**UNCLASSIFIED**

**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

**THIS PAGE IS UNCLASSIFIED**
UR20 — Process Environment Integration
Reusable Graphical Browser Version 0.3.2

STARS—RC—01080/002/00

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

Distribution limited to
U.S. Government and U.S. Government Contractors only:
Administrative (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: Teri F. Payton, System Architect

Approved by: Hans W. Polzer, Program Manager
Contents

1 Introduction .................................................. 1
   1.1 Purpose ................................................. 1
   1.2 Scope ................................................. 1
   1.3 Organization ........................................... 1

2 Overview of The Reusable Browser ...................... 1
   2.1 Architecture .......................................... 2
   2.2 Operation ............................................. 5
      2.2.1 The ERA Data Model .............................. 6
      2.2.2 Graphs ........................................... 6
      2.2.3 Views ............................................ 7
      2.2.4 Node and Arc Depictions ....................... 8
      2.2.5 Layout Algorithms ............................... 9
      2.2.6 Command Lists ................................... 9
      2.2.7 Menus ........................................... 10
      2.2.8 Text Displays .................................... 10
      2.2.9 Dialog Boxes ..................................... 11
      2.2.10 Events ......................................... 11
      2.2.11 Callbacks ....................................... 12

3 Reusing The Reusable Browser ............................ 12
   3.1 Model of Reuse ....................................... 13
   3.2 Step-By-Step Instructions ......................... 15
      3.2.1 Step 1: Define Data Types for Instantiation .... 15
      3.2.2 Step 2: Define Subprograms for Instantiation .... 16
      3.2.3 Step 3: Instantiate The BROWSER Package ....... 19
      3.2.4 Step 4: Define Graphs ............................ 20
      3.2.5 Step 5: Define Views of Each Graph .............. 23
      3.2.6 Step 6: Lay Out Each View ...................... 33
      3.2.7 Step 7: Display A View .......................... 39
      3.2.8 Step 8: Allow User Interaction With The Display .... 39
      3.2.9 Step 9: Define Responses To User Selections .... 41

A Appendix: Ada Specifications .......................... 45

B Appendix: User Interface ................................. 98
   B.1 Model of User Interaction ......................... 98
      B.1.1 Output ........................................... 99
      B.1.2 Input ........................................... 100
   B.2 Tailoring The User Interface ..................... 103
      B.2.1 Contents ....................................... 103
      B.2.2 Presentation Style ................................ 106

C Appendix: Limitations .................................... 112
C.1 Capabilities Not Yet Implemented ............................................. 112
C.2 Limitations On Existing Capabilities ...................................... 112
C.3 Potential Problems .............................................................. 112
C.4 Compiler Dependencies ....................................................... 113
C.5 X Toolkit Version Dependencies ............................................ 114

D Appendix: Acronyms ............................................................... 115

List of Figures
1 Browser Tool Architecture ..................................................... 3
2 Browser Screen Layout ......................................................... 100
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 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

1UNIX 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:

```
-- Excerpt from application/browser_params.a:

subtype file_id is file_system.infoptr;  -- Identified by file system info

type link_id is -- Identified by source and destination file IDs
    record
        source : file_id;
        dest   : file_id;
    end record;


type file.kind is (file, directory, other); -- Kinds of UNIX file system entities

type link.kind is (structural); -- Kinds of UNIX file system relationships (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```
begin
  
  ----
  -- Just return nothing
  ----
  
  return "";
  
end get_attributes;

function hash ( -- Hash node ID to a bucket number
  the_node: file_id; -- Node ID
  number_of_buckets: positive -- Number of buckets
) return positive is

begin
  
  ----
  -- Return the inode value modulo the number of buckets
  ----
  
  return ( positive ( the_node.sbuf.st_ino ) mod number_of_buckets ) + 1;

end hash;

function get_attributes ( -- Return attributes of arc
  the_arc: link_id -- Arc ID
) return string is

begin
  
  ----
  -- Just return nothing
  ----
  
  return "";

end get_attributes;

function hash ( -- Hash arc ID to a bucket number
  the_arc: link_id; -- Arc ID
  number_of_buckets: positive -- Number of buckets
) return positive is

begin


return ( positive ( the_arc.dest.sbuf.st_ino ) mod number_of_buckets ) + 1;
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:

