globals.init_vista)
6        then  -- event expected
7          case exit_event.kind is
8            when browser_instance.command_select =>
9              case exit_event.command is
10               when 1 =>
11                 browser_instance.views.display_menu (
12                   the_menu => static_menus.view_menu,
13                   for_view => globals.init_vista);
14               when 2 =>
15                 exit;
16               when others =>
17                 null;
18             end case;
19           when others =>  -- not the expected event
20             text_io.put_line ("Unexpected exit from browser.");
21             text_io.put_line ("Unhandled event: "
22               & browser_instance.event_kind'image (exit_event.kind));
23         end case;
24       else -- unexpected event
25         text_io.put_line ("Unexpected exit from browser.");
26         text_io.put_line ("Unhandled event: "
27           & browser_instance.event_kind'image (exit_event.kind));
28       end if;
29     end loop;
```

Since the UNIX file system browser does not define a callback procedure for the command list associated with the initial view, it must process any selections from that command list following the return from the browse procedure. Any other selections that causes the browse procedure to exit are treated as unexpected, since callback procedures should have been defined for all of them.

3.2.9 Step 9: Define Responses To User Selections

The nineth step in constructing a browser tool is to define responses to user selections. For selections for which callback procedures have been defined, this is a matter of specifying the bodies of the callback procedures. For selections for which no callback procedures have been defined, it is a matter of specifying the processing to be performed upon return from the <browser_instance>.browse procedure.

Callback procedures may perform whatever processing you want them to perform. Be forewarned, however, that while a callback procedure is executing no further user selections can be processed. Consequently, if your callback procedures perform too much processing, your tool will not be very responsive to user input.

Similarly, your tool may perform any processing whatsoever between invocations of the <browser_instance>.browse procedure; but no further selections can be processed until the next invocation. Therefore, if your tool performs too much processing between invocations of this procedure, it will not be very responsive to user input.

Here are some of the kinds of processing that you might want to perform in a callback procedure or upon return from a call to the browse procedure:

- create a menu;
- delete a menu;
- turn a view's topology display on or off;
- filter a view;
- display text;
- prompt for keyboard input;
- display a menu;
- layout or re-layout a view;
- redisplay the current view;
- display a different view;
- erase a view;
- create additional graphs;
- create additional views.

The choices at the top of this list involve the least processing; the choices at the bottom involve the most processing.

We have already looked at an example of how to process selections upon return from the browse procedure. We will now look at some examples of how to process selections within callback procedures.

WARNING: One thing you should never do from within a callback procedure is execute a call to <browser_instance>.browse. That procedure *IS NOT REENTRANT.*

In the case of the UNIX file system browser, the callback procedure defined for selection of nodes displays a static menu of operations that may be applied to the selected node. Which menu is displayed depends on which kind of node was selected. The body of this callback procedure looks like the following:

```
1  -- Excerpts from application/callbacks_b.a:
2
3  procedure process_node_selection (
4      to_process_event:       in browser_instance.event_ptr
5  ) is
6    node_kind : constant browser_params.file_kind
7              := browser_instance.graphs.kind_of(
8                 browser_instance.views.graph_node_of(to_process_event.node));
9    menu_for_node : browser_instance.menus.menu_type;
10 begin
11
12     case node_kind is
13       when file =>
14         menu_for_node := static_menus.file_node_menu;
15       when directory =>
16         menu_for_node := static_menus.dir_node_menu;
17       when other =>
18         menu_for_node := static_menus.other_node_menu;
19     end case;
20
21     browser_instance.views.display_menu (the_menu => menu_for_node,
22                                          for_node => to_process_event.node,
23                                          of_view => to_process_event.view);
24
25 end process_node_selection;
```

Upon returning from the node selection callback procedure, the user may then select an item from the displayed menu. The UNIX file system browser also defines callback procedures for selections from the node menus. These callback procedures are declared as follows:

```
1  -- Excerpts from application/callbacks_b.a:
```

```
2
3  procedure process_file_node_menu_selection (
4      to_process_event:       in browser_instance.event_ptr
5  );
6
7  procedure process_dir_node_menu_selection (
8      to_process_event:       in browser_instance.event_ptr
9  );
10
11  procedure process_other_node_menu_selection (
12      to_process_event:       in browser_instance.event_ptr
13  );
```

The processing that is performed by the menu callback depends on which item the user selects. For example, the body of the file node menu callback procedure looks like the following:

```
1  -- Excerpts from application/callbacks_b.a:
2
3  procedure process_file_node_menu_selection (
4      to_process_event:       in browser_instance.event_ptr
5  ) is
6
7    temp_display : browser_instance.text.display_type;
8
9    procedure suppress_arc (the_arc  : in browser_instance.views.arc_type;
10                           continue : out boolean) is
11     begin
12       browser_instance.views.set_suppression(of_arc => the_arc,
13                                              to_value => true);
14       continue := true;
15     end suppress_arc;
16
17     procedure suppress_arcs_to_node is
18          new browser_instance.views.iterate_arcs_to (visit => suppress_arc);
19
20     procedure remove_arc (the_arc  : in browser_instance.views.arc_type;
21                           continue : out boolean) is
22       local_arc : browser_instance.views.arc_type := the_arc;
23     begin
24       browser_instance.views.remove_arc(the_arc => local_arc,
25                                         from_view => to_process_event.view);
26       continue := true;
27     end remove_arc;
28
```

```
29    procedure remove_arcs_to_node is
30          new browser_instance.views.iterate_arcs_to (visit => remove_arc);
31
32  begin
33
34    case to_process_event.n_item is
35      when 1 => -- Display Attributes
36        display_node_attributes (to_process_event.n_node);
37      when 2 => -- Display Contents
38        declare
39          leaf : file_system.infoptr
40               := browser_instance.graphs.id_of (
41                  browser_instance.views.graph_node_of (to_process_event.n_node));
42        begin
43          browser_instance.text.display_text (
44            the_display => temp_display,
45            from_file => file_system.full_pathname(leaf),
46            use_title => "Contents of  " & string(browser_instance.views.label_of(
47                                to_process_event.n_node)),
48            quit_action => text_quit_callback);
49        end;
50      when 3 => -- Suppress File
51        browser_instance.views.set_suppression(of_node => to_process_event.n_node,
52                                               to_value => true);
53        suppress_arcs_to_node (the_node => to_process_event.n_node);
54        browser_instance.views.display_view (the_view => to_process_event.view);
55      when 4 => -- Delete File
56        remove_arcs_to_node (the_node => to_process_event.n_node);
57        browser_instance.views.remove_node (the_node => to_process_event.n_node,
58                                            from_view => to_process_event.view);
59        browser_instance.views.display_view (the_view => to_process_event.view);
60      when others =>
61        null;
62    end case;
63
64  end process_file_node_menu_selection;
```

A Appendix: Ada Specifications

The Ada specifications presented herein delineate the applications interface for the Reusable Graphical Browser. That interface consists of a single Ada package, named *BROWSER*. Package *BROWSER* bundles together all capabilities of the Reusable Graphical Browser. It is a generic package, and must therefore be instantiated by the application before use.

Within package *BROWSER* are a number of subpackages. These subpackages, which are listed below, represent various abstract data types and utilities to manipulate them:

- CALLBACKS – abstract data type for application-defined procedures to be invoked in response to user actions;
- MENUS – abstract data type for modal menus;
- CMD_LISTS – abstract data type for non-modal (continually selectable) commands;
- GRAPHS – abstract data type for application-defined graph structures;
- VIEWS – abstract data type for graph depiction information;
- TEXT – abstract data type for text to be displayed;
- VIEW_UTILITIES – utilities for constructing and filtering views;
- LAYOUT_ALGORITHMS – utilities for automatically laying out (assigning display coordinates to) the nodes and arcs in a view.

Extensive comments in the Ada specifications for package *BROWSER* make it fairly self-documenting. Unfortunately, these comments also make it difficult to quickly locate a particular declaration. The following index, which gives the line number of each declaration, should help considerably in that respect:

2066 procedure initialize
2074 procedure browse
2089 procedure quit;

```
1  --
2  -- START OF REUSABLE BROWSER PACKAGE SPEC
3  --
4  with system;
5  with x_windows;
6  WITH TEXT_IO; -- for stubs of unimplemented routines
7  WITH TBD;     -- ditto
8  with intrinsics;
9  with shell_public;
10  with command_public;
11  with node_public;
12  with arc_public;
13  with label_public;
14  with viewport_public;
15  with xw_bboard_public;
16  PRAGMA ELABORATE(TEXT_IO, TBD, intrinsics, shell_public, command_public,
17          node_public, arc_public, label_public, viewport_public,
18          xw_bboard_public); -- ditto
19  generic
20
21    -- Imported information about the instantiating application's graph objects.
22
23    type graph_id is private; -- uniquely identifies the instantiator's graphs
24    with function designates_same_graph (graph_a, graph_b : graph_id)
25                                          return boolean is "=";
26
27    -- Imported information about the instantiating application's node objects.
28
29    type node_id is private; -- uniquely identifies the instantiator's nodes
30    type node_kind is (<>); -- differentiates between different kinds of nodes
31    with function designates_same_node (node_a, node_b : node_id)
32                                        return boolean is "=";
33
34    -- Imported information about the instantiating application's arc objects.
35
36    type arc_id is private; -- uniquely identifies the instantiator's arcs
37    type arc_kind is (<>); -- differentiates between different kinds of arcs
38    with function designates_same_arc (arc_a, arc_b : arc_id)
39                                       return boolean is "=";
40
```

```
41    -- Imported hash functions, for improving performance when constructing
42    -- views with large numbers of nodes and arcs.
43
44    with function hash_of_node (the_node          : node_id;
45                                number_of_buckets : positive) return positive;
46      -- maps node IDs into integers in the range 1..number_of_buckets with
47      -- a linear (or near-linear) distribution.
48
49    with function hash_of_arc (the_arc           : arc_id;
50                               number_of_buckets : positive) return positive;
51      -- maps arc IDs into integers in the range 1..number_of_buckets with
52      -- a linear (or near-linear) distribution.
53
54  package browser is
55    --
56    -- A reusable graphical browser, capable of browsing over any object
57    -- management system whose objects and the relationships among them can
58    -- be depicted as a directed graph.  It supports application-defined
59    -- data types for unique node and arc identifiers.  It also supports
60    -- application-defined data types for distinguishing between different
61    -- kinds of nodes and arcs.  It supports the association of
62    -- application-defined attributes with individual nodes and arcs.  And
63    -- lastly, it supports tuning of translations between objects in the
64    -- application domain and those in the browser domain.
65    --
66
67    -- Browser Global Types:
68
69    -- Events are user actions that are reported to the application by the
70    -- reusable browser.  The following type defines the kinds of events
71    -- that are reported.
72    type event_kind is (position_select,
73                        command_select,
74                        subcommand_select,
75                        menu_item_select,
76                        node_menu_item_select,
77                        arc_menu_item_select,
78                        menu_cancel,
79                        node_select,
80                        arc_select,
81                        text_quit,
82                        string_input,
83                        input_cancel,
84                        browser_quit);
85    type event_info (kind : event_kind
```

```
86                          := event_kind'first); -- describes the event in detail
87      type event_ptr is access event_info; -- points to an event description
88
89      -- Browser Global Constants:
90
91      anticipated_arcs : constant := 100; -- optimized for this many by default
92      anticipated_nodes : constant := 100; -- optimized for this many by default
93
94      ---
95      package callbacks is
96        --
97        -- Abstraction for callback procedure type.  This abstraction allows the
98        -- the application to define procedures that will handle specific user
99        -- actions (events).
100        --
101
102       -- Types:
103
104       type callback_type is private; -- a handle for callback procedures
105
106       -- Constants:
107
108       no_callback : constant callback_type; -- a value corresponding to no
109         -- callback procedure; all objects of type callback_type are initialized
110         -- to this value by default.
111
112       -- Exceptions:
113
114       callback_undefined : exception; -- an attempt was made to call an undefined
115         -- callback procedure
116
117       -- Operations:
118
119       generic
120         with procedure the_procedure (to_process_event : in event_ptr);
121       package callback is
122         function proc_id return callback_type;
123       end callback;
124       --
125       -- Synopsis:  Instantiation of this package defines a callback procedure.
126       -- The generic parameter is the procedure to serve as a callback procedure.
127       -- The function proc_id returns a handle that can be used to refer
128       -- to the procedure (i.e., for the purpose of assigning it to process a
129       -- particular event).
130       -- WARNING:  THE SPECIFIED PROCEDURE MUST NOT BE NESTED WITHIN ANOTHER
```

```
131       -- PROCEDURE OR A TASK.  OTHERWISE, A PROGRAM_ERROR EXCEPTION MAY BE RAISED
132       -- WHEN IT IS INVOKED VIA "CALL".
133
134       procedure call (the_proc_id      : in callback_type;
135                       to_process_event : in event_ptr);
136       --
137       -- Synopsis:  This procedure is used to invoke a callback procedure.
138       -- It calls the specified procedure, passing it the specified event to
139       -- process.
140       -- If the specified procedure has not been defined as a callback procedure,
141       -- callback_undefined is raised.
142
143     private
144       type callback_rep;
145       type callback_type is access callback_rep;
146       no_callback : constant callback_type := null;
147     end callbacks;
148
149   ---
150     package menus is
151       --
152       -- Modal menus to be displayed (popped-up) by the browser, and from which
153       -- user selections are to be immediately obtained.
154       --
155
156       -- Types:
157
158       type menu_type is private; -- abstract type for a menu
159
160       type menu_title is new string; -- menu title to be displayed
161       type menu_item is new string; -- item in a menu
162       subtype item_index is positive; -- position of an item in a menu
163
164       -- Constants:
165
166       no_menu : constant menu_type; -- a value for an undefined menu;
167         -- all objects of type menu_type are initialized to this value by
168         -- default, and are set to this value when deleted.
169
170       -- Exceptions:
171
172       menu_undefined : exception; -- the specified menu has not been defined
173         -- or is no longer defined
174       menu_overflow : exception; -- there are insufficient resources available
175         -- to create or expand the menu
```

```
176     no_such_item : exception; -- there is no such item in the specified menu
177
178     -- Operations:
179
180     procedure create_menu (the_menu     : in out menu_type;
181                            with_title   : in menu_title;
182                          number_of_items  : in positive);
183     --
184     -- Synopsis:  This procedure creates a menu with the specified number
185     -- of items and gives it the specified title.  The menu is initially
186     -- empty (i.e., all of the items are blank).
187     -- Menu_overflow is raised if there are insufficient resources available
188     -- to create the menu.
189
190     procedure delete_menu (the_menu : in out menu_type);
191     --
192     -- Synopsis:  This procedure deletes the specified menu.
193     -- If the specified menu is not defined to begin with,
194     -- menu_undefined is raised.
195     -- WARNING:  THIS OPERATION MAY LEAVE DANGLING REFERENCES, IF THE
196     -- APPLICATION HAS CREATED MULTIPLE ALIASES FOR THE MENU VIA ASSIGNMENT.
197
198     procedure set_item (in_menu  : in menu_type;
199                         the_item : in item_index;
200                         to_value : in menu_item);
201     --
202     -- Synopsis:  This procedure sets the specified item in the specified
203     -- menu to the specified value.
204     -- If the specified menu has not been defined or has been deleted,
205     -- menu_undefined is raised.
206     -- If there is no such item as the specified item in the menu,
207     -- no_such_item is raised.
208     -- Menu_overflow is raised if there are insufficient resources available
209     -- to add the item to the menu.
210
211     function get_item (from_menu : menu_type;
212                        the_item : item_index) return menu_item;
213     --
214     -- Synopsis:  This function returns the specified item from the specified
215     -- menu.
216     -- If the specified menu has not been defined or has been deleted,
217     -- menu_undefined is raised.
218     -- If there is no such item as the specified item in the menu,
219     -- no_such_item is raised.
220
```

```
221     function title_of (the_menu : menu_type) return menu_title;
222     --
223     -- Synopsis:  This function returns the title of the specified
224     -- menu.
225     -- If the specified menu has not been defined or has been deleted,
226     -- menu_undefined is raised.
227
228     function number_of_items_in (the_menu : menu_type) return positive;
229     --
230     -- Synopsis:  This function returns the number of items in the specified
231     -- menu.
232     -- If the specified menu has not been defined or has been deleted,
233     -- menu_undefined is raised.
234
235     procedure set_action (the_menu   : in menu_type;
236                           the_action : in callbacks.callback_type);
237     --
238     -- Synopsis:  This procedure defines a callback procedure to be invoked
239     -- automatically when the user selects an item from the specified menu.
240     -- If the specified menu has not been defined or has been deleted,
241     -- menu_undefined is raised.
242
243     function get_action (the_menu : menu_type)
244                          return callbacks.callback_type;
245     --
246     -- Synopsis:  This function returns the previously defined callback
247     -- procedure for the specified menu.
248     -- If the specified menu has not been defined or has been deleted,
249     -- menu_undefined is raised.
250
251   private
252     type menu_structure (number_of_items : positive); -- full type declaration
253       -- deferred to the package body
254     type menu_type is access menu_structure;
255     no_menu : constant menu_type := null;
256   end menus;
257
258   ---
259   package cmd_lists is
260     --
261     -- Non-modal (continually selectable) commands to be associated with
262     -- individual browser views, specifying operations that may be
263     -- performed in those views at any time.  A two-level hierarchy
264     -- of commands is supported, consisting of top-level commands and their
265     -- associated subcommands (if any).
```

```
266     --
267
268     -- Types:
269
270     type cmd_list_type is private; -- abstract type for a list of commands
271
272     type cmd_item is new string; -- item in a command list
273
274     subtype cmd_index is positive; -- position of a command in a list
275
276     -- Constants:
277
278     no_cmd_list : constant cmd_list_type; -- a value for an undefined command
279       -- list; all objects of type cmd_list_type are initialized to this value
280       -- by default, and are set to this value when deleted.
281
282     -- Exceptions:
283
284     cmd_list_undefined : exception; -- the specified command list has not been
285       -- defined or is no longer defined
286     cmd_list_overflow : exception; -- there are insufficient resources available
287       -- to create or expand the command list
288     subcmds_already_exist : exception; -- a subcommand list already exists for
289       -- the specified command
290     no_such_cmd : exception; -- there is no such command in the specified
291       -- command list
292
293     -- Operations:
294
295     procedure create_cmd_list (the_cmd_list     : in out cmd_list_type;
296                                number_of_cmds   : in positive);
297     --
298     -- Synopsis:  This procedure creates a command list with room for the
299     -- specified number of commands.  The list is initially empty (i.e., all
300     -- of the items are blank).
301     -- Cmd_list_overflow is raised if there are insufficient resources available
302     -- to create the command list.
303
304     procedure create_subcmd_list (the_cmd_list  : in cmd_list_type;
305                                   the_cmd       : in cmd_index;
306                                   number_of_subcmds : in positive);
307     --
308     -- Synopsis:  This procedure creates a subcommand list associated with
309     -- the specified command and with room for the specified number of
310     -- subcommands.  The list is initially empty (i.e., all of the items
```

```
311     -- are blank).
312     -- If the specified command list has not been defined or has been deleted,
313     -- cmd_list_undefined is raised.
314     -- If there is no such command as the specified command in the command list,
315     -- no_such_cmd is raised.
316     -- If a subcommand list already exists for the specified command,
317     -- subcmds_already_exist is raised.
318     -- Cmd_list_overflow is raised if there are insufficient resources available
319     -- to create the subcommand list.
320
321     procedure delete_cmd_list (the_cmd_list : in out cmd_list_type);
322     --
323     -- Synopsis:  This procedure deletes the specified command list.
324     -- If the specified command list is not defined to begin with,
325     -- cmd_list_undefined is raised.
326     -- WARNING:  THIS OPERATION MAY LEAVE DANGLING REFERENCES, IF THE
327     -- APPLICATION HAS CREATED MULTIPLE ALIASES FOR THE COMMAND LIST VIA
328     -- ASSIGNMENT.
329
330     procedure delete_subcmd_list (the_cmd_list  : in cmd_list_type;
331                                   the_cmd       : in cmd_index);
332     --
333     -- Synopsis:  This procedure deletes the list of subcommands associated
334     -- with the specified command.
335     -- If the specified command list has not been defined or has been deleted,
336     -- cmd_list_undefined is raised.
337     -- If there is no such command as the specified command in the command list,
338     -- no_such_cmd is raised.
339     -- If the specified subcommand list is not defined to begin with,
340     -- cmd_list_undefined is raised.
341
342     procedure set_cmd (in_cmd_list : in cmd_list_type;
343                        the_cmd     : in cmd_index;
344                        to_value    : in cmd_item);
345     --
346     -- Synopsis:  This procedure sets the specified command in the specified
347     -- command list to the specified value.
348     -- If the specified command list has not been defined or has been deleted,
349     -- cmd_list_undefined is raised.
350     -- If there is no such command as the specified command in the command list,
351     -- no_such_cmd is raised.
352     -- Cmd_list_overflow is raised if there are insufficient resources available
353     -- to add the command to the command list.
354
355     procedure set_subcmd (in_cmd_list : in cmd_list_type;
```

```
356                         the_cmd     : in cmd_index;
357                         the_subcmd  : in cmd_index;
358                         to_value    : in cmd_item);
359     --
360     -- Synopsis:  This procedure sets the specified subcommand (of the
361     -- specified command) in the specified command list to the specified value.
362     -- If the specified command list has not been defined or has been deleted,
363     -- cmd_list_undefined is raised.
364     -- If there is no such command as the specified command in the command list,
365     -- no_such_cmd is raised.
366     -- If there is no such subcommand as the specified subcommand in the
367     -- command list, no_such_cmd is raised.
368     -- Cmd_list_overflow is raised if there are insufficient resources available
369     -- to add the subcommand to the command list.
370
371     function get_cmd (from_cmd_list : cmd_list_type;
372                       the_cmd       : cmd_index) return cmd_item;
373     --
374     -- Synopsis:  This function returns the specified command from the
375     -- specified command list.
376     -- If the specified command list has not been defined or has been deleted,
377     -- cmd_list_undefined is raised.
378     -- If there is no such command as the specified command in the command list,
379     -- no_such_cmd is raised.
380
381     function get_subcmd (from_cmd_list : cmd_list_type;
382                          the_cmd       : cmd_index;
383                          the_subcmd    : cmd_index) return cmd_item;
384     --
385     -- Synopsis:  This function returns the specified subcommand (of the
386     -- specified command) from the specified command list.
387     -- If the specified command list has not been defined or has been deleted,
388     -- cmd_list_undefined is raised.
389     -- If there is no such command as the specified command in the command list,
390     -- no_such_cmd is raised.
391     -- If there is no such subcommand as the specified subcommand in the
392     -- command list, no_such_cmd is raised.
393
394     function number_of_cmds_in (the_cmd_list : cmd_list_type)
395                                 return positive;
396     --
397     -- Synopsis:  This function returns the number of top-level commands in
398     -- the specified command list.
399     -- If the specified command list has not been defined or has been deleted,
400     -- cmd_list_undefined is raised.
```

```
401
402     function number_of_subcmds_of (the_cmd_list : cmd_list_type;
403                                    the_cmd      : cmd_index)
404                                   return natural;
405     --
406     -- Synopsis:  This function returns the number of subcommands associated
407     -- with the specified command.  If there are no subcommands of the specified
408     -- command in the specified command list, zero (0) is returned.
409     -- If the specified command list has not been defined or has been deleted,
410     -- cmd_list_undefined is raised.
411     -- If there is no such command as the specified command in the command list,
412     -- no_such_cmd is raised.
413
414     procedure set_action (the_cmd_list : in cmd_list_type;
415                           the_action   : in callbacks.callback_type);
416     --
417     -- Synopsis:  This procedure defines a callback procedure to be invoked
418     -- automatically when the user selects an item from the specified command
419     -- list.
420     -- If the specified command list has not been defined or has been deleted,
421     -- cmd_list_undefined is raised.
422
423     function get_action (the_cmd_list : cmd_list_type)
424                         return callbacks.callback_type;
425     --
426     -- Synopsis:  This function returns the previously defined callback
427     -- procedure for the specified command list.
428     -- If the specified command list has not been defined or has been deleted,
429     -- cmd_list_undefined is raised.
430
431   private
432     type cmd_list_structure (number_of_cmds : positive); -- full type
433       -- declaration deferred to the package body
434     type cmd_list_type is access cmd_list_structure;
435     no_cmd_list : constant cmd_list_type := null;
436   end cmd_lists;
437
438   ---
439   package graphs is
440     --
441     -- Graph structures defined by the application.  A graph structure
442     -- consists of a collection of of nodes and the arcs connecting them.
443     -- The reusable browser caches the application-defined unique identifier
444     -- (ID) associated with each node and each arc, as well as the
445     -- application-defined kind indication for each node and each arc.
```

```
446     -- The purpose of the graph structure is to translate between objects
447     -- in the browser domain and those in the application domain, and to
448     -- facilitate the construction and maintenance of views.
449     --
450
451     -- Types:
452
453     type graph_type is private; -- abstract type for a graph definition
454
455     type node_type is private; -- abstract type for a node
456     type node_name is new string; -- the textual name associated with a node
457
458     type arc_type is private; -- abstract type for an arc
459     type arc_name is new string; -- the textual name associated with an arc
460     type arc_direction is (one_way, two_ways);
461
462     -- Constants:
463
464     no_graph : constant graph_type; -- a value for an undefined graph;
465       -- all objects of type graph_type are initialized to this value
466       -- by default, and are set to this value when deleted.
467     no_node : constant node_type; -- a value for an undefined node;
468       -- all objects of type node_type are initialized to this value
469       -- by default, and are set to this value when deleted.
470     no_arc : constant arc_type; -- a value for an undefined arc;
471       -- all objects of type arc_type are initialized to this value
472       -- by default, and are set to this value when deleted.
473
474     -- Exceptions:
475
476     graph_already_exists : exception; -- a graph with the specified ID already
477       -- exists
478     graph_not_found : exception; -- the specified graph could not be found
479     graph_undefined : exception; -- the specified graph has not been defined or
480       -- is no longer defined
481     graph_overflow : exception; -- there are insufficient resources available
482       -- to create or expand the graph
483
484     node_already_exists : exception; -- a node with the specified ID already
485       -- exists
486     node_undefined : exception; -- the specified node has not been defined or
487       -- is no longer defined
488     node_not_found : exception; -- the specified node could not be found
489     node_has_references : exception; -- the specified node cannot be deleted,
490       -- because it is referenced by one or more arcs
```

```
491
492     arc_already_exists : exception; -- an arc with the specified ID already
493       -- exists
494     arc_undefined : exception; -- the specified arc has not been defined or
495       -- is no longer defined
496     arc_not_found : exception; -- the specified arc could not be found
497
498     -- Graph Operations:
499
500     procedure create_graph (    the_graph       : in out graph_type;
501                                 with_id         : in graph_id;
502                                 estimated_arcs  : in positive
503                                                 := anticipated_arcs;
504                                 estimated_nodes : in positive
505                                                 := anticipated_nodes);
506     --
507     -- Synopsis:  This procedure creates a graph definition having the
508     -- specified ID.  The translation of objects from the application domain
509     -- into objects in the browser domain is optimized for a graph having
510     -- the estimated number of arcs and nodes.
511     -- Attempting to create a second graph definition with the same ID
512     -- raises graph_already_exists.
513     -- Graph_overflow is raised if there are insufficient resources available
514     -- to create the graph definition.
515
516     procedure destroy_graph (   the_graph : in out graph_type);
517     --
518     -- Synopsis:  This procedure destroys the specified graph definition.
519     -- All arcs and nodes are deleted from the graph, and the graph becomes
520     -- undefined.
521     -- If the specified graph is not defined to begin with, graph_undefined
522     -- is raised.
523     -- WARNING:  THIS OPERATION MAY LEAVE DANGLING REFERENCES, IF THE
524     -- APPLICATION HAS CREATED MULTIPLE ALIASES FOR THE GRAPH OR FOR ANY
525     -- OF ITS NODES OR ARCS VIA ASSIGNMENT.
526
527     function is_defined (the_graph : graph_type) return boolean;
528     --
529     -- Synopsis:  This function indicates whether or not the specified graph
530     -- definition is currently defined (i.e., whether or not it currently
531     -- exists).
532
533     function id_of (    the_graph : graph_type) return graph_id;
534     --
535     -- Synopsis:  This function returns the ID of the specified graph
```

```
536     -- definition (i.e., the ID that was specified when the graph definition
537     -- was created).
538     -- If the specified graph definition has not been defined or has been
539     -- destroyed, graph_undefined is raised.
540
541     function get_graph (with_id   : graph_id) return graph_type;
542     --
543     -- Synopsis:  This function returns the graph having the specified ID.
544     -- If there is no such graph, graph_not_found is raised.
545
546     -- Node Operations:
547
548     procedure add_node (        the_node  : in out node_type;
549                                 with_id   : in node_id;
550                                 with_name : in node_name
551                                           := "";
552                                 of_kind   : in node_kind
553                                           := node_kind'first;
554                                 to_graph  : in graph_type);
555     --
556     -- Synopsis:  This procedure defines a node having the specified ID
557     -- and name, and distinguished as being of the specified kind, and adds
558     -- it to the specified graph definition.
559     -- Note that the node kind defaults to the base value of the node_kind
560     -- type.  This is useful for applications where there is no notion of
561     -- node kind (i.e., where all nodes are of the same kind).
562     -- If the specified graph definition has not been defined or has been
563     -- destroyed, graph_undefined is raised.
564     -- Attempting to add a second node with the same ID to the same graph
565     -- raises node_already_exists.
566     -- Graph_overflow is raised if there are insufficient resources available
567     -- to add the node.
568
569     procedure delete_node (     the_node   : in out node_type;
570                                 from_graph : in graph_type);
571     --
572     -- Synopsis:  This procedure deletes the specified node from the specified
573     -- graph.  The specified node is deleted and becomes undefined.
574     -- If the specified graph definition has not been defined or has been
575     -- destroyed, graph_undefined is raised.
576     -- If the specified node is not defined to begin with, node_undefined
577     -- is raised.
578     -- If there is no such node in the specified graph, node_not_found
579     -- is raised.
580     -- If the specified node is referenced by any arcs (as either a source
```

```
581     -- node or a destination node), node_has_references is raised.
582     -- WARNING:  THIS OPERATION MAY LEAVE DANGLING REFERENCES, IF THE
583     -- APPLICATION HAS CREATED MULTIPLE ALIASES FOR THE NODE VIA ASSIGNMENT.
584
585     function is_defined (       the_node    : node_type) return boolean;
586     --
587     -- Synopsis:  This function indicates whether or not the specified node is
588     -- currently defined.
589
590     function is_a_member_of(    the_node    : node_type;
591                                 the_graph   : graph_type) return boolean;
592     --
593     -- Synopsis:  This function indicates whether or not the specified node is
594     -- a member of the specified graph.
595     -- If the specified graph definition has not been defined or has been
596     -- destroyed, graph_undefined is raised.
597     -- If the specified node has not been defined or has been deleted,
598     -- node_undefined is raised.
599
600     function graph_of (the_node : node_type) return graph_type;
601     --
602     -- Synopsis:  This function returns the graph that the node was originally
603     -- added to.
604     -- If the specified node has not been defined or has been deleted,
605     -- node_undefined is raised.
606     -- If the graph's definition has been destroyed, graph_undefined is
607     -- raised.
608
609     function has_references (   the_node    : node_type) return boolean;
610     --
611     -- Synopsis:  This function indicates whether or not there are any arcs
612     -- currently referencing the specified node.
613     -- If the specified node has not been defined or has been deleted,
614     -- node_undefined is raised.
615
616     function id_of (    the_node : node_type) return node_id;
617     --
618     -- Synopsis:  This function returns the ID of the specified node (i.e.,
619     -- the ID that was specified when the node was defined).
620     -- If the specified node has not been defined or has been deleted,
621     -- node_undefined is raised.
622
623     function name_of (  the_node : node_type) return node_name;
624     --
625     -- Synopsis:  This function returns the name of the specified node (i.e.,
```