```ada
with browser;
with browser_params;
package browser_instance is new browser (
  graph_id => integer,  -- Graph ID
  designates_same_graph => standard."=",
  node_id => browser_params.file_id,  -- Node ID
  node_kind => browser_params.file_kind,  -- Node kind
  designates_same_node => browser_params."=",
  hash_of_node => browser_params.hash,  -- Hash of node ID
  arc_id => browser_params.link_id,  -- Arc ID
  arc_kind => browser_params.link_kind,  -- Arc kind
  hash_of_arc => browser_params.hash,  -- Hash of arc ID
  designates_same_arc => browser_params."="  -- Equality operator
);```

---

Page 19
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
```
begin

-- Walk through the children

curr := parent.child;
while curr /= NULL
loop

-- Create the child node

browser_instance.graphs.add_node ( 
  the_node =>   kid,
  with_id =>   curr,
  with_name =>   browser_instance.graphs.node_name ( curr.name ),
  of_kind =>   to_file_kind (curr.sbuf.st_mode.s_ifmt),
  to_graph =>   graph );

-- Add an arc between the parent and this node

browser_instance.graphs.add_arc ( 
  the_arc =>    arc,
  with_id =>    browser_params.link_id'(parent, curr),
  from_node =>   node,
  to_node =>    kid,
  to_graph =>    graph );

arc_count := arc_count - 1;

-- Recurse if this is a directory

if file_system."="/"(curr.child, NULL) then
  addanode ( graph, kid, curr );
end if;

curr := curr.next;
end loop;
function make_graph (root: in file_system.infoptr) return browser_instance.graphs.graph_type is

graph: browser_instance.graphs.graph_type; -- Graph object created
node: browser_instance.graphs.node_type; -- Node object created

begin

**** Set initial arc count for assigning unique arc IDs:

arc_count := file_system.arc_count;

****

-- Create the graph object

browser_instance.graphs.create_graph (the_graph => graph,
with_id => 1,
estimated_arcs => positive (file_system.arc_count + 1),
estimated_nodes => positive (file_system.node_count));

****

-- Now create the root node

browser_instance.graphs.add_node (the_node => node,
with_id => root,
with_name => browser_instance.graphs.node_name (root.name),
of_kind => to_file_kind(root.sbuf.st_mode.s_ifmt),
to_graph => graph);

****

-- Create the rest of the graph from this node's children

addanode (graph, node, root);

return graph;

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>.view_utilities.construct_view` procedure to construct the desired view of a graph.
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:

```plaintext
--- Excerpts from application/callbacks.b.a

procedure process_node_selection (to_process_event: in browser_instance.event_ptr);

procedure process_arc_selection (to_process_event: in browser_instance.event_ptr);

package node_selection is new browser_instance.callbacks.callback (the_procedure => process_node_selection);

package arc_selection is new browser_instance.callbacks.callback (the_procedure => process_arc_selection);

--- Excerpts from application/utilities.b.a:

function always_true (the_node: browser_instance.graphs.node_type) return boolean is
begin
  return true;
end always_true;