```
626     -- the name that was specified when the node was defined).
627     -- If the specified node has not been defined or has been deleted,
628     -- node_undefined is raised.
629
630     function kind_of (  the_node : node_type) return node_kind;
631     --
632     -- Synopsis:  This function returns the kind of the specified node (i.e.,
633     -- the node kind that was specified when the node was defined).
634     -- If the specified node has not been defined or has been deleted,
635     -- node_undefined is raised.
636
637     function get_node ( with_id    : node_id;
638                         from_graph : graph_type) return node_type;
639     --
640     -- Synopsis:  This function returns the node having the specified ID
641     -- from the specified graph.
642     -- If the specified graph definition has not been defined or has been
643     -- destroyed, graph_undefined is raised.
644     -- If there is no such node in the specified graph, node_not_found
645     -- is raised.
646
647     generic
648       with procedure visit (the_node : in  node_type;
649                             continue : out boolean);
650     procedure iterate_nodes (of_the_graph : in graph_type);
651     --
652     -- Synopsis:  This procedure "visits" each node of the graph, executing
653     -- the specified "visit" procedure for each node.  The iteration order
654     -- is not defined.
655     -- If the specified graph definition has not been defined or has been
656     -- destroyed, graph_undefined is raised.
657
658     -- Arc Operations:
659
660     procedure add_arc ( the_arc   : in out arc_type;
661                         with_id   : in arc_id;
662                         with_name : in arc_name
663                                   := "";
664                         from_node : in node_type;
665                         to_node   : in node_type;
666                         directed  : in arc_direction
667                                   := one_way;
668                         of_kind   : in arc_kind
669                                   := arc_kind'first;
670                         to_graph  : in graph_type);
```

```
671     --
672     -- Synopsis:  This procedure defines an arc having the specified ID
673     -- and name, connecting the specified nodes in the implied direction(s),
674     -- and distinguished as being of the specified kind, and adds it to
675     -- the specified graph definition.
676     -- Note that if no arc direction is specified, the arc is assumed to be
677     -- unidirectional (emanating from from_node and terminating at to_node).
678     -- Note also that the arc kind defaults to the base value of the arc_kind
679     -- type.  This is useful for applications where there is no notion of
680     -- arc kind (i.e., where all arcs are of the same kind).
681     -- If the specified graph definition has not been defined or has been
682     -- destroyed, graph_undefined is raised.
683     -- Attempting to add a second arc with the same ID to the same graph
684     -- raises arc_already_exists.
685     -- If either of the specified nodes has not been defined as a member of
686     -- the specified graph or has been deleted, node_undefined is raised.
687     -- Graph_overflow is raised if there are insufficient resources available
688     -- to add the arc.
689
690     procedure delete_arc (      the_arc    : in out arc_type;
691                                 from_graph : in graph_type);
692     --
693     -- Synopsis:  This procedure deletes the specified arc from the specified
694     -- graph.  The specified arc is deleted and becomes undefined.
695     -- If the specified graph definition has not been defined or has been
696     -- destroyed, graph_undefined is raised.
697     -- If the specified arc is not defined to begin with, arc_undefined
698     -- is raised.
699     -- If there is no such arc in the specified graph, arc_not_found
700     -- is raised.
701     -- WARNING:  THIS OPERATION MAY LEAVE DANGLING REFERENCES, IF THE
702     -- APPLICATION HAS CREATED MULTIPLE ALIASES FOR THE ARC VIA ASSIGNMENT.
703
704     function is_defined (       the_arc   : arc_type) return boolean;
705     --
706     -- Synopsis:  This function indicates whether or not the specified arc is
707     -- currently defined.
708
709     function is_a_member_of(    the_arc   : arc_type;
710                                 the_graph : graph_type) return boolean;
711     --
712     -- Synopsis:  This function indicates whether or not the specified arc is
713     -- a member of the specified graph.
714     -- If the specified graph definition has not been defined or has been
715     -- destroyed, graph_undefined is raised.
```

```
716     -- If the specified arc has not been defined or has been deleted,
717     -- arc_undefined is raised.
718
719     function graph_of (the_arc : arc_type) return graph_type;
720     --
721     -- Synopsis:  This function returns the graph that the arc was originally
722     -- added to.
723     -- If the specified arc has not been defined or has been deleted,
724     -- arc_undefined is raised.
725     -- If the graph's definition has been destroyed, graph_undefined is
726     -- raised.
727
728     function id_of (    the_arc : arc_type) return arc_id;
729     --
730     -- Synopsis:  This function returns the ID of the specified arc (i.e.,
731     -- the ID that was specified when the arc was defined).
732     -- If the specified arc has not been defined or has been deleted,
733     -- arc_undefined is raised.
734
735     function name_of (  the_arc : arc_type) return arc_name;
736     --
737     -- Synopsis:  This function returns the name of the specified arc (i.e.,
738     -- the name that was specified when the arc was defined).
739     -- If the specified arc has not been defined or has been deleted,
740     -- arc_undefined is raised.
741
742     function source_of (the_arc : arc_type) return node_type;
743     --
744     -- Synopsis:  This function returns the source node of the specified
745     -- arc (i.e., the node that was specified as from_node when the arc
746     -- was defined).
747     -- If the specified arc has not been defined or has been deleted,
748     -- arc_undefined is raised.
749
750     function destination_of (the_arc : arc_type) return node_type;
751     --
752     -- Synopsis:  This function returns the destination node of the specified
753     -- arc (i.e., the node that was specified as to_node when the arc was
754     -- defined).
755     -- If the specified arc has not been defined or has been deleted,
756     -- arc_undefined is raised.
757
758     function direction_of (the_arc : arc_type) return arc_direction;
759     --
760     -- Synopsis:  This function returns the direction of the specified arc
```

```
761      -- (i.e., the direction that was specified when the arc was defined).
762      -- This indicates whether the arc is unidirectional or bidirectional.
763      -- If the specified arc has not been defined or has been deleted,
764      -- arc_undefined is raised.
765
766      function kind_of (  the_arc : arc_type) return arc_kind;
767      --
768      -- Synopsis:  This function returns the kind of the specified arc (i.e.,
769      -- the arc kind that was specified when the arc was defined).
770      -- If the specified arc has not been defined or has been deleted,
771      -- arc_undefined is raised.
772
773      function get_arc (  with_id    : arc_id;
774                          from_graph : graph_type) return arc_type;
775      --
776      -- Synopsis:  This function returns the arc having the specified ID
777      -- from the specified graph.
778      -- If the specified graph definition has not been defined or has been
779      -- destroyed, graph_undefined is raised.
780      -- If there is no such arc in the specified graph, arc_not_found
781      -- is raised.
782
783      generic
784        with procedure visit (the_arc  : in  arc_type;
785                              continue : out boolean);
786      procedure iterate_arcs (of_the_graph : in graph_type);
787      --
788      -- Synopsis:  This procedure "visits" each arc of the graph, executing
789      -- the specified "visit" procedure for each arc.  The iteration order
790      -- is not defined.
791      -- If the specified graph definition has not been defined or has been
792      -- destroyed, graph_undefined is raised.
793
794    private
795      type graph_structure (node_hash_buckets : positive;
796                            arc_hash_buckets  : positive); -- full type
797        -- declaration deferred to the package body
798      type graph_type is access graph_structure;
799      type node_info; -- full type declaration deferred to the package body
800      type node_type is access node_info;
801      type arc_info; -- full type declaration deferred to the package body
802      type arc_type is access arc_info;
803      no_graph : constant graph_type := null;
804      no_node : constant node_type := null;
805      no_arc : constant arc_type := null;
```

```
806   end graphs;
807
808   ---
809   package views is
810     --
811     -- The user's view of application-defined graph structures, as presented
812     -- to the user by the reusable browser.  This view abstraction enables
813     -- the application to control both the manner of depiction of the graph
814     -- structure and the semantics of user interactions with the graph
815     -- structure.  In particular, it allows applications to control such
816     -- factors as which nodes and arcs are presented to the user, how they
817     -- are laid out on the display screen, how they are labelled, and what
818     -- actions are taken in response to their selection by the user.
819     -- Note that for some applications it may be desirable to present users
820     -- with several views of the same graph structure, or even views of
821     -- several different graph structures, simultaneously.  Different views
822     -- may show different graphs, different subsets of the same graph, or
823     -- the same subset of the same graph but depicted differently.
824     -- Alternatively, different views of the same graph may be depicted
825     -- identically, but may respond differently to user actions.
826     --
827
828     -- Types:
829
830     type view_type is private; -- abstract type for a view of a graph
831     type view_title is new string; -- view title to be displayed
832
833     type node_type is private; -- abstract type for a view of a node
834     type node_label is new string; -- label to be displayed for a node
835
836     type arc_type is private; -- abstract type for a view of an arc
837     type arc_label is new string; -- label to be displayed for an arc
838
839     type attributes is new string; -- attributes to be displayed for node/arc
840       -- multiple lines of text may be separated by ASCII.LF characters
841
842     type coordinates_type is
843       record
844         x_coordinate : natural := 0;
845         y_coordinate : natural := 0;
846       end record; -- specifies positions of nodes and arcs in the x-y plane
847
848     type dimensions_type is
849       record
850         width  : natural := 0; -- x dimension
```

```
851         height : natural := 0; -- y dimension
852       end record; -- specifies dimensions of nodes and arcs
853
854     type orientation_type is (none, vertical, horizontal); -- specifies
855       -- orientation of layout for the view
856
857     -- Constants:
858
859     no_view : constant view_type; -- a value for an undefined view;
860       -- all objects of type view_type are initialized to this value
861       -- by default, and are set to this value when deleted.
862     no_node : constant node_type; -- a value for an undefined node;
863       -- all objects of type node_type are initialized to this value
864       -- by default, and are set to this value when deleted.
865     no_arc : constant arc_type; -- a value for an undefined arc;
866       -- all objects of type arc_type are initialized to this value
867       -- by default, and are set to this value when deleted.
868
869     no_attributes : constant attributes := ""; -- no attributes for node/arc
870
871     -- Exceptions:
872
873     view_undefined : exception; -- the specified view has not been defined or
874       -- is no longer defined
875     view_overflow : exception; -- there are insufficient resources available
876       -- to create or expand the view
877     view_is_displayed : exception; -- the specified view is currently displayed,
878       -- so it cannot be deleted
879
880     node_undefined : exception; -- the specified node has not been defined or
881       -- is no longer defined
882     node_not_found : exception; -- the specified node is not in the view
883     node_has_references : exception; -- cannot remove a node that is the
884       -- destination of at least one arc.
885
886     arc_undefined : exception; -- the specified arc has not been defined or
887       -- is no longer defined
888     arc_not_found : exception; -- the specified arc is not in the view
889
890     not_laid_out : exception; -- the view was not laid out before displaying it
891
892     -- View Operations:
893
894     procedure create_view (the_view        : in out view_type;
895                            of_graph        : in graphs.graph_type;
```

```
                        with_cmd_list   : in cmd_lists.cmd_list_type;
                        with_title      : in view_title;
                        estimated_arcs  : in positive
                                        := anticipated_arcs;
                        estimated_nodes : in positive
                                        := anticipated_nodes);
    --
    -- Synopsis:  This procedure creates a view of the specified graph,
    -- providing the specified list of commands to operate on it, and
    -- assigns it the specified title.  Insertion, deletion and lookup
    -- operations involving arcs and nodes of the view are optimized for
    -- a view having the estimated number of arcs and nodes, respectively.
    -- The view is initially empty (i.e., it contains no nodes or arcs).
    -- If the specified graph is not defined or has been deleted,
    -- graphs.graph_undefined is raised.
    -- If the specified command list has not been defined or has been deleted,
    -- cmd_lists.cmd_list_undefined is raised.
    -- View_overflow is raised if there are insufficient resources available
    -- to create the view.

    procedure delete_view (     the_view : in out view_type);
    --
    -- Synopsis:  This procedure deletes the specified view.
    -- All arcs and nodes are removed from the view, and the view becomes
    -- undefined.
    -- If the specified view is not defined to begin with, view_undefined
    -- is raised.
    -- If the specified view is currently displayed, view_is_displayed
    -- is raised.
    -- WARNING:  THIS OPERATION MAY LEAVE DANGLING REFERENCES, IF THE
    -- APPLICATION HAS CREATED MULTIPLE ALIASES FOR THE VIEW OR FOR ANY
    -- OF ITS NODES OR ARCS VIA ASSIGNMENT.

    function is_defined (the_view : view_type) return boolean;
    --
    -- Synopsis:  This function indicates whether or not the specified view
    -- is currently defined (i.e., whether or not it currently exists).

    function graph_of (the_view: view_type) return graphs.graph_type;
    --
    -- Synopsis:  This function returns the graph that the view is a view of.
    -- If the specified view is not defined or has been deleted, view_undefined
    -- is raised.

    function cmd_list_of (the_view: view_type)
```

```
941                         return cmd_lists.cmd_list_type;
942     --
943     -- Synopsis:  This function returns the command list associated with
944     -- the specified view.
945     -- If the specified view is not defined or has been deleted, view_undefined
946     -- is raised.
947
948     function title_of (the_view: view_type) return view_title;
949     --
950     -- Synopsis:  This function returns the title of the specified
951     -- view.
952     -- If the specified view is not defined or has been deleted, view_undefined
953     -- is raised.
954
955     function number_of_nodes_in (the_view : view_type) return natural;
956     --
957     -- Synopsis:  This function returns the total number of nodes in the
958     -- specified view.
959     -- If the specified view has not been defined or has been deleted,
960     -- view_undefined is raised.
961
962     function number_of_arcs_in (the_view : view_type) return natural;
963     --
964     -- Synopsis:  This function returns the total number of arcs in the
965     -- specified view.
966     -- If the specified view has not been defined or has been deleted,
967     -- view_undefined is raised.
968
969     procedure set_orientation (of_view  : in view_type;
970                                to_value : in orientation_type);
971     --
972     -- Synopsis:  This procedure sets the orientation of the layout of the
973     -- specified view to the specified value (i.e., vertical or horizontal).
974     -- If the specified view has not been defined or has been deleted,
975     -- view_undefined is raised.
976
977     function get_orientation (of_view : view_type) return orientation_type;
978     --
979     -- Synopsis:  This function returns the orientation of the layout
980     -- of the specified view.
981     -- If the specified view has not been defined or has been deleted,
982     -- view_undefined is raised.
983
984     procedure set_arrow_spacing (of_view  : in view_type;
985                                  to_value : in natural);
```

```
986      --
987      -- Synopsis:  This procedure sets the spacing between the arrow heads
988      -- and tails drawn for the arcs emmanating from and terminating at each
989      -- node of the specified view.  The specified value indicates the number
990      -- of pixels to offset the arrow heads and tails from the center of each
991      -- node's icon.  For acyclic layouts (containing no backward-directed
992      -- arrows), a value of zero is appropriate; this is the default value if
993      -- the arrow spacing is not explicitly set.  For cyclic layouts, though,
994      -- a non-zero value should be set:  if the layout is vertical, the
995      -- specified value should not exceed one half the width of the narrowest
996      -- node icon; if the layout is horizontal, it should not exceed one half
997      -- the height of the shortest node icon.  These limits will be enforced
998      -- automatically for any nodes for which they are exceeded.
999      -- If the specified view has not been defined or has been deleted,
1000      -- view_undefined is raised.
1001
1002     function get_arrow_spacing (of_view : view_type) return natural;
1003     --
1004     -- Synopsis:  This function returns the arrow spacing value for the
1005     -- specified view.
1006     -- If the specified view has not been defined or has been deleted,
1007     -- view_undefined is raised.
1008
1009     procedure set_outdated_flag (of_view  : in view_type;
1010                                  to_value : in boolean);
1011     --
1012     -- Synopsis:  This procedure sets the outdated flag for the
1013     -- specified view to the specified value, indicating whether or not the
1014     -- view is currently up to date.
1015     -- If the specified view is not defined or has been deleted,
1016     -- view_undefined is raised.
1017
1018     function is_outdated (the_view : view_type) return boolean;
1019     --
1020     -- Synopsis:  This function returns an indication of whether of not the
1021     -- specified view is currently up to date.
1022     -- If the specified view is not defined or has been deleted, view_undefined
1023     -- is raised.
1024
1025     procedure set_topology_display (of_view  : in view_type;
1026                                     to_value : in boolean);
1027     --
1028     -- Synopsis:  This procedure sets the topology display flag for the
1029     -- specified view to the specified value, indicating whether or not the
1030     -- view's topology is to be displayed.  If the view is currently displayed,
```

```
1031      -- the topology display is immediately made visible or invisible (depending
1032      -- on the specified value).
1033      -- If the specified view is not defined or has been deleted,
1034      -- view_undefined is raised.
1035
1036      function get_topology_display (of_view : view_type) return boolean;
1037      --
1038      -- Synopsis:  This function returns the topology display flag for the
1039      -- specified view.
1040      -- If the specified view is not defined or has been deleted, view_undefined
1041      -- is raised.
1042
1043      procedure display_view (the_view : in view_type);
1044      --
1045      -- Synopsis: This procedure displays the specified view.  Only nodes and
1046      -- arcs that have been layed out and are not suppressed are displayed.
1047      -- If the view is already displayed, that display is revised to conform
1048      -- with current layout and suppression settings.
1049      -- If the specified view is not defined or has been deleted,
1050      -- view_undefined is raised.
1051      -- If the view has not been completely laid out (including specifying its
1052      -- orientation), not_laid_out is raised.
1053
1054      function is_displayed (the_view : view_type) return boolean;
1055      --
1056      -- Synopsis:  This function returns an indication of whether of not the
1057      -- specified view is currently displayed.
1058      -- If the specified view is not defined or has been deleted, view_undefined
1059      -- is raised.
1060
1061      procedure search_view (the_view : in view_type;
1062                            for_node : in graphs.node_type);
1063      --
1064      -- Synopsis:  This procedure searches the specified view for the specified
1065      -- node and (if found) centers it in the display area.
1066      -- If the specified view is not defined or has been deleted,
1067      -- view_undefined is raised.
1068      -- If the specified node is not defined or has been deleted,
1069      -- graphs.node_undefined is raised.
1070      -- If the specified node is not found in the specified view,
1071      -- node_not_found is raised.
1072
1073      procedure search_view (the_view : in view_type;
1074                            for_arc  : in graphs.arc_type);
1075      --
```

```
1076      -- Synopsis:  This procedure searches the specified view for the specified
1077      -- arc and (if found) centers it in the display area.
1078      -- If the specified view is not defined or has been deleted,
1079      -- view_undefined is raised.
1080      -- If the specified arc is not defined or has been deleted,
1081      -- graphs.arc_undefined is raised.
1082      -- If the specified arc is not found in the specified view,
1083      -- arc_not_found is raised.
1084
1085      procedure erase_view (the_view : in view_type);
1086      --
1087      -- Synopsis:  This procedure erases the display of the specified view.
1088      -- If the specified view is not currently displayed, this procedure has no
1089      -- effect.
1090      -- If the specified view is not defined or has been deleted,
1091      -- view_undefined is raised.
1092
1093      procedure display_menu (the_menu : in menus.menu_type;
1094                              for_view : in view_type);
1095      --
1096      -- Synopsis:  This procedure displays the specified modal menu, associating
1097      -- it with the specified view, and constrains user input exclusively
1098      -- to that menu.  The menu is erased automatically after the user has
1099      -- made a selection.
1100      -- If the specified menu is not defined or has been deleted,
1101      -- menus.menu_undefined is raised.
1102
1103      procedure display_menu (the_menu : in menus.menu_type;
1104                              for_node : in node_type;
1105                              of_view  : in view_type);
1106      --
1107      -- Synopsis:  This procedure displays the specified modal menu, associating
1108      -- it with the specified node, and constrains user input exclusively
1109      -- to that menu.  The menu is erased automatically after the user has
1110      -- made a selection.
1111      -- If the specified node is not defined or has been deleted,
1112      -- node_undefined is raised.
1113      -- If the specified menu is not defined or has been deleted,
1114      -- menus.menu_undefined is raised.
1115
1116      procedure display_menu (the_menu : in menus.menu_type;
1117                              for_arc  : in arc_type;
1118                              of_view  : in view_type);
1119      --
1120      -- Synopsis:  This procedure displays the specified modal menu, associating
```

```
1121      -- it with the specified arc, and constrains user input exclusively
1122      -- to that menu.  The menu is erased automatically after the user has
1123      -- made a selection.
1124      -- If the specified arc is not defined or has been deleted,
1125      -- arc_undefined is raised.
1126      -- If the specified menu is not defined or has been deleted,
1127      -- menus.menu_undefined is raised.
1128
1129      generic
1130        with procedure visit (the_view : in view_type;
1131                              continue : out boolean);
1132      procedure iterate_views;
1133      --
1134      -- Synopsis:  This procedure iterates through all views, and executes the
1135      -- specified "visit" procedure for each view.  The iteration order is not
1136      -- defined.  Iteration terminates when the "visit" procedure returns a
1137      -- value of FALSE for the continue parameter.
1138
1139      -- Node Operations:
1140
1141      procedure insert_node (     the_view_node  : in out node_type;
1142                                  of_graph_node  : in graphs.node_type;
1143                                  into_view      : in view_type;
1144                                  label_as       : in node_label
1145                                                 := "";
1146                                  set_attributes : in attributes
1147                                                 := no_attributes;
1148                                  set_action     : in callbacks.callback_type
1149                                                 := callbacks.no_callback);
1150      --
1151      -- Synopsis:  This procedure creates a view node corresponding to the
1152      -- specified graph node, indicating that the node is to be depicted with
1153      -- the specified label and attributes and that the specified action is to
1154      -- be taken upon node selection, and inserts it into the specified view.
1155      -- Note that by default the nodes are depicted with no labels or attributes.
1156      -- If the specified view has not been created or has been deleted,
1157      -- view_undefined is raised.
1158      -- View_overflow is raised if there are insufficient resources available
1159      -- to insert the node into the view.
1160      -- WARNING:  THERE IS NO PROTECTION AGAINST INSERTING TWO VIEW NODES THAT
1161      -- BOTH REFERENCE THE SAME GRAPH NODE INTO THE SAME VIEW; DOING SO MAY
1162      -- PRODUCE UNEXPECTED RESULTS.  IT IS THE APPLICATION'S RESPONSIBILITY
1163      -- TO ENSURE THAT THIS DOES NOT HAPPEN.
1164
1165      procedure remove_node (     the_node  : in out node_type;
```

```
1166                                    from_view : in view_type);
1167        --
1168        -- Synopsis:  This procedure removes the specified node from the specified
1169        -- view.  The specified node is deleted and becomes undefined.
1170        -- If the specified view has not been defined or has been
1171        -- deleted, view_undefined is raised.
1172        -- If the specified node is not defined to begin with, node_undefined
1173        -- is raised.
1174        -- If there is no such node in the specified view, node_not_found
1175        -- is raised.
1176        -- If the specified node is referenced by any arcs in the view (as either
1177        -- a source node or a destination node), node_has_references is raised.
1178        -- WARNING:  THIS OPERATION MAY LEAVE DANGLING REFERENCES, IF THE
1179        -- APPLICATION HAS CREATED MULTIPLE ALIASES FOR THE NODE VIA ASSIGNMENT.
1180
1181        function is_defined (       the_node   : node_type) return boolean;
1182        --
1183        -- Synopsis:  This function indicates whether or not the specified node is
1184        -- currently defined.
1185
1186        function is_a_member_of(    the_node : node_type;
1187                                    the_view : view_type) return boolean;
1188        --
1189        -- Synopsis:  This function indicates whether or not the specified node is
1190        -- a member of the specified view.
1191        -- If the specified view has not been defined or has been deleted,
1192        -- view_undefined is raised.
1193        -- If the specified node has not been defined or has been deleted,
1194        -- node_undefined is raised.
1195
1196        function graph_node_of (the_node : node_type) return graphs.node_type;
1197        --
1198        -- Synopsis:  This function returns the graph node that the view node is
1199        -- a view of.
1200        -- If the specified view node is not defined or has been removed,
1201        -- node_undefined is raised.
1202
1203        function is_in_view (graph_node : graphs.node_type;
1204                             the_view   : view_type) return boolean;
1205        --
1206        -- Synopsis:  This function indicates whether or not the specified graph
1207        -- node is depicted in the specified view.
1208        -- If the specified view has not been defined or has been deleted,
1209        -- view_undefined is raised.
1210        -- If the specified graph node has not been defined or has been deleted,
```

```
1211      -- graphs.node_undefined is raised.
1212
1213      function view_node_of (graph_node : graphs.node_type;
1214                             from_view  : view_type) return node_type;
1215      --
1216      -- Synopsis:  This function returns the node of the specified view
1217      -- that depicts the specified graph node.
1218      -- If the specified view has not been defined or has been destroyed,
1219      -- view_undefined is raised.
1220      -- If the specified graph node has not been defined or has been deleted,
1221      -- graphs.node_undefined is raised.
1222      -- If there is no such node in the specified view, node_not_found
1223      -- is raised.
1224
1225      function label_of ( the_node : node_type) return node_label;
1226      --
1227      -- Synopsis:  This function returns the label associated with the
1228      -- specified node (i.e., the label that was specified when the node was
1229      -- inserted into the view).
1230      -- If the specified node has not been defined or has been deleted,
1231      -- node_undefined is raised.
1232
1233      function has_attributes (the_node : node_type) return boolean;
1234      --
1235      -- Synopsis:  This function indicates whether or not any attributes have
1236      -- been set for the specified node.
1237      -- If the specified node has not been defined or has been deleted,
1238      -- node_undefined is raised.
1239
1240      procedure refresh_attributes (the_node       : in node_type;
1241                                    new_attributes : in attributes);
1242      --
1243      -- Synopsis:  This procedure refreshes the attributes displayed for the
1244      -- specified node.  If no attributes were set for the node when it was
1245      -- first inserted into the view, this procedure has no effect.
1246      -- If the specified node has not been defined or has been deleted,
1247      -- node_undefined is raised.
1248
1249      procedure set_action (the_node   : in node_type;
1250                            the_action : in callbacks.callback_type);
1251      --
1252      -- Synopsis:  This procedure defines a callback procedure to be invoked
1253      -- automatically when the user selects the specified node.
1254      -- If the specified node has not been defined or has been deleted,
1255      -- node_undefined is raised.
```

```
1256
1257      function get_action (the_node : node_type)
1258                           return callbacks.callback_type;
1259      --
1260      -- Synopsis:  This function returns the previously defined callback
1261      -- procedure for the specified node.
1262      -- If the specified node has not been defined or has been deleted,
1263      -- node_undefined is raised.
1264
1265      procedure set_position (of_node     : in node_type;
1266                              to_position : in coordinates_type);
1267      --
1268      -- Synopsis:  This procedure sets the x-y coordinates of the specified
1269      -- node to the specified values.
1270      -- If the specified node has not been defined or has been deleted,
1271      -- node_undefined is raised.
1272
1273      function get_position (of_node : node_type) return coordinates_type;
1274      --
1275      -- Synopsis:  This function returns the x-y coordinates of the specified
1276      -- node.
1277      -- If the specified node has not been defined or has been deleted,
1278      -- node_undefined is raised.
1279
1280      function get_dimensions (of_node          : node_type;
1281                               with_orientation : orientation_type)
1282                              return dimensions_type;
1283      --
1284      -- Synopsis:  This function returns overall the x-y dimensions (i.e.,
1285      -- width and height) for layout of the specified node, including its icon
1286      -- and any labels or attributes, in the specified orientation.
1287      -- If the specified node has not been defined or has been deleted,
1288      -- node_undefined is raised.
1289
1290      function get_icon_dimensions (of_node : node_type) return dimensions_type;
1291      --
1292      -- Synopsis:  This function returns the x-y dimensions (i.e., width and
1293      -- height) of the icon representing the node, exclusive of any labels or
1294      -- attributes.
1295      -- If the specified node has not been defined or has been deleted,
1296      -- node_undefined is raised.
1297
1298      procedure set_suppression (of_node  : in node_type;
1299                                 to_value : in boolean);
1300      --
```

```
1301      -- Synopsis:  This procedure sets the suppression flag for the
1302      -- specified node to the specified value, indicating whether or not
1303      -- the node is to be suppressed from the display.
1304      -- If the specified node has not been defined or has been deleted,
1305      -- node_undefined is raised.
1306
1307      function is_suppressed (the_node : node_type) return boolean;
1308      --
1309      -- Synopsis:  This function indicates whether or not the node is to be
1310      -- suppressed from the display.
1311      -- If the specified node has not been defined or has been deleted,
1312      -- node_undefined is raised.
1313
1314      function number_of_arcs_from (the_node : node_type) return natural;
1315      --
1316      -- Synopsis:  This function returns the number of arcs emanating from
1317      -- the specified node.
1318      -- If the specified node has not been defined or has been deleted,
1319      -- node_undefined is raised.
1320
1321      function number_of_arcs_to (the_node : node_type) return natural;
1322      --
1323      -- Synopsis:  This function returns the number of arcs terminating
1324      -- at (i.e., directed to) the specified node.
1325      -- If the specified node has not been defined or has been deleted,
1326      -- node_undefined is raised.
1327
1328      generic
1329        with procedure visit (the_node : in node_type;
1330                              continue : out boolean);
1331      procedure iterate_nodes (of_the_view: in view_type);
1332      --
1333      -- Synopsis:  This procedure iterates through all of the nodes in the
1334      -- specified view, and executes the specified "visit" procedure for each
1335      -- node.  The iteration order is not defined.  Iteration terminates when
1336      -- the "visit" procedure returns a value of FALSE for the continue
1337      -- parameter.
1338      -- If the specified view has not been defined or has been deleted,
1339      -- view_undefined is raised.
1340
1341      -- Arc Operations:
1342
1343      procedure insert_arc (      the_view_arc    : in out arc_type;
1344                                  of_graph_arc    : in graphs.arc_type;
1345                                  into_view       : in view_type;
```

```
1346                                label_as       : in arc_label
1347                                               := "";
1348                                set_attributes : in attributes
1349                                               := no_attributes;
1350                                set_action     : in callbacks.callback_type
1351                                               := callbacks.no_callback);
1352        --
1353        -- Synopsis:  This procedure creates a view arc corresponding to the
1354        -- specified graph arc, indicating that the arc is to be depicted with
1355        -- the specified label and attributes and that the specified action is to
1356        -- be taken upon arc selection, and inserts it into the specified view.
1357        -- Note that by default the arcs are depicted with no labels or attributes.
1358        -- If the specified view has not been created or has been deleted,
1359        -- view_undefined is raised.
1360        -- If either the source or destination node of the specified arc has not
1361        -- been inserted into the specified view or has been deleted,
1362        -- node_undefined is raised.
1363        -- View_overflow is raised if there are insufficient resources available
1364        -- to insert the arc into the view.
1365        -- WARNING:  THERE IS NO PROTECTION AGAINST INSERTING TWO VIEW ARCS THAT
1366        -- BOTH REFERENCE THE SAME GRAPH ARC INTO THE SAME VIEW.
1367
1368        procedure remove_arc (      the_arc   : in out arc_type;
1369                                    from_view : in view_type);
1370        --
1371        -- Synopsis:  This procedure removes the specified arc from the specified
1372        -- view.  The specified arc is deleted and becomes undefined.
1373        -- If the specified view has not been defined or has been
1374        -- deleted, view_undefined is raised.
1375        -- If the specified arc is not defined to begin with, arc_undefined
1376        -- is raised.
1377        -- If there is no such arc in the specified view, arc_not_found
1378        -- is raised.
1379        -- WARNING:  THIS OPERATION MAY LEAVE DANGLING REFERENCES, IF THE
1380        -- APPLICATION HAS CREATED MULTIPLE ALIASES FOR THE ARC VIA ASSIGNMENT.
1381
1382        function is_defined (       the_arc   : arc_type) return boolean;
1383        --
1384        -- Synopsis:  This function indicates whether or not the specified arc is
1385        -- currently defined.
1386
1387        function is_a_member_of(    the_arc  : arc_type;
1388                                    the_view : view_type) return boolean;
1389        --
1390        -- Synopsis:  This function indicates whether or not the specified arc is
```