function file_name (
the_node: browser_instance.graphs.node.type
) return browser_instance.views.node_label is
leaf: file_system.infoptr;
begin
leaf := browser_instance.graphs.id.of ( the_node );
case leaf.sbuf.st_mode.s_ifmt is
when stat.s_unknown =>
return browser_instance.views.node_label ( leaf.name & '?' );
when stat.s_ififo =>
return browser_instance.views.node_label ( leaf.name );
when stat.s_ifchr =>
return browser_instance.views.node_label ( leaf.name );
when stat.s_ifdir =>
return browser_instance.views.node_label ( leaf.name & '/' );
when stat.s_ifblk =>
return browser_instance.views.node_label ( leaf.name );
when stat.s_ifreg =>
if leaf.sbuf.st_mode.s_iexecu or else
leaf.sbuf.st_mode.s_iexecg or else
leaf.sbuf.st_mode.s_iexeco then
return browser_instance.views.node_label ( leaf.name & '*' );
else
return browser_instance.views.node_label ( leaf.name );
end if;
when stat.s_iflink =>
return browser_instance.views.node_label ( leaf.name & '@' );
when stat.s_ifsock =>
return browser_instance.views.node_label ( leaf.name & '=' );
when others =>
return browser_instance.views.node_label ( leaf.name );
end case;
end file_name;

function no_attributes (the_node: browser_instance.graphs.node_type) return browser_instance.views.attributes is
begin
    return "";
end no_attributes;

function node_action (the_node: browser_instance.graphs.node_type) return browser_instance.callbacks.callback_type is
begin
    return callbacks.node_callback;
end node_action;

function always_true (the_arc: browser_instance.graphs.arc_type) return boolean is
begin
    return true;
end always_true;

function no_label (the_arc: browser_instance.graphs.arc_type) return browser_instance.views.arc_label is
begin
    return "";
end no_label;

function arc_attributes (the_arc: browser_instance.graphs.arc_type) return browser_instance.views.attributes is
arc_id : browser_params.link_id := browser_instance.graphs.id_of (the_arc);
begin
    return browser_instance.views.attributes("Source: " & arc_id.source.name & ascii.1f & "Dest: " & arc_id.dest.name);
end arc_attributes;

function arc_action(
procedure construct_view_of_all_files is
new browser_instance.viewUtilities.construct_view (  
  include_node => always_true,  
  label_for_node => file_name,  
  attributes_for_node => no_attributes,  
  action_for_node => node_action,  
  include_arc => always_true,  
  label_for_arc => no_label,  
  attributes_for_arc => arc_attributes,  
  action_for_arc => arc_action  
);  