```
1391      -- a member of the specified view.
1392      -- If the specified view has not been defined or has been deleted,
1393      -- view_undefined is raised.
1394      -- If the specified arc has not been defined or has been deleted,
1395      -- arc_undefined is raised.
1396
1397      function graph_arc_of (the_arc : arc_type) return graphs.arc_type;
1398      --
1399      -- Synopsis:  This function returns the graph arc that the view arc is
1400      -- a view of.
1401      -- If the specified view arc is not defined or has been removed,
1402      -- arc_undefined is raised.
1403
1404      function is_in_view (graph_arc : graphs.arc_type;
1405                           the_view  : view_type) return boolean;
1406      --
1407      -- Synopsis:  This function indicates whether or not the specified graph
1408      -- arc is depicted in the specified view.
1409      -- If the specified view has not been defined or has been deleted,
1410      -- view_undefined is raised.
1411      -- If the specified graph arc has not been defined or has been deleted,
1412      -- graphs.arc_undefined is raised.
1413
1414      function view_arc_of (graph_arc  : graphs.arc_type;
1415                            from_view  : view_type) return arc_type;
1416      --
1417      -- Synopsis:  This function returns the arc of the specified view
1418      -- that depicts the specified graph arc.
1419      -- If the specified view has not been defined or has been destroyed,
1420      -- view_undefined is raised.
1421      -- If the specified graph arc has not been defined or has been deleted,
1422      -- graphs.arc_undefined is raised.
1423      -- If there is no such arc in the specified view, arc_not_found
1424      -- is raised.
1425
1426      function label_of ( the_arc : arc_type) return arc_label;
1427      --
1428      -- Synopsis:  This function returns the label associated with the
1429      -- specified arc (i.e., the label that was specified when the arc was
1430      -- inserted into the view).
1431      -- If the specified arc has not been defined or has been deleted,
1432      -- arc_undefined is raised.
1433
1434      function source_of (the_arc : arc_type) return node_type;
1435      --
```

```
1436    -- Synopsis:  This function returns the source node of the specified
1437    -- arc (i.e., the node that was specified as from_node when the arc
1438    -- was defined).
1439    -- If the specified arc has not been defined or has been deleted,
1440    -- arc_undefined is raised.
1441
1442    function destination_of (the_arc : arc_type) return node_type;
1443    --
1444    -- Synopsis:  This function returns the destination node of the specified
1445    -- arc (i.e., the node that was specified as to_node when the arc was
1446    -- defined).
1447    -- If the specified arc has not been defined or has been deleted,
1448    -- arc_undefined is raised.
1449
1450    procedure redirect_arc (the_arc    : in arc_type;
1451                            of_view    : in view_type;
1452                            new_source : in node_type;
1453                            new_dest   : in node_type);
1454    --
1455    -- Synopsis:  This procedure redirects an arc from the newly-specified
1456    -- source node to the newly-specified destination node.  It is intended
1457    -- to support automatic layout procedures by allowing them to temporarily
1458    -- break cycles in a cyclic graph.  It is expected that such automatic
1459    -- layout procedures will restore redirected arcs to their original
1460    -- directions upon completion of the layout algorithm.
1461    -- If the specified view has not been created or has been deleted,
1462    -- view_undefined is raised.
1463    -- If the specified arc has not been defined or has been deleted,
1464    -- arc_undefined is raised.
1465    -- If there is no such arc in the specified view, arc_not_found
1466    -- is raised.
1467    -- If either of the specified nodes has not been inserted into
1468    -- the specified view or has been deleted, node_undefined is raised.
1469
1470    function has_attributes (the_arc : arc_type) return boolean;
1471    --
1472    -- Synopsis:  This function indicates whether or not any attributes have
1473    -- been set for the specified arc.
1474    -- If the specified arc has not been defined or has been deleted,
1475    -- arc_undefined is raised.
1476
1477    procedure refresh_attributes (the_arc        : in arc_type;
1478                                  new_attributes : in attributes);
1479    --
1480    -- Synopsis:  This procedure refreshes the attributes displayed for the
```

```
1481      -- specified arc.  If no attributes were set for the arc when it was
1482      -- first inserted into the view, this procedure has no effect.
1483      -- If the specified arc has not been defined or has been deleted,
1484      -- arc_undefined is raised.
1485
1486      procedure set_action (the_arc    : in arc_type;
1487                            the_action : in callbacks.callback_type);
1488      --
1489      -- Synopsis:  This procedure defines a callback procedure to be invoked
1490      -- automatically when the user selects the specified arc.
1491      -- If the specified arc has not been defined or has been deleted,
1492      -- arc_undefined is raised.
1493
1494      function get_action (the_arc : arc_type)
1495                           return callbacks.callback_type;
1496      --
1497      -- Synopsis:  This function returns the previously defined callback
1498      -- procedure for the specified arc.
1499      -- If the specified arc has not been defined or has been deleted,
1500      -- arc_undefined is raised.
1501
1502      procedure set_position (of_arc      : in arc_type;
1503                              to_position : in coordinates_type);
1504      --
1505      -- Synopsis:  This procedure sets the x-y coordinates of the specified
1506      -- arc to the specified values.
1507      -- If the specified arc has not been defined or has been deleted,
1508      -- arc_undefined is raised.
1509
1510      function get_position (of_arc : arc_type) return coordinates_type;
1511      --
1512      -- Synopsis:  This function returns the x-y coordinates of the specified
1513      -- arc.
1514      -- If the specified arc has not been defined or has been deleted,
1515      -- arc_undefined is raised.
1516
1517      function get_dimensions (of_arc : arc_type;
1518                               with_orientation : orientation_type)
1519                              return dimensions_type;
1520      --
1521      -- Synopsis:  This function returns overall the x-y dimensions (i.e.,
1522      -- width and height) for layout of the specified arc, including its icon
1523      -- and any labels or attributes, in the specified orientation.
1524      -- If the specified arc has not been defined or has been deleted,
1525      -- arc_undefined is raised.
```

```
1526
1527      function get_icon_dimensions (of_arc : arc_type) return dimensions_type;
1528      --
1529      -- Synopsis:  This function returns the x-y dimensions (i.e., width and
1530      -- height) of the icon representing the arc, exclusive of any labels or
1531      -- attributes.
1532      -- If the specified arc has not been defined or has been deleted,
1533      -- arc_undefined is raised.
1534
1535      procedure set_suppression (of_arc    : in arc_type;
1536                                 to_value : in boolean);
1537      --
1538      -- Synopsis:  This procedure sets the suppression flag for the
1539      -- specified arc to the specified value, indicating whether or not
1540      -- the arc is to be suppressed from the display.
1541      -- If the specified arc has not been defined or has been deleted,
1542      -- arc_undefined is raised.
1543
1544      function is_suppressed (the_arc : arc_type) return boolean;
1545      --
1546      -- Synopsis:  This function indicates whether or not the arc is to be
1547      -- suppressed from the display.
1548      -- If the specified arc has not been defined or has been deleted,
1549      -- arc_undefined is raised.
1550
1551      generic
1552        with procedure visit (the_arc  : in arc_type;
1553                              continue : out boolean);
1554      procedure iterate_arcs (of_the_view : in view_type);
1555      --
1556      -- Synopsis:  This procedure iterates through all of the arcs in the
1557      -- specified view, and executes the specified "visit" procedure for each
1558      -- arc.  The iteration order is not defined.  Iteration terminates when
1559      -- the "visit" procedure returns a value of FALSE for the continue
1560      -- parameter.
1561      -- If the specified view has not been defined or has been deleted,
1562      -- view_undefined is raised.
1563
1564      generic
1565        with procedure visit (the_arc  : in arc_type;
1566                              continue : out boolean);
1567      procedure iterate_arcs_from (the_node : in node_type);
1568      --
1569      -- Synopsis:  This procedure iterates through all of the arcs emanating
1570      -- from the specified node, and executes the specified "visit" procedure
```

```
1571      -- for each arc.  The iteration order is not defined.  Iteration terminates
1572      -- when the "visit" procedure returns a value of FALSE for the continue
1573      -- parameter.
1574      -- If the specified node has not been defined or has been deleted,
1575      -- node_undefined is raised.
1576
1577      generic
1578        with procedure visit (the_arc  : in arc_type;
1579                              continue : out boolean);
1580      procedure iterate_arcs_to (the_node : in node_type);
1581      --
1582      -- Synopsis:  This procedure iterates through all of the arcs terminating
1583      -- at the specified node, and executes the specified "visit" procedure
1584      -- for each arc.  The iteration order is not defined.  Iteration terminates
1585      -- when the "visit" procedure returns a value of FALSE for the continue
1586      -- parameter.
1587      -- If the specified node has not been defined or has been deleted,
1588      -- node_undefined is raised.
1589
1590    private
1591      type view_structure (node_hash_buckets : positive;
1592                           arc_hash_buckets  : positive); -- full type
1593        -- declaration deferred to the package body
1594      type view_type is access view_structure;
1595      type node_info; -- full type declaration deferred to the package body
1596      type node_type is access node_info;
1597      type arc_info; -- full type declaration deferred to the package body
1598      type arc_type is access arc_info;
1599      no_view : constant view_type := null;
1600      no_node : constant node_type := null;
1601      no_arc : constant arc_type := null;
1602    end views;
1603
1604    ---
1605    package text is
1606      --
1607      -- Abstract data types and utilities for text-based interactions.
1608      --
1609
1610      -- Types:
1611
1612      type buffer_type is private;
1613
1614      type display_type is private;
1615      type display_kind is (buffer_display, file_display);
```

```
1616
1617      -- Exceptions:
1618
1619      buffer_undefined : exception; -- the specified buffer does not exist
1620      buffer_overflow : exception; -- insufficient resources are available to
1621        -- create the buffer or to append to it
1622      buffer_in_use : exception; -- the buffer contents are currently displayed
1623
1624      display_undefined : exception; -- the specified text display does not exist
1625      display_overflow : exception; -- insufficient resources are available to
1626        -- create the text display
1627      wrong_display_kind : exception; -- operation not compatible with the
1628        -- kind of display specified
1629
1630      -- Constants:
1631
1632      no_display : constant display_type; -- a value for an undefined text
1633        -- display; all objects of type display_type are initialized to this
1634        -- value by default, and are set to this value when erased.
1635
1636      no_buffer : constant buffer_type; -- a value for an undefined text
1637        -- buffer; all objects of type buffer_type are initialized to this
1638        -- value by default, and are set to this value when deleted.
1639
1640      max_input_length : constant := 100; -- max length of arbitrary string input
1641
1642      -- Buffer Operations:
1643
1644      procedure create_buffer (the_buffer : in out buffer_type;
1645                               size       : in positive);
1646      --
1647      -- Synopsis:  This procedure creates a text buffer of the specified size
1648      -- (in characters).
1649      -- Buffer_overflow is raised if there are insufficient resources available
1650      -- to create the buffer.
1651
1652      procedure delete_buffer (the_buffer : in out buffer_type);
1653      --
1654      -- Synopsis:  This procedure deletes the specified text buffer.
1655      -- If the specified text buffer does not exist in the first place,
1656      -- buffer_undefined is raised.
1657      -- If the contents of the specified text buffer are currently being
1658      -- displayed, buffer_in_use is raised.
1659      -- WARNING:  THIS OPERATION MAY LEAVE DANGLING REFERENCES, IF THE
1660      -- APPLICATION HAS CREATED MULTIPLE ALIASES FOR THE TEXT BUFFER VIA
```

```
1661    -- ASSIGNMENT.
1662
1663    procedure clear_buffer (the_buffer : in out buffer_type);
1664    --
1665    -- Synopsis:  This procedure clears the specified text buffer.
1666    -- If the specified text buffer does not exist, buffer_undefined is raised.
1667
1668    procedure append (the_buffer : in out buffer_type;
1669                      the_text   : in string);
1670    --
1671    -- Synopsis:  This procedure appends the specified string to the
1672    -- specified text buffer.
1673    -- If the specified text buffer does not exist, buffer_undefined is raised.
1674    -- If appending the specified text to the buffer would cause it to overflow,
1675    -- buffer_overflow is raised.
1676
1677    procedure append_line (the_buffer : in out buffer_type;
1678                           the_text   : in string);
1679    --
1680    -- Synopsis:  This procedure appends the specified string and an end-of-
1681    -- line character to the specified text buffer.
1682    -- If the specified text buffer does not exist, buffer_undefined is raised.
1683    -- If appending this text to the buffer would cause it to overflow,
1684    -- buffer_overflow is raised.
1685
1686    procedure new_line (the_buffer : in out buffer_type;
1687                        count      : in positive := 1);
1688    --
1689    -- Synopsis:  This procedure appends the specified number of end-of-line
1690    -- characters to the specified text buffer.
1691    -- If the specified text buffer does not exist, buffer_undefined is raised.
1692    -- If appending the specified number of end-of-line characters to the
1693    -- buffer would cause it to overflow, buffer_overflow is raised.
1694
1695    function length_of (the_buffer : in buffer_type) return natural;
1696    --
1697    -- Synopsis:  This function returns the length of the contents of the
1698    -- specified text buffer, in characters.
1699    -- If the specified text buffer does not exist, buffer_undefined is raised.
1700
1701    function contents_of (the_buffer : in buffer_type) return string;
1702    --
1703    -- Synopsis:  This function returns the contents of the specified text
1704    -- buffer.
1705    -- If the specified text buffer does not exist, buffer_undefined is raised.
```

```
1706
1707      -- Display Operations:
1708
1709      procedure display_text (the_display : in out display_type;
1710                              from_buffer : in buffer_type;
1711                              use_title   : in string := " ";
1712                              quit_action : in callbacks.callback_type
1713                                          := callbacks.no_callback);
1714      --
1715      -- Synopsis:  This procedure displays text from the specified buffer on
1716      -- the screen, exhibiting the specified title, and returns a display object
1717      -- that may later be used to erase or refresh the display.  The text
1718      -- remains displayed until explicitly erased by the application (via a
1719      -- call to erase_text).  The user may request that the text display be
1720      -- erased by selecting its "QUIT" button.  In that case, the application
1721      -- is notified of the user's request via a text_quit event.  The
1722      -- application may specify a callback procedure (quit_action) to handle
1723      -- the text_quit event.
1724      -- If the specified text buffer does not exist, buffer_undefined is raised.
1725
1726      procedure display_text (the_display : in out display_type;
1727                              from_file   : in string;
1728                              use_title   : in string := " ";
1729                              quit_action : in callbacks.callback_type
1730                                          := callbacks.no_callback);
1731      --
1732      -- Synopsis:  This procedure displays text from the specified file (i.e.,
1733      -- the file having the specified external name) on the screen, exhibiting
1734      -- the specified title, and returns a display object that may later be used
1735      -- to erase or refresh the display.  The text remains displayed until
1736      -- explicitly erased by the application (via a call to erase_text).  The
1737      -- user may request that the text display be erased by selecting its
1738      -- "QUIT" button.  In that case, the application is notified of the user's
1739      -- request via a text_quit event.  The application may specify a callback
1740      -- procedure (quit_action) to handle the text_quit event.
1741
1742      procedure erase_text (the_display : in out display_type);
1743      --
1744      -- Synopsis:  This procedure erases the specified text display from the
1745      -- screen and then makes the display object undefined.
1746      -- If the specified display object is initially undefined,
1747      -- display_undefined is raised.
1748      -- WARNING:  THIS OPERATION MAY LEAVE DANGLING REFERENCES, IF THE
1749      -- APPLICATION HAS CREATED MULTIPLE ALIASES FOR THE DISPLAY OBJECT VIA
1750      -- ASSIGNMENT.
```

```
1751
1752      procedure refresh_text (the_display : in display_type);
1753      --
1754      -- Synopsis:  This procedure updates the specified text display, using
1755      -- the latest contents of its associated buffer or file.
1756      -- If the specified display object is undefined, display_undefined is
1757      -- raised.
1758
1759      function kind_of (the_display : display_type) return display_kind;
1760      --
1761      -- Synopsis:  This function indicates whether the text being displayed
1762      -- is from file or a buffer.
1763      -- If the specified display object is undefined, display_undefined is
1764      -- raised.
1765
1766      function get_buffer (the_display : display_type) return buffer_type;
1767      --
1768      -- Synopsis:  This function obtains the text buffer that is associated
1769      -- with the specified display.
1770      -- If the specified display object is undefined, display_undefined is
1771      -- raised.
1772      -- If the specified display is not of kind buffer_display,
1773      -- wrong_display_kind is raised.
1774
1775      function get_file (the_display : display_type) return string;
1776      --
1777      -- Synopsis:  This function obtains the name of the text file that is
1778      -- associated with the specified display.
1779      -- If the specified display object is undefined, display_undefined is
1780      -- raised.
1781      -- If the specified display is not of kind file_display,
1782      -- wrong_display_kind is raised.
1783
1784      -- Miscellaneous Text Utilities:
1785
1786      procedure display_alert (the_alert_msg : in string);
1787      --
1788      -- Synopsis:  This procedure displays an alert message (e.g., an error
1789      -- message), which remains displayed until acknowledged by the user.
1790      -- The user must acknowledge the alert message before any further
1791      -- selections can be made.
1792
1793      procedure prompt_for_input (the_prompt    : in string
1794                                                := " ";
1795                                  input_length : in positive
```