procedure construct_view_of_directories is
new browser_instance.viewUtilities.construct_view (  
  include_node => is_directory,  
  label_for_node => file_name,  
  attributes_for_node => no_attributes,  
  action_for_node => node_action,  
  include_arc => is_directory,  
  label_for_arc => no_label,  
  attributes_for_arc => arc_attributes,  
  action_for_arc => arc_action  
);
Finally, the views are constructed by invoking the view construction utilities like so:

```plaintext
-- Excerpt from application/utilities_b.a:

procedure make_full_view (     -- Make a view showing all files/directories
  view : in out browser_instance.views.view_type; -- The view to make
  graph: in     browser_instance.graphs.graph_type; -- To make it from
  cmd_list: in     browser_instance.cmd_lists.cmd_list_type -- For the view
) is

  construct_view_of_all_files (the_view => view,
    of_graph => graph,
    with_cmd_list => cmd_list,
    with_title => "All Files",
    estimated_arcs => file_system.arc_count,
    estimated_nodes => file_system.node_count);

end make_full_view;

procedure make_dir_view (     -- Make a view showing directories only
  view : in out browser_instance.views.view_type; -- The view to make
  graph: in     browser_instance.graphs.graph_type; -- To make it from
  cmd_list: in     browser_instance.cmd_lists.cmd_list_type -- For the view
) is

  construct_view_of_directories (the_view => view,
    of_graph => graph,
    with_cmd_list => cmd_list,
    with_title => "Directories Only",
    estimated_arcs => file_system.arc_count,
    estimated_nodes => file_system.node_count);

end make_dir_view;

-- Excerpt from application/main.a

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

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  
20      : constant := 2;  
21    item_1  
22      : constant cmd_lists.cmd_item := "Select View";  
23    item_2  
24      : constant cmd_lists.cmd_item := "Quit";  
25  begin
26
27  -- Create the cmd_list object.  
28  cmd_lists.create_cmd_list(the_cmd_list, n_cmds);  
29  -- Set up the cmd_list callback.  
30  cmd_lists.set_action(the_cmd_list, the_callback);  
31  -- Set the commands.  
32  cmd_lists.set_cmd(in_cmd_list => the_cmd_list, the_cmd => 1,
to_value => item_1);
  cmd_lists.set_cmd(in_cmd_list => the_cmd_list, the_cmd => 2,
                   to_value => item_2);
end init_initial_cmd_list;

init_full_view_cmd_list:
declare
  the_cmd_list : cmd_lists.cmd_list_type renames full_view_cmd_list;
  the_callback : constant browser_instance.callbacks.callback_type :=
                callbacks.full_view_cmd_list_callback;
  n_cmds : constant := 3;
  item_1 : constant cmd_lists.cmd_item := "Filter View";
  item_2 : constant cmd_lists.cmd_item := "Topology";
  item_3 : constant cmd_lists.cmd_item := "Quit";
begin
  — Create the cmd_list object.
  cmd_lists.create_cmd_list(the_cmd_list, n_cmds);
  — Set up the cmd_list callback.
  cmd_lists.set_action(the_cmd_list, the_callback);
  — Set the commands.
  cmd_lists.set_cmd(in_cmd_list => the_cmd_list, the_cmd => 1,
                   to_value => item_1);
  cmd_lists.set_cmd(in_cmd_list => the_cmd_list, the_cmd => 2,
                   to_value => item_2);
  cmd_lists.set_cmd(in_cmd_list => the_cmd_list, the_cmd => 3,
                   to_value => item_3);
  — Initialize subcommand lists.
  declare
    cmd_index : constant := 1; -- Filter View
    n_subcmds : constant := 3;
    subitem_1 : constant cmd_lists.cmd_item := "Suppress Files";
    subitem_2 : constant cmd_lists.cmd_item := "Unsuppress Files";
    subitem_3 : constant cmd_lists.cmd_item := "Unsuppress All";
begin
  — Create the subcommand list.
  cmd_lists.create_subcmd_list(the_cmd_list, cmd_index, n_subcmds);
  — Set the subcommands.
  cmd_lists.set_subcmd(in_cmd_list => the_cmd_list, the_cmd => cmd_index,
                       the_subcmd => 1,
                       to_value => subitem_1);
  cmd_lists.set_subcmd(in_cmd_list => the_cmd_list, the_cmd => cmd_index,
                       the_subcmd => 2,
                       to_value => subitem_2);
  cmd_lists.set_subcmd(in_cmd_list => the_cmd_list, the_cmd => cmd_index,
                       the_subcmd => 3,
to_value => subitem_3);
end;
end init_full_view_cmd_list;

init_dir_view_cmd_list:
declare
the_cmd_list : cmd_lists.cmd_list_type renames dir_view_cmd_list;
the_callback : constant browser_instance.callbacks.callback_type
:= callbacks.dir_view_cmd_list_callback;
n_cmds : constant := 3;
item_1 : constant cmd_lists.cmd_item := "Filter View";
item_2 : constant cmd_lists.cmd_item := "Topology";
item_3 : constant cmd_lists.cmd_item := "Quit";
begin
-- Create the cmd_list object.
cmd_lists.create_cmd_list(the_cmd_list, n_cmds);
-- Set up the cmd_list callback.
cmd_lists.set_action(the_cmd_list, the_callback);
-- Set the commands.
cmd_lists.set_cmd(in_cmd_list => the_cmd_list, the_cmd => 1,
to_value => item_1);
cmd_lists.set_cmd(in_cmd_list => the_cmd_list, the_cmd => 2,
to_value => item_2);
cmd_lists.set_cmd(in_cmd_list => the_cmd_list, the_cmd => 3,
to_value => item_3);
-- Initialize subcommand lists.
declare
cmd_index : constant := 1; -- Filter View
n_subcmds : constant := 1;
subitem_1 : constant cmd_lists.cmd_item := "Unsuppress All";
begin
-- Create the subcommand list.
cmd_lists.create_subcmd_list(the_cmd_list, cmd_index, n_subcmds);
-- Set the subcommands.
cmd_lists.set_subcmd(in_cmd_list => the_cmd_list, the_cmd => cmd_index,
the_subcmd => 1,
to_value => subitem_1);
end;
end init_dir_view_cmd_list;
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.