```
1796                                          := max_input_length;
1797                              the_action  : in callbacks.callback_type
1798                                          := callbacks.no_callback);
1799      --
1800      -- Synopsis:  This procedure prompts the user to type in a string of at
1801      -- most the specified input length, and causes the specified action
1802      -- routine to be invoked once the user has responded.  The application
1803      -- is notified of the user's response via a string_input event.  The
1804      -- application may define a callback procedure (the_action) to handle
1805      -- the string_input event.  The user must respond to the prompt before
1806      -- any further selections can be made.
1807
1808    private
1809      type buffer_structure (size : positive); -- full type declaration deferred
1810        -- to the package body
1811      type buffer_type is access buffer_structure;
1812      no_buffer : constant buffer_type := null;
1813      type display_info (kind : display_kind); -- full type declaration deferred
1814        -- to the package body
1815      type display_type is access display_info;
1816      no_display : constant display_type := null;
1817    end text;
1818
1819    ---
1820    package view_utilities is
1821      --
1822      -- Generic utility routines to facilitate the construction and maintenance
1823      -- of views.
1824      --
1825
1826      generic
1827
1828        -- Node selection function and functions to supply representation
1829        -- and behavioral characteristics for each node.
1830
1831        with function include_node (the_node : graphs.node_type)
1832                                    return boolean;
1833        with function label_for_node (the_node : graphs.node_type)
1834                                    return views.node_label;
1835        with function attributes_for_node (the_node : graphs.node_type)
1836                                    return views.attributes;
1837        with function action_for_node (the_node : graphs.node_type)
1838                                    return callbacks.callback_type;
1839
1840        -- Arc selection function and functions to supply representation
```

```
1841        -- and behavioral characteristics for each arc.
1842
1843        with function include_arc (the_arc : graphs.arc_type)
1844                                   return boolean;
1845        with function label_for_arc (the_arc : graphs.arc_type)
1846                                   return views.arc_label;
1847        with function attributes_for_arc (the_arc : graphs.arc_type)
1848                                   return views.attributes;
1849        with function action_for_arc (the_arc : graphs.arc_type)
1850                                   return callbacks.callback_type;
1851
1852      procedure construct_view (the_view        : in out views.view_type;
1853                                of_graph        : in graphs.graph_type;
1854                                with_cmd_list   : in cmd_lists.cmd_list_type;
1855                                with_title      : in views.view_title;
1856                                estimated_arcs  : in positive
1857                                                := anticipated_arcs;
1858                                estimated_nodes : in positive
1859                                                := anticipated_nodes);
1860      --
1861      -- Synopsis:  This procedure automates the construction of a view of a
1862      -- graph.  It  scans all nodes and arcs of the specified graph, applies
1863      -- the "include" function to determine whether or not to include each one,
1864      -- and inserts the included arcs and nodes into the view.
1865      -- The "label_for", "attributes_for" and "action_for" functions are applied
1866      -- to each included node and arc to obtain the required representation
1867      -- and behavioral characteristics.
1868      -- The view is created with the specified command list and title.
1869      -- Insertion, deletion and lookup operations involving arcs and nodes of
1870      -- the view are optimized for a view having the estimated number of arcs
1871      -- and nodes, respectively.
1872
1873      generic
1874
1875        -- Node selection function and functions to supply representation
1876        -- and behavioral characteristics for each node.
1877
1878        with function include_node (the_node : graphs.node_type)
1879                                   return boolean;
1880        with function label_for_node (the_node : graphs.node_type)
1881                                   return views.node_label;
1882        with function attributes_for_node (the_node : graphs.node_type)
1883                                   return views.attributes;
1884        with function action_for_node (the_node : graphs.node_type)
1885                                   return callbacks.callback_type;
```

```
1886
1887        -- Arc selection function and functions to supply representation
1888        -- and behavioral characteristics for each arc.
1889
1890        with function include_arc (the_arc : graphs.arc_type)
1891                                   return boolean;
1892        with function label_for_arc (the_arc : graphs.arc_type)
1893                                   return views.arc_label;
1894        with function attributes_for_arc (the_arc : graphs.arc_type)
1895                                   return views.attributes;
1896        with function action_for_arc (the_arc : graphs.arc_type)
1897                                   return callbacks.callback_type;
1898
1899        -- Layout algorithm for re-layout.
1900
1901        with procedure layout (the_view : in views.view_type);
1902
1903      procedure refresh_view (the_view : in views.view_type);
1904      --
1905      -- Synopsis:  This procedure updates a view to conform to the current
1906      -- state of its associated graph.  It scans all nodes and arcs of the
1907      -- graph, applies the "include" function to each one to determine which
1908      -- ones are included in the view, and updates the corresponding nodes
1909      -- and arcs of the view.  Included nodes and arcs that have been deleted
1910      -- from the graph are removed from the view.  Included nodes and arcs
1911      -- that have been added to the graph are inserted into the view.
1912      -- Other included nodes and arcs are updated to show the latest attribute
1913      -- values. The "label_for", "attributes_for" and "action_for" functions
1914      -- are applied to nodes and arcs of the graph to obtain the required
1915      -- representation and behavioral characteristics for insertion into the
1916      -- view.  The "attributes_for" functions are applied to nodes and arcs
1917      -- of the graph to obtain the latest attribute values.
1918      -- The view is then re-laid-out and re-displayed.
1919
1920      generic
1921
1922        -- Node selection function (selects which nodes to suppress).
1923
1924        with function suppress_node (the_node : views.node_type)
1925                                   return boolean;
1926
1927        -- Arc selection function (selects which arcs to suppress).
1928
1929        with function suppress_arc (the_arc : views.arc_type)
1930                                   return boolean;
```

```
1931
1932      procedure filter_view (the_view : in views.view_type);
1933      --
1934      -- Synopsis:  This procedure filters an existing view, by suppressing
1935      -- the display of some or all of its nodes and arcs.  The nodes are
1936      -- filtered first, then the arcs.  Any node that is not explicitly
1937      -- suppressed is unsuppressed by default.  Any arc that is not
1938      -- explicitly suppressed is unsuppressed if and only if neither its
1939      -- source nor destination node has been suppressed.  The filtering
1940      -- is not apparent to the user until the view is redisplayed (via a
1941      -- call to the views.display_view procedure).
1942
1943    end view_utilities;
1944
1945    ---
1946    package layout_algorithms is
1947      --
1948      -- Algorithms for laying out a view of a directed graph.
1949      --
1950
1951      -- Constants:
1952
1953      procedure cyclic_horizontal (the_view : in views.view_type;
1954                                   x_pad    : in natural := 30;
1955                                   y_pad    : in natural := 10);
1956      --
1957      -- Synopsis:  This procedure lays out the specified view horizontally,
1958      -- using an algorithm accomodates cycles.  The algorithm is based on
1959      -- an algorithm presented in the following paper:
1960      -- Robins, G., "The ISI Grapher:  a Portable Tool for Displaying Graphs
1961      -- Pictorially," ISI/RS-87-196, USC/Information Sciences Institute,
1962      -- reprinted from the Proceedings of Symboliikka '87, Helsinki, Finland,
1963      -- August 17-18, 1987.
1964      -- It has been optimized somewhat, however, and has been modified
1965      -- substantially to accomodate cycles and to lay out arcs as well as nodes.
1966      -- The x_pad and y_pad parameters specify the minimum spacing (in pixels)
1967      -- between node and arc depictions in the X and Y directions, respectively.
1968
1969      procedure acyclic_horizontal (the_view : in views.view_type;
1970                                    x_pad    : in natural := 30;
1971                                    y_pad    : in natural := 10);
1972      --
1973      -- Synopsis:  This procedure lays out the specified view horizontally
1974      -- as well, but uses an algorithm that does not handle cycles.  This is
1975      -- essentially the same algorithm as is used for the cyclic_horizontal
```

```
1976      -- layout procedure, but without the modifications to accomodate cycles.
1977      -- The x_pad and y_pad parameters specify the minimum spacing (in pixels)
1978      -- between node and arc depictions in the X and Y directions, respectively.
1979      -- Because it does not have to check for cycles, this algorithm is faster
1980      -- than the algorithm use for the cyclic_horizontal procedure.  Therefore,
1981      -- this procedure should be preferred for applications that can guarantee
1982      -- that the view does not contain a cycle.
1983      -- WARNING:  IF USED ON A VIEW CONTAINING A CYCLE, THIS PROCEDURE WILL
1984      -- EITHER HANG OR RAISE A STORAGE_ERROR EXCEPTION.
1985
1986      procedure cyclic_vertical (the_view : in views.view_type;
1987                                 x_pad    : in natural := 10;
1988                                 y_pad    : in natural := 30);
1989      --
1990      -- Synopsis:  This procedure lays out the specified view vertically,
1991      -- using an algorithm accomodates cycles.  The algorithm is based on
1992      -- an algorithm presented in the following paper:
1993      -- Robins, G., "The ISI Grapher:  a Portable Tool for Displaying Graphs
1994      -- Pictorially," ISI/RS-87-196, USC/Information Sciences Institute,
1995      -- reprinted from the Proceedings of Symboliikka '87, Helsinki, Finland,
1996      -- August 17-18, 1987.
1997      -- It has been rewritten to produce a vertical layout, rather than a
1998      -- horizontal layout.  It has also been optimized somewhat, and has been
1999      -- modified to accomodate cycles and to lay out arcs as well as nodes.
2000      -- The x_pad and y_pad parameters specify the minimum spacing (in pixels)
2001      -- between node and arc depictions in the X and Y directions, respectively.
2002
2003      procedure acyclic_vertical (the_view : in views.view_type;
2004                                  x_pad    : in natural := 10;
2005                                  y_pad    : in natural := 30);
2006      --
2007      -- Synopsis:  This procedure lays out the specified view vertically
2008      -- as well, but uses an algorithm that does not handle cycles.  This is
2009      -- essentially the same algorithm as is used for the cyclic_vertical
2010      -- layout procedure, but without the modifications to accomodate cycles.
2011      -- The x_pad and y_pad parameters specify the minimum spacing (in pixels)
2012      -- between node and arc depictions in the X and Y directions, respectively.
2013      -- Because it does not have to check for cycles, this algorithm is faster
2014      -- than the algorithm use for the cyclic_vertical procedure.  Therefore,
2015      -- this procedure should be preferred for applications that can guarantee
2016      -- that the view does not contain a cycle.
2017      -- WARNING:  IF USED ON A VIEW CONTAINING A CYCLE, THIS PROCEDURE WILL
2018      -- EITHER HANG OR RAISE A STORAGE_ERROR EXCEPTION.
2019
2020    end layout_algorithms;
```

```
2021
2022    ---
2023    -- The following is the full definition of type event, which describes
2024    -- user actions (events) in detail.
2025    type event_info (kind : event_kind
2026                          := event_kind'first) is -- describes the event in detail
2027      record
2028        view : views.view_type;
2029        case kind is
2030          when position_select =>
2031            position : views.coordinates_type;
2032          when command_select =>
2033            command : cmd_lists.cmd_index;
2034          when subcommand_select =>
2035            topcommand : cmd_lists.cmd_index;
2036            subcommand : cmd_lists.cmd_index;
2037          when menu_item_select =>
2038            menu : menus.menu_type;
2039            item : menus.item_index;
2040          when node_menu_item_select =>
2041            n_node : views.node_type;
2042            n_menu : menus.menu_type;
2043            n_item : menus.item_index;
2044          when arc_menu_item_select =>
2045            a_arc : views.arc_type;
2046            a_menu : menus.menu_type;
2047            a_item : menus.item_index;
2048          when menu_cancel =>
2049            null;
2050          when node_select =>
2051            node : views.node_type;
2052          when arc_select =>
2053            arc : views.arc_type;
2054          when text_quit =>
2055            display : text.display_type;
2056          when string_input =>
2057            input  : string (1..text.max_input_length);
2058            length : natural;
2059          when input_cancel =>
2060            null;
2061          when browser_quit =>
2062            null;
2063        end case;
2064      end record;
2065
```

```
2066    procedure initialize (main_commands : in cmd_lists.cmd_list_type);
2067    --
2068    -- Synopsis:  This procedure displays the browser application's main
2069    -- window and the specified command list.  If the main window is already
2070    -- displayed, this procedure has no effect.
2071    -- If the specified command list has not been defined or has been deleted,
2072    -- cmd_lists.cmd_list_undefined is raised.
2073
2074    procedure browse (event : out event_info);
2075    --
2076    -- Synopsis:  This procedure activates the browser, thereby allowing the
2077    -- user to interact with the display.  Ideally, the application would
2078    -- define actions procedures (callbacks) for all display objects, which
2079    -- would be automatically invoked by the browser in response to user
2080    -- actions.  In this case, the browse procedure would never exit.  This
2081    -- is the preferred style of interaction, since it avoids interference
2082    -- with the window system.  If, however, it is desired that the browser
2083    -- return control to the application in response to certain user actions,
2084    -- the application need only refrain from defining action procedures for
2085    -- those actions.  If the user performs some action for which there is
2086    -- no action procedure defined, the browse procedure exits and returns
2087    -- an event indicating the nature of the user action.
2088
2089    procedure quit;
2090    --
2091    -- Synopsis:  This procedure erases and destroys all windows created
2092    -- by the browser application and terminates its connection with the
2093    -- underlying window system.  If this procedure is called from within a
2094    -- callback procedure, the browse procedure will exit upon completion of
2095    -- the callback and will return a browser_quit event.
2096
2097    ---
2098 end browser;
```

B Appendix: User Interface

In order to promote a common "look and feel" across all browser tools, it was necessary to somewhat constrain the user interface implemented by the Reusable Graphical Browser. Yet, in order to promote reuse, the user interface had to be made general enough to support a wide variety of browsing applications. These apparently conflicting goals have been reconciled by implementing a generic user interface that is tailorable for specific browser tools. This appendix describes the nature of that generic user interface and discusses the ways in which it can be tailored.

B.1 Model of User Interaction

As evidenced by its name, the Reusable Graphical Browser provides a primarily graphical user interface. This is not to say that the user interface is exclusively graphical. Provisions are made for text-based interaction as well, where appropriate.

In general, the user is presented with a graphical display of the objects within the OMS, the relationships between them, and the operations that may be performed on them. The user then interacts with the graphical display via a pointing device (e.g., a mouse). The user may use the pointing device to scan over the objects and relationships, select from among them or select an operation. When an object or relationship is selected, information about it may be displayed either graphically or textually. Similarly, the user may enter information concerning an object or relationship either graphically (via a pointing device) or textually (via a keyboard).

In the current implementation of the Reusable Graphical Browser, input and output are performed under the control of the X Window System. The Reusable Graphical Browser uses the X Toolkit to create a viewport widget (a scrollable window) in which to display the objects and relationships, to create widgets representing individual objects and relationships, to create command and menu widgets listing available operations, and to create text widgets and dialog boxes for text-based interactions. Mouse clicks on the viewport widget's scrollbars are handled internally by the X Toolkit, which scrolls the window in the selected direction. Mouse clicks on the object, relationship, command or menu widgets are reported to the Reusable Graphical Browser, which in turn dispatches them to predefined application callback procedures. Similarly, text entered into a dialog box is dispatched to a predefined application callback procedure. The application callback procedures may use facilities provided by the Reusable Graphical Browser to switch views or to display additional widgets (e.g., menus, dialog boxes or text) which the user may then interact with.

If a mouse click occurs which would normally be dispatched to an application callback procedure but the application has not defined such a procedure, the X Toolkit main loop is exited. The user's selection is then reported to the application via a return from the browser_instance.browse procedure instead of via a callback. In this case, the application must issue another call to the browser_instance.browser procedure before any more user inputs can be processed by the X Toolkit. Before issuing this call, the application may (if

so desired) use facilities provided by the Reusable Graphical Browser to switch views or to display additional widgets.

B.1.1 Output

Figure 2 illustrates the general screen layout supported by the Reusable Graphical Browser. Multiple instances of screens can be displayed simultaneously. Figure 2 demonstrates three instances of screens. Nevertheless, only one screen instance is allowed for each view. For example, the three screen instances in figure 2 represent three different views. A screen layout consists of the following elements:

- a single scrollable window, with both vertical and horizontal scrollbars, for displaying a view of the objects and relationships within an OMS;
- a label showing the title of the currently displayed view;
- a row of command buttons for commands (non-modal operations) associated with the view;
- menus of subcommands for individual commands;
- icons and associated labels (optional) and attributes (optional), depicting individual objects within the OMS;
- directed line segments connecting these object depictions, themselves marked with icons and associated labels (optional) and attributes (optional), depicting relationships between the objects;
- pop-up menus of modal operations that may be associated with individual objects or relationships or with individual views;
- scrollable pop-up text windows, with quit buttons, for displaying arbitrary text (e.g., attributes of objects or relationships);
- pop-up text windows, with confirmation buttons, for displaying alert messages;
- dialog boxes, with confirmation buttons, for inputting arbitrary text strings.
- optional topology display window, with scrollbars (if necessary), for navigating over the view.

For the purposes of this illustration, solid lines are used to indicate elements that are more or less permanent, whereas dashed lines are used to indicate elements that are transient. None of these elements would actually have dashed borders when they appear on the screen, however.

In addition to these screen elements, the *twm* window manager decorates the main application window and the pop-up text windows with title bars. The user can interact with these title bars to raise, lower, move, iconify or resize the windows. For further information on such interactions, please refer to the documentation provided with *twm* on the X window system distribution tape.

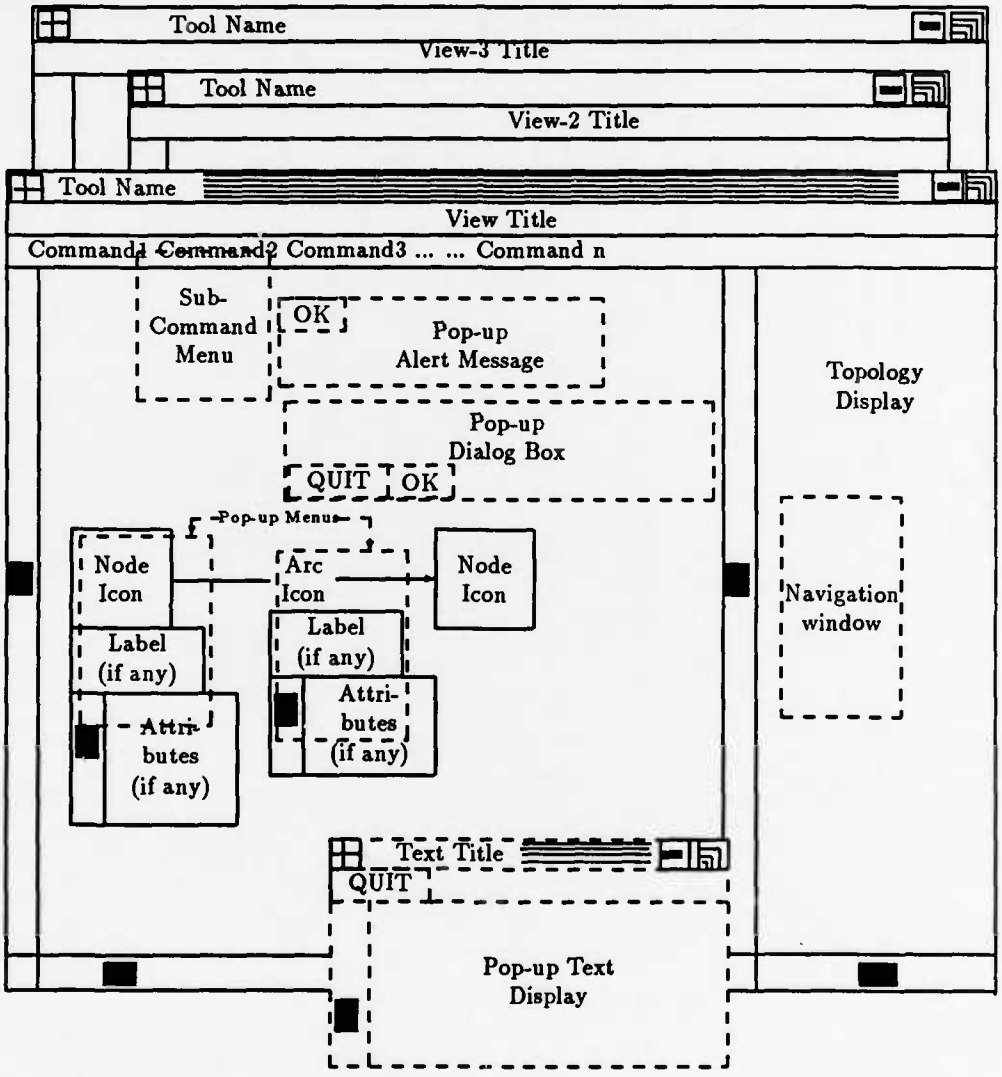

Figure 2: Browser Screen Layout

B.1.2 Input

The user supplies input to a browser tool by using the mouse to select various items appearing on the screen. To select an item, the user positions the mouse so that the pointer (a cursor in the shape of an arrow) is over the item and then clicks (depresses and releases) a mouse

button. Not all items appearing on the screen are selectable, however, nor are they all sensitive to the same mouse buttons. Of the screen elements listed above, only the following are selectable:

- vertical scrollbars;
- horizontal scrollbars;
- command buttons;
- menu items;
- icons depicting objects and relationships;
- quit buttons on text displays;
- confirmation buttons on alert messages;
- confirmation buttons on dialog boxes;
- topology display.

Vertical scrollbars are selected with either the left, right or middle mouse button. Clicking the left mouse button scrolls the window down. Clicking the right mouse button scrolls the window up. The amount by which the window scrolls for each mouse click depends on the position of the pointer at the time the mouse button is released: the closer to the top of the scrollbar, the smaller the increment; the closer to the bottom of the scrollbar, the larger the increment. Clicking the middle mouse button shifts the window directly to the position indicated by the pointer. In this case, the vertical position of the pointer relative to the scrollbar is interpreted as the desired position of the top of the window relative to the image over which it is scrolling. If the middle mouse button is depressed and held, the window position tracks the pointer position as the pointer is moved vertically (provided that it is moved only very slowly), giving the impression of smooth-scrolling. Note that the current position and extent of the window, with respect to the image over which it is scrolling, is indicated by the dark column within the scrollbar.

Horizontal scrollbars are selected in much the same manner as vertical scrollbars, except that they are oriented from left to right rather than from top to bottom. Clicking the left mouse button scrolls the window to the left. Clicking the right mouse button scrolls the window to the right. Just as for vertical scrollbars, the amount by which the window scrolls for each mouse click depends on the position of the pointer at the time the mouse button is released: the closer to the left of the scrollbar, the smaller the increment; the closer to the right of the scrollbar, the larger the increment. Clicking the middle mouse button shifts the window directly to the position indicated by the pointer: the horizontal position of the pointer relative to the scrollbar is interpreted as the desired position of the left of the window relative to the image over which it is scrolling. If the middle mouse button is depressed and held, the window position tracks the pointer position as the pointer is moved horizontally

(provided that it is moved only very slowly), giving the impression of smooth-scrolling. As with vertical scrollbars, the current position and extent of the window, with respect to the image over which it is scrolling, is indicated by the dark column within the scrollbar.

Command buttons are selected by clicking the left mouse button. While the pointer is positioned over a command button, the command button is displayed in reverse video (i.e., the black parts become white and the white parts become black). If the mouse button is then clicked, the command button returns to normal video and any subcommands associated with the selected command are displayed in a menu. If there are no subcommands associated with the selected command, no menu is displayed. In that event, the Reusable Graphical Browser immediately invokes a predefined application callback procedure to process the command. If no callback procedure is defined for the view's command buttons, the user's selection is reported to the application via a return from the browser_instance.browse procedure instead.

Individual items appearing in a menu are also selected by clicking the left mouse button. Whenever a menu is displayed, the user is forced to make a selection from it; all other selections are disabled. Positioning the pointer over an item in the menu causes the item's border to be highlighted. If the mouse button is then clicked, the menu is erased from the display. In that event, the Reusable Graphical Browser immediately invokes a predefined application callback procedure to process the selection. If no callback procedure is defined for the menu, the user's selection is reported to the application via a return from the browser_instance.browse procedure instead. Alternatively, the mouse button may be clicked while the pointer is positioned over the menu title (the top line of the menu). In that event, the menu is erased, but no selection is reported to the application (i.e., the menu is cancelled).

Icons depicting objects and relationships are selected in exactly the same manner as command buttons. While the pointer is positioned over one of these icons, the icon is displayed in reverse video (i.e., the black parts become white and the white parts become black). If the left mouse button is then clicked, the icon returns to normal video and the Reusable Graphical Browser immediately invokes a predefined application callback procedure to process the selection. If no callback procedure is defined for the corresponding object or relationship, the user's selection is reported to the application via a return from the browser_instance.browse procedure instead.

Quit buttons on text displays (labelled "QUIT") are also selected in exactly the same manner as command buttons. The effects of that selection are somewhat different, however. While the pointer is positioned over a quit button, the button is displayed in reverse video (i.e., the black parts become white and the white parts become black). If the left mouse button is then clicked, the button returns to normal video and the text window is erased. The application is not notified of the selection.

Confirmation buttons on alert messages (labelled "OK") are selected in the same manner as quit buttons on text displays, and have similar effects. One difference, though, is that the user is forced to confirm an alert message immediately; all selections other than the confirmation button are disabled until the confirmation button is selected. While the pointer

is positioned over one of these confirmation buttons, the button is displayed in reverse video (i.e., the black parts become white and the white parts become black). If the left mouse button is then clicked, the button returns to normal video and the alert message is erased. The application is not notified of the selection.

Confirmation buttons on dialog boxes are selected in the same manner as confirmation buttons on alert messages, although their effects are somewhat different. There are two such buttons, labelled "QUIT" and "OK". The purpose of the "QUIT" button is to cancel the dialog altogether; the purpose of the "OK" button is to confirm that the user's keyboard input is ready to be reported to the application. Like the confirmation buttons on alert messages, the user is forced to select a confirmation button before any other selections can be made. While the pointer is positioned over one of these confirmation buttons, the button is displayed in reverse video (i.e., the black parts become white and the white parts become black). If the left mouse button is then clicked, the button returns to normal video and the dialog box is erased. If the "QUIT" button is selected the application is not notified of the selection. If the "OK" button is selected, however, the application is notified of the selection. In that event, the Reusable Graphical Browser immediately invokes a predefined application callback procedure to process the user's keyboard input. If no callback procedure is defined for the menu, the user's keyboard input is reported to the application via a return from the browser_instance.browse procedure instead.

The topology display can be selected by clicking the left mouse button anywhere in the topology display. The application is not notified of the selection. The effect of the selection is that the navigation rectangle centers around the selected point. Also, the main display area is updated, showing the portion of the view that is currently covered by the navigation rectangle. View navigation using the topology display can also be done using scrollbars in the topology display.

B.2 Tailoring The User Interface

There are two kinds of tailoring that can be performed with respect to the user interface: tailoring of contents and tailoring of presentation style. The latter is not supported by the Reusable Graphical Browser, per se, but rather by the underlying window system. Different window systems may support this kind of tailoring to different degrees. Tailoring of contents, on the other hand, is supported directly by the Reusable Graphical Browser, and is intended as the primary mechanism for tailoring the user interface to a particular application.

B.2.1 Contents

The features of the user interface that may be tailored using facilities provided directly by the Reusable Graphical Browser are the following:

- the title displayed for a view;

- the commands (and subcommands) displayed for a view;
- the effects of selecting a particular command (or subcommand);
- the particular objects and relationships displayed in a viewport;
- the positions of objects and relationships displayed in a viewport;
- the labels and attributes (if any) displayed for individual objects and relationships;
- the effects of selecting a particular object or relationship;
- the items in each pop-up menu;
- the object or relationship (if any) with which a pop-up menu is to be associated;
- the effects of selecting a particular item from a pop-up menu;
- the text displayed in a pop-up text window;
- the text of an alert message;
- the prompt string and input constraints for a dialog box.

The title displayed for a view is the title that is specified when the view is created. To tailor this feature, simply specify the desired title when creating the view.

The commands (and subcommands) displayed for a view are also determined when the view is created. In particular, the commands (and subcommands) are those in the command list specified for the view when the view is created. To tailor the commands (and subcommands), create the desired command list and specify it when creating the view.

The effects of selecting a particular command (or subcommand) are determined by the actions of the application callback procedure invoked by the Reusable Graphical Browser when the command (or subcommand) is selected. To tailor the effects of selecting a command (or subcommand), install the desired callback procedure for the command list before displaying a view with which the command list is associated. Note that only one callback procedure is defined for each command list; so if different effects are desired for different commands (or subcommands) in the same command list, the callback procedure must take into account which command (or subcommand) was selected.

The particular objects and relationships displayed in a viewport are determined when a view is displayed. At that time, any objects and relationships in the view that have not been suppressed are displayed in the viewport. To tailor which objects and relationships are displayed, first create the desired view by inserting and/or removing nodes and arcs. The view must then be laid out before it can be displayed. By default, all nodes and arcs in the view will be displayed when the view is displayed. If desired, however, individual nodes and arcs may be temporarily eliminated from the display by suppressing them before displaying or redisplaying the view. They may later be made to reappear by unsuppressing them and

then redisplaying the view. For convenience, a view filtering utility may be applied to the view to suppress and unsuppress some combination of nodes and arcs all at once.

The positions of objects and relationships displayed in the viewport are determined when the view is laid out. This is normally accomplished by means of a layout utility that sets the positions of individual nodes and arcs in the view. Several predefined layout utilities are provided by the Reusable Graphical Browser. They are all essentially based on the same algorithm, which is presented in reference [5]. Some of them also make use of a topological sort algorithm, which is presented in reference [2], in order to break cycles. An application is not required to use one of the predefined layout utilities; it may use some layout utility of its own, if desired. To tailor the positions of objects and relationships displayed in the viewport, simply invoke the desired layout utility to process the view. Note that it is possible to re-layout a view that has already been layed out, but the new layout does not take effect until the view is re-displayed.

The labels displayed for individual objects and relationships are determined as the view is being constructed. Note that there are several ways to construct a view: by using a utility to automatically select objects (nodes) and relationships (arcs) from an application-defined graph and insert them into the view; by using a utility to automatically copy objects and relationships from another view; or by directly inserting objects (nodes) and relationships (arcs) from an application-defined graph into the view one-at-a-time. Using the first method, the application must supply functions that determine the labels for each object and relationship. Using the second method, the labels are inherited from the source view. Using the third method, the labels must be explicitly specified as each object and relationship is inserted into the view. Consequently, the mechanism for tailoring the labels depends on which method is used to construct the view. Regardless of which method is used, specifying a null string for the label causes no label to be displayed whatsoever.

The attributes displayed for individual objects and relationships are also determined as the view is being constructed. As each node or arc is inserted into the view, the Reusable Graphical Browser invokes an application-defined function to obtain an attributes string for that node or arc. If the function returns a null string, no attributes whatsoever are displayed for the specified node or arc. To tailor the attributes, instantiate the browser with functions that return the appropriate attributes strings for each node and arc.

The effects of selecting a particular object or relationship are determined by the actions of the application callback procedure invoked by the Reusable Graphical Browser when the object or relationship is selected. One callback procedure may be defined for each object, and one for each relationship. To tailor the effects of selecting a particular object or relationship, install the desired callback procedure when constructing the view. Depending on the method used to construct the view, the callback procedure is either determined by an application-supplied function, inherited from a source view or specified explicitly as the object or relationship is inserted into the view. Note that once a callback procedure has been defined for a particular object or relationship it may be superseded by a different callback procedure, but the new callback procedure does not take effect until the view is re-displayed.

The items displayed in each pop-up menu are those specified by the application when the menu is constructed. To construct a menu, the application first creates a menu with room enough for the desired number of items and then sets the individual items as desired. Once a menu has been constructed, the individual items may be changed; the changes will be apparent the next time the menu is displayed. To tailor the pop-up menus, simply construct them and/or modify them as desired.

The object or relationship (if any) with which a pop-up menu is to be associated is determined by which procedure is used to display the pop-up menu. The Reusable Graphical Browser provides three such procedures: one associates the pop-up menu with a specified object (node); another associates the pop-up menu with a specified relationship (arc); the third does not associate the pop-up menu with any object or relationship at all.

The effects of selecting a particular item from a pop-up menu are determined by the actions of the application callback procedure invoked by the Reusable Graphical Browser when the item is selected. The same callback procedure is invoked for all items in the same menu, since there is only one callback procedure per menu. To tailor the effects of selecting an item from a pop-up menu, install the desired callback procedure for the menu before displaying the menu. Note that if different effects are desired for different items in the menu, the callback procedure must take into account which item was selected.

The text displayed in a pop-up text window is determined by the source that is specified when calling the procedure to display the text. The source may be either a text file or a text buffer. If a text file is specified, the contents of the file are displayed. If a text buffer is specified, the contents of the buffer are displayed. To tailor the text displayed in a pop-up text window, create a text file or text buffer containing the desired text and then specify it as the source for the display call.

The text of an alert message is specified explicitly when calling the procedure that displays the alert message. It is passed to that procedure as a string parameter. To tailor this text, simply pass the desired string when calling the display procedure.

The prompt string and the input constraints for a dialog box are specified explicitly when calling the procedure that displays the dialog box. They are passed as parameters to that procedure. To tailor them, simply pass the desired values when calling the display procedure.

B.2.2 Presentation Style

Tailoring of presentation style encompasses such things as changing icons, fonts, border widths, border patterns, colors (if a color display is used) and sensitivity (to user inputs) of individual widgets or groups of widgets. This kind of tailoring is accomplished by modifying the X Toolkit resource file used by the application. For applications that use the Reusable Graphical Browser, this file is named "Browser" and its directory path is indicated by the environment variable $XAPPLRESDIR. An example of such a file is provided in the source directory for the sample application that is distributed with the Reusable Graphical Browser.

The syntax of the X Toolkit resource file is described in detail in references [3] and [4]. Basically, it is a simple text file, where each line of text specifies a value for a particular widget resource. It may also contain comment lines, which are designated by a sharp sign (#) at the beginning of the line. Each resource specification line consists of either a widget instance name or a widget class name, followed by the resource name, followed by a colon (:), followed by the resource value. Widget instance names or widget class names specify a path through the widget instance hierarchy rooted at the the application's top-level widget. The names of individual widget instances or widget instance classes in the path are separated using dot (.) notation. Alternatively, asterisks (*) may be substituted as wildcard separators that match any number of intervening widget instances in the path. Resource names, data types and default values are listed in the documentation for each individual widget class.

The following table listing shows all the widget names and classes used in the Reusable Graphical Browser.