```
procedure isi_grapher (the_view : in out views.view_type) is

-- An implementation of the original layout algorithm on which the
-- Reusable Graphical Browser's predefined layout utilities are based.
-- This is a literal implementation of the algorithm stated in a
-- research report by Gabriel Robins entitled, "The ISI Grapher:
-- A Portable Tool for Displaying Graphs Pictorially," reprinted from
-- Proceedings of Symboliikka '87, held in Helsinki, Finland, August
-- 17-18, 1987. It is available from Information Sciences Institute
-- (ISI), 4676 Admiralty Way, Marina del Rey, CA 90292-6695, as
-- Report No. ISI/RS-87-196.

-- Data Objects:

x_pad : constant := 30; -- amount of horizontal spacing between nodes
last_y : natural := 0;
orientation : constant views.orientation.type := views.horizontal;

-- Specs for procedures needed to instantiate generics:

procedure zero_y_and_continue (the_node : in views.node_type;
                               continue : out boolean);

procedure zero_x_and_continue (the_node : in views.node_type;
                               continue : out boolean);

procedure layout_a_child (the_arc : in views.arc_type;
                          continue : out boolean);
```
procedure layout_a_parent (the_arc : in views.arc_type;
            continue : out boolean);

procedure layout_y_if_root (the_node : in views.node_type;
            continue : out boolean);

procedure layout_x_if_leaf (the_node : in views.node_type;
            continue : out boolean);

procedure layout_y (the_node : in views.node_type);

procedure layout_x (the_node : in views.node_type);

-- Generic Instantiations:

procedure zero_y_for_all_nodes is
new views.iterate_nodes(visit => zero_y_and_continue);

procedure zero_x_for_all_nodes is
new views.iterate_nodes(visit => zero_x_and_continue);

procedure layout_y_for_children_of_node is
new views.iterate_arcs_from (visit => layout_a_child);

procedure layout_x_for_parents_of_node is
new views.iterate_arcs_to (visit => layout_a_parent);

procedure layout_y_for_all_roots is
new views.iterate_nodes (visit => layout_y_if_root);

procedure layout_x_for_all_leaves is
new views.iterate_nodes (visit => layout_x_if_leaf);

-- Procedure and Function Bodies:

procedure zero_y_and_continue (the_node : in views.node_type;
            continue : out boolean) is
begin
views.set_position (of_node => the_node,
            to_position => (views.get_position(the_node).x_coordinate,0));
            continue := true;
end zero_y_and_continue;

procedure zero_x_and_continue (the_node : in views.node_type;
            continue : out boolean) is
begin
    views.set_position (of_node => the_node,
    to_position => (0, views.get_position(the_node).y_coordinate));
    continue := true;
end zero_x_and_continue;

function is_a_root (the_node : views.node_type) return boolean is
begin
    return views.number_of_arcs_to(the_node) = 0;
end is_a_root;

function is_a_leaf (the_node : views.node_type) return boolean is
begin
    return views.number_of_arcs_from(the_node) = 0;
end is_a_leaf;

procedure layout_y_if_root (the_node : in views.node_type;
continue : out boolean) is
begin
    if is_a_root(the_node) then
        layout_y(the_node);
    end if;
    continue := true;
end layout_y_if_root;

procedure layout_x_if_leaf (the_node : in views.node_type;
continue : out boolean) is
begin
    if is_a_leaf(the_node) then
        layout_x(the_node);
    end if;
    continue := true;
end layout_x_if_leaf;

procedure layout_a_child (the_arc : in views.arc_type;
continue : out boolean) is
    child : views.node_type := views.destination.of (the_arc);
begin
    layout_y (the_node => child);
    continue := true;
end layout_a_child;

procedure layout_a_parent (the_arc : in views.arc_type;
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;
procedure layout_y (the_node : in views.node_type) is
begin
  if views.get_position(the_node).y_coordinate = 0 then
    if has.any.unlaid.out.children(the.node) then
      layout_y_for_children_of_node (the_node);
      views.set_position(of_node => the_node,
                         to_position => (views.get_position(the_node).x_coordinate,
                                        average_y_for_children_of_node(the_node)));