```
1  Widget names and classes of the RGB widget hierarchy :
2  ------------------------------------------------------
3
4  Name                                                                  Class
5  --------------------                                                  ------
6  bshell                                                                application
7                                                                        shell
8  bshell*bshell_bb                                                      bulletin board
9  bshell*bshell_bb*vp                                                   viewport
10
11  bshell*bshell_bb*vp*arc_<arc_kind>                                    command
12  bshell*bshell_bb*vp*arc_attrib_<arc_kind>                             ascii string
13  bshell*bshell_bb*vp*arc_label_<arc_kind>                              label
14
15  bshell*bshell_bb*vp*node_<node_kind>                                  command
16  bshell*bshell_bb*vp*node_attrib_<node_kind>                           ascii string
17  bshell*bshell_bb*vp*node_label_<node_kind>                            label
18
19  bshell*bshell_main_bb                                                 bulletin board
20  bshell*bshell_main_bb*popup_alert_text_psh                            transient shell
21  bshell*bshell_main_bb*popup_alert_text_bb                             bulletin board
22  bshell*bshell_main_bb*popup_alert_text_bb*popup_alert_text_quit       command
23  bshell*bshell_main_bb*popup_alert_text_bb*popup_alert_text            ascii string
24
25  bshell*bshell_main_bb*poup_dialog_psh                                 transient shell
26  bshell*bshell_main_bb*poup_dialog_psh*popup_dialog                    dialog
27
28  *popup_str_text_sh                                                    top level shell
29  *popup_str_text_sh*popup_str_text_bb                                  bulletin board
30  *popup_str_text_sh*popup_str_text_bb*popup_str_text                   ascii string
```

```
*popup_str_text_sh*popup_str_text_bb*popup_str_text_quit        command

*popup_file_text_sh                                             top level shell
*popup_file_text_sh*popup_file_text_bb                          bulletin board
*popup_file_text_sh*popup_file_text_bb*popup_file_text          ascii string
*popup_file_text_sh*popup_file_text_bb*popup_file_text_quit     command

*menu_shell                                                     transient shell
*menu_shell*menu_bb                                             bulletin board
*menu_shell*menu_bb*menu_label                                  label
*menu_shell*menu_bb*<menu_item_name>                            command

*bb_menu_shell                                                  transient shell
*bb_menu_shell*bb_menu_bb                                       bulletin board
*bb_menu_shell*bb_menu_bb*bb_menu_label                         label
*bb_menu_shell*bb_menu_bb*<menu_item_name>                      command

*node_menu_shell                                                transient shell
*node_menu_shell*node_menu_bb                                   bulletin board
*node_menu_shell*node_menu_bb*node_menu_label                   label
*node_menu_shell*node_menu_bb*<menu_item_name>                  command

*arc_menu_shell                                                 transient shell
*arc_menu_shell*arc_menu_bb                                     bulletin board
*arc_menu_shell*arc_menu_bb*arc_menu_label                      label
*arc_menu_shell*arc_menu_bb*<menu_item_name>                    command

*outdated_flag_down                                             label
*outdated_flag_up                                               label
```

The following file listing is from the Browser file, which is the resource file for the sample application distributed with the Reusable Graphical Browser.

```
1  ##############################################################################
2  # Resource file for the Reusable Graphical Browser sample application
3  ##############################################################################
4  # RGB resource file name = Browser
5  # RGB resource directory path = $XAPPLRESDIR
6  #
7  # bitmap file path (must be set to directory containing bitmap files)
8  #-----------------
9  *bitmapFilePath: /usr/zoo/chen/stars/code/application/bitmaps
10
11 # initial size of the shell
12 bshell.width: 350
```

```
13  bshell.height: 450
14
15  # initial size of the view port
16  #----------------------------
17  bshell*bshell_bb*vp.width: 300
18  bshell*bshell_bb*vp.height: 400
19
20  # initial size of the shell's bulletin board
21  #-----------------------------------------
22  ##bshell*bshell_bb.width: 700
23  ##bshell*bshell_bb.height: 500
24  bshell*bshell_bb.width: 800
25  bshell*bshell_bb.height: 600
26
27  # fix up the arcs
28  #---------------------
29  ##bshell*bshell_bb*vp*arc_STRUCTURAL.borderWidth: 0 -- perm set to 0 in code
30  bshell*bshell_bb*vp*arc_STRUCTURAL.internalHeight: 0
31  bshell*bshell_bb*vp*arc_STRUCTURAL.internalWidth: 0
32
33  # show the arcs using bitmaps
34  #-----------------------------
35  ##bshell*bshell_bb*vp*arc_STRUCTURAL.sensitive: false
36  ##bshell*bshell_bb*vp*arc_STRUCTURAL.bitmap: point.xbm
37  bshell*bshell_bb*vp*arc_STRUCTURAL.bitmap: smalltriangle.xbm
38
39  # set dimensions of the arcs' atribute widgets
40  #-------------------------------------------------------------
41  bshell*bshell_bb*vp*arc_attrib_STRUCTURAL.height: 30
42  bshell*bshell_bb*vp*arc_attrib_STRUCTURAL.width: 150
43
44  # set dimensions of the arcs' label widgets
45  #-------------------------------------------
46  ##bshell*bshell_bb*vp*arc_label_STRUCTURAL.height:20
47  ##bshell*bshell_bb*vp*arc_label_STRUCTURAL.height:40
48  bshell*bshell_bb*vp*arc_label_STRUCTURAL.border_width: 1
49
50  # show the nodes using bitmaps
51  #--------------------------
52  bshell*bshell_bb*vp*node_FILE.bitmap: bigsquare.xbm
53  bshell*bshell_bb*vp*node_DIRECTORY.bitmap: bigcircle.xbm
54  bshell*bshell_bb*vp*node_OTHER.bitmap: bigquestion.xbm
55
56  # set dimensions of the nodes
57  #--------------------------
```

```
58  bshell*bshell_bb*vp*node_FILE.borderWidth: 1
59  bshell*bshell_bb*vp*node_FILE.internalWidth: 0
60  bshell*bshell_bb*vp*node_FILE.internalHeight: 0
61  bshell*bshell_bb*vp*node_DIRECTORY.borderWidth: 1
62  bshell*bshell_bb*vp*node_DIRECTORY.internalWidth: 0
63  bshell*bshell_bb*vp*node_DIRECTORY.internalHeight: 0
64  bshell*bshell_bb*vp*node_OTHER.borderWidth: 1
65  bshell*bshell_bb*vp*node_OTHER.internalWidth: 0
66  bshell*bshell_bb*vp*node_OTHER.internalHeight: 0
67
68  # set dimensions of the nodes' attributes widgets
69  #-----------------------------------------------------
70  bshell*bshell_bb*vp*node_attrib_FILE.height: 50
71  bshell*bshell_bb*vp*node_attrib_FILE.width: 100
72  bshell*bshell_bb*vp*node_attrib_DIRECTORY.height: 50
73  bshell*bshell_bb*vp*node_attrib_DIRECTORY.width: 100
74
75  # set dimensions of the nodes' label widgets
76  #---------------------------------------
77  bshell*bshell_bb*vp*node_label_FILE.borderWidth: 0
78  bshell*bshell_bb*vp*node_label_DIRECTORY.borderWidth: 0
79
80  # set dimensions of the popup text window
81  #-------------------------------------
82  *popup_str_text_sh.width: 150
83  *popup_str_text_sh.height: 85
84  *popup_str_text.width: 150
85  *popup_str_text.height: 85
86  #
87  *popup_file_text_sh.width: 150
88  *popup_file_text_sh.height: 85
89  *popup_file_text.width: 150
90  *popup_file_text.height: 85
91  #
92  *popup_alert_text_psh.width: 300
93  *popup_alert_text_psh.height: 50
94  *popup_alert_text.width: 300
95  *popup_alert_text.height: 50
96
97  # set dimensions of the popup dialog window
98  #---------------------------------------
99  *popup_dialog_psh.width: 600
100  *popup_dialog_psh.height: 90
101  *popup_dialog.width: 600
102  *popup_dialog.height: 90
```

```
103
104  # customize menu labels
105  #---------------------
106  *menu_label.background: black
107  *menu_label.sensitive: false
108  *bb_menu_label.background: black
109  *bb_menu_label.sensitive: false
110  *node_menu_label.background: black
111  *arc_menu_label.background: black
112  *menu_label.foreground: white
113  *bb_menu_label.foreground: white
114  *node_menu_label.foreground: white
115  *arc_menu_label.foreground: white
116
117  # customize outdated flag icons
118  #-----------------------------
119  *outdated_flag_down.bitmap: od_flag_down.xbm
120  *outdated_flag_up.bitmap: od_flag_up.xbm
```

C Appendix: Limitations

In keeping with the goals of reuse, every effort has been made to assure that the Reusable Graphical Browser is as flexible and portable as possible. Nevertheless, it still has its limitations. These are listed below.

C.1 Capabilities Not Yet Implemented

Since the Reusable Graphical Browser is not yet complete, there are a number of limitations arising from unimplemented or partially-implemented capabilities. The current plan is to eventually eliminate these limitations:

- Mechanisms for synchronization of multiple views (e.g., user-selectable refresh) are not yet supported.
- Interactive editing capabilities are not yet fully supported – although interactive deletion of objects and relationships is supported.
- Incremental re-layout and re-display of a view is not yet supported.

C.2 Limitations On Existing Capabilities

Some of the capabilities that have already been implemented also have limitations on their use:

- Before any menus can be displayed, a view must first be displayed.
- Before a view can be displayed, it must be laid out; all of its nodes and arcs must be assigned coordinates, and its orientation must be set.
- View layouts are constrained to a coordinate space having dimensions 32767 x 32767; the coordinates (0,0) are reserved to indicate un-laid-out nodes and arcs.
- Dialog box (text) input is limited to a single line of no more than 100 characters.

C.3 Potential Problems

The following problems are known to exist in this version of the Reusable Graphical Browser:

- A CONSTRAINT_ERROR exception may occur during display of a view (particularly if the view is large or if vertical layout orientation is used). This is due to node and arc coordinates exceeding the pixel address range supported by the X window system. Possible workarounds include the following:

- use horizontal layout instead of vertical layout.
- specify smaller x_pad and/or y_pad values when calling the layout procedure.
- use smaller fonts and/or icons for displaying the nodes and arcs (these can be changed via the resource file, without recompiling the application).
- construct views having fewer nodes and/or arcs to begin with.

There is no fix for this problem. It is inherent in the implementation of the X window system and the X Toolkit.

- Arc depictions (icons, labels, etc.) may overlap on the screen. This is caused by current limitations in the layout algorithm. A workaround is to change the sizes of the node and/or arc widget depictions (icons, labels, etc.), via the resource file, such that the arc depictions take up less space on the screen than the depictions of their destination nodes depictions. An alternative workaround is to use larger x_pad and/or y_pad values when calling the layout procedure.

- A view's pop-up and text windows are not necessarily erased when the view is erased. Similarly, they are not necessarily raised, lowered or iconified along with the view's main window. This problem is currently being investigated.

- I/O Error (Broken Pipe) may occur when resizing a viewport to a larger size. This problem seems to occur most often when the view contains many nodes and arcs and their attributes are integrated with the display. This problem is also being investigated. It is most likely an Xlib or X Server problem, since it is caused by loss of the socket connection between the application and the X server.

- The scrollbars are missing from the attribute (text) displays that are integrated with the screen layout. It was necessary to remove them in order to work around a bug in the X Toolkit.

- If the cursor is positioned too close to the edge of the screen when a pop-up dialog box is displayed, the dialog box's buttons may end up being displayed off screen. Since the user is forced to select one of these buttons before proceeding, the browser is effectively hung.

- Limitation: before any menus may be displayed, a view must first be displayed. If this limitation is violated, a CONSTRAINT_ERROR is raised. Application developers that wish to display menus before displaying a view may work around this limitation by first creating and displaying an empty view.

C.4 Compiler Dependencies

Although it was intended that the code for the Reusable Graphical Browser be compiler-independent, certain compiler dependencies were unavoidable. For one, the naming conventions used for the source files are compiler-dependent. For another, certain implementation-dependent pragmas (e.g., pragma interface) were used in constructing the Ada bindings to

the X library routines (Xlib). Consequently, the source files may have to be renamed and/or modified slightly in order to compile with a different compilation system.

Furthermore, Tools constructed from the Reusable Graphical Browser may not function properly if compiled by an Ada compiler whose parameter passing conventions are incompatible with those of the C compiler used to compile the file "call_ada.c". This is because a C language routine (whose source is in call_ada.c) is used to implement the callback mechanism described in Section 2. For example, such incompatibilities have been observed in the Sun-3 UNIX environment between the Alsys Ada compiler and the Sun C compiler.

C.5 X Toolkit Version Dependencies

The Reusable Graphical Browser has a version dependency on the X Toolkit. It uses Version 2.4 of the *MIT X Toolkit: Ada Language Interface* developed by UNISYS. It may not be fully compatible with later releases of that product, depending on the degree to which those later releases differ from Version 2.4. Furthermore, since the Toolkit is dependent on Version 11, Release 3 of the X window system (X11R3), there is an implied dependence of the Reusable Graphical Browser on X11R3 as well.

D Appendix: Acronyms

The following acronyms are used in this manual:

BSD Berkeley System Distribution

OMS Object Management System

VADS Verdix Ada Development System

X11R3 X Window System, Version 11, Release 3

References

[1] Booch, Grady, *Software Components With Ada*, Benjamin/Cummings, Menlo Park, CA., 1987.

[2] Knuth, Donald E., *The Art of Computer Programming: Fundamental Algorithms*, Second ed., Addison-Wesley, Reading, MA., 1975.

[3] Nye, Adrian and Tim O'Reilly, *The X Window System Series, Vol. 4: X Toolkit Intrinsics Programming Manual*, O'Reilly & Associates, Inc., Sebastapol, CA., January 1990.

[4] O'Reilly, Tim, ed., *The X Window System Series, Vol. 5: X Toolkit Intrinsics Reference Manual*, O'Reilly & Associates, Inc., Sebastapol, CA., January 1990.

[5] Robins, Gabriel, "The ISI Grapher: A Portable Tool For Displaying Graphs Pictorially," reprinted from *Proceedings of Symboliikka '87*, USC/Information Sciences Institute, Marina del Rey, CA., September 1987.