    else
      views.set_position(of_node => the.node,
                         to_position => (views.get_position(the_node).x_coordinate,
                                        last_y + views.get.dimensions(the.node,orientation).height));
      last_y := views.get_position(the_node).y_coordinate;
    end if;
  end if;
end layout_y;

function max_x_plus_width_of_parents (the_node : views.node_type)
return natural is
max : natural := 0;
temp : natural := 0;
procedure check_parent (the.arc : in views.arc_type;
                         continue : out boolean) is
parent : views.node_type := views.source.of(the.arc);
begi
  temp := views.get_position(parent).x_coordinate +
         views.get.dimensions(parent,orientation).width;
  if temp > max then
    max := temp;
  end if;
  continue := true;
end check_parent;

procedure scan_parents is
  new views.iterate_arcs_to (visit => check_parent);
begin
  scan_parents (the_node => the_node);
  return max;
end max_x_plus_width_of_parents;

function has_parents (the_node : in views.node_type) return boolean is
begin
  return views.number_of_arcs_to(the_node) /= 0;
end has_parents;
procedure layout_x (the_node : in views.node_type) is
begin
  if views.get_position(the_node).x_coordinate = 0 then
    if has_parents(the_node) then
      layout_x_for_parents_of_node (the_node);
      views.set_position(of_node => the_node,
      to_position => (max_x_plus_width_of_parents(the_node) + x_pad,
      views.get_position(the_node).y_coordinate));
    end if;
  end if;
end if;
end layout_x;

begin
  -- Pass 1: Layout all Y coordinates, beginning at each root, and
  -- traversing the graph depth-first.
  zero_y_for_all_nodes (of_the_view => the_view);
  last_y := 0;
  layout_y_for_all_roots (of_the_view => the_view);

  -- Pass 2: Layout all X coordinates, beginning at each leaf, and
  -- traversing the graph in the other direction (towards the roots)
  -- depth-first.
  zero_x_for_all_nodes (of_the_view => the_view);
  layout_x_for_all_leaves (of_the_view => the_view);

  -- Indicate orientation of layout.
  views.set.orientation (of_view => the_view,
  to_value => orientation);
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.
browser_instance.views.set_orientation (of_view => globals.init_vista,
    to_value => browser_instance.views.horizontal);

-- Lay out the full view (all files, directories, etc.).

browser_instance.layout_algorithms.cyclic_horizontal (
    the_view => globals.full_vista);

-- Lay out the directory view (directories only).

browser_instance.layout_algorithms.acyclic_vertical (
    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:

-- Excerpt from application/main.a:

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:

```plaintext
-- Excerpt from application/main.a:

loop
  browser_instance.browse ( exit_event );
  if browser_instance.views."="(exit_event.view, globals.init_vista)
    then -- event expected
      case exit_event.kind is
        when browser_instance.command_select =>
          case exit_event.command is
            when 1 =>
              browser_instance.views.display_menu ( 
                the_menu => static_menus.view_menu,
                for_view => globals.init_vista);
            when 2 =>
              exit;
            when others =>
              null;
          end case;
        when others => -- not the expected event
          text_io.put_line ("Unexpected exit from browser.");
          text_io.put_line ("Unhandled event: "
            & browser_instance.event_kind'image (exit_event.kind));
      end case;
    else -- unexpected event
      text_io.put_line ("Unexpected exit from browser.");
      text_io.put_line ("Unhandled event: "
        & browser_instance.event_kind'image (exit_event.kind));
  end if;
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
```

Page 42
procedure process_file_node_menu_selection (
  to_process_event: in browser_instance.event_ptr
); 

procedure process_dir_node_menu_selection (
  to_process_event: in browser_instance.event_ptr
); 

procedure process_other_node_menu_selection (
  to_process_event: in browser_instance.event_ptr
); 

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:

```{} -- Excerpts from application/callbacks_b.a:

procedure process_file_node_menu_selection (
  to_process_event: in browser_instance.event_ptr
) is 
    temp_display : browser_instance.text.display_type;

procedure suppress_arc (the_arc : in browser_instance.views.arc_type;
                        continue : out boolean) is 
    begin 
        browser_instance.views.set_suppression(of_arc => the.arc,
                                               to_value => true);
        continue := true;
    end suppress_arc;

procedure suppress_arcs_to_node is 
    new browser_instance.views.iterate_arcs_to (visit => suppress_arc);

procedure remove_arc (the_arc : in browser_instance.views.arc_type;
                      continue : out boolean) is 
    local_arc : browser_instance.views.arc_type := the_arc;
    begin 
        browser_instance.views.remove_arc(the_arc => local_arc,
                                           from_view => to_process_event.view);
        continue := true;
    end remove_arc;
```
procedure remove_arcs_to_node is
  new browser_instance.views.iterate_arcs_to (visit => remove_arc);
begin
  case to_process_event.n_item is
  when 1 => -- Display Attributes
    display_node_attributes (to_process_event.n_node);
  when 2 => -- Display Contents
    declare
      leaf : file_system.infoptr := browser_instance.graphs.id_of (browser_instance.views.graph_node_of (to_process_event.n_node));
    begin
      browser_instance.text.display_text (the_display => temp_display,
        from_file => file_system.full_pathname(leaf),
        use_title => "Contents of " & string(browser_instance.views.label_of( to_process_event.n_node)),
        quit_action => text_quit_callback);
    end;
  when 3 => -- Suppress File
    browser_instance.views.set_suppression(of_node => to_process_event.n_node,
      to_value => true);
    suppress_arcs_to_node (the_node => to_process_event.n_node);
    browser_instance.views.display_view (the_view => to_process_event.view);
  when 4 => -- Delete File
    remove_arcs_to_node (the_node => to_process_event.n_node);
    browser_instance.views.remove_node (the_node => to_process_event.n_node, from_view => to_process_event.view);
    browser_instance.views.display_view (the_view => to_process_event.view);
  when others =>
    null;
  end case;
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:

54 package browser
    72 type event_kind
    85 type event_info (incomplete)
    87 type event_ptr
    91 anticipated_arcs
    92 anticipated_nodes
    95 package callbacks
       104 type callback_type
       108 no_callback
       114 callback_undefined
       121 package callback
          122 function proc_id
150  package menus
    158  type menu_type
    160  type menu_title
    161  type menu_item
    162  subtype item_index
    166  no_menu
    172  menu_undefined
    174  menu_overflow
    178  no_such_item
    180  procedure create_menu
    190  procedure delete_menu
    198  procedure set_item
    211  function get_item
    221  function title_of
    228  function number_of_items_in
    235  procedure set_action
    243  function get_action

259  package cmd_lists
    270  type cmd_list_type
    272  type cmd_item
    274  subtype cmd_index
    278  no_cmd_list
    284  cmd_list_undefined
    286  cmd_list_overflow
    288  subcmds_already_exist
    290  no_such_cmd
    295  procedure create_cmd_list
    304  procedure create_subcmd_list
    321  procedure delete_cmd_list
    330  procedure delete_subcmd_list
    342  procedure set_cmd
    355  procedure set_subcmd
    371  function get_cmd
    381  function get_subcmd
    394  function number_of_cmds_in
    402  function number_of_subcmds_of
    414  procedure set_action
    423  function get_action

439  package graphs
    453  type graph_type
    455  type node_type
    456  type node_name
    458  type arc_type
459  type arc_name
460  type arc_direction
464  no_graph
467  no_node
470  no_arc
476  graph_already_exists
478  graph_not_found
479  graph_undefined
481  graph_overflow
484  node_already_exists
486  node_undefined
488  node_not_found
489  node_has_references
492  arc_already_exists
494  arc_undefined
496  arc_not_found
500  procedure create_graph
516  procedure destroy_graph
527  function is_defined (graph)
533  function id_of (graph)
541  function get_graph
548  procedure add_node
569  procedure delete_node
585  function is_defined (node)
590  function is_a_member_of (node)
600  function graph_of (node)
609  function has_references
616  function id_of (node)
623  function name_of (node)
630  function kind_of (node)
637  function get_node
650  procedure iterate_nodes
660  procedure add_arc
690  procedure delete_arc
704  function is_defined (arc)
709  function is_a_member_of (arc)
719  function graph_of (arc)
728  function id_of (arc)
735  function name_of (arc)
742  function source_of
750  function destination_of
758  function direction_of
766  function kind_of (arc)
773  function get_arc
786  procedure iterate_arcs
package views
    type view_type
    type view_title
    type node_type
    type node_label
    type arc_type
    type arc_label
    type attributes
    type coordinates_type
    type dimensions_type
    type orientation_type
    no_view
    no_node
    no_arc
    no_attributes
    view_undefined
    view_overflow
    view_is_displayed
    node_undefined
    node_not_found
    node_has_references
    arc_undefined
    arc_not_found
    not_laid_out
    procedure create_view
    procedure delete_view
    function is_defined (view)
    function graph_of
    function cmd_list_of
    function title_of
    function number_of_nodes_in
    function number_of_arcs_in
    procedure set_orientation
    function get_orientation
    function set_arrow_spacing
    function get_arrow_spacing
    procedure set_outdated_flag
    function is_outdated
    procedure set_topology_display
    function get_topology_display
    procedure display_view
    function is_displayed
    procedure search_view (node)
    procedure search_view (arc)
    procedure erase_view
1093 procedure display_menu (view)
1103 procedure display_menu (node)
1116 procedure display_menu (arc)
1132 procedure iterate_views;
1141 procedure insert_node
1165 procedure remove_node
1181 function is_defined (node)
1186 function is_a_member_of (node)
1196 function graph_node_of
1203 function is_in_view
1213 function view_node_of
1225 function label_of (node)
1233 function has_attributes (node)
1240 procedure refresh_attributes (node)
1249 procedure set_action (node)
1257 function get_action (node)
1265 procedure set_position (node)
1273 function get_position (node)
1280 function get_dimensions (node)
1290 function get_icon_dimensions (node)
1298 procedure set_suppression (node)
1307 function is_suppressed (node)
1314 function number_of_arcs_from
1321 function number_of_arcs_to
1331 procedure iterate_nodes
1343 procedure insert_arc
1368 procedure remove_arc
1382 function is_defined (arc)
1387 function is_a_member_of (arc)
1397 function graph_arc_of
1404 function is_in_view
1414 function view_arc_of
1426 function label_of (arc)
1434 function source_of
1442 function destination_of
1450 procedure redirect_arc
1470 function has_attributes (arc)
1477 procedure refresh_attributes (arc)
1486 procedure set_action (arc)
1494 function get_action (arc)
1502 procedure set_position (arc)
1510 function get_position (arc)
1517 function get_dimensions (arc)
1527 function get_icon_dimensions (arc)
1535 procedure set_suppression (arc)
1544 function is_suppressed (arc)
1554 procedure iterate_arcs
1567 procedure iterate_arcs_from
1580 procedure iterate_arcs_to

1605 package text
1612 type buffer_type
1614 type display_type
1615 type display_kind
1619 buffer_undefined
1620 buffer_overflow
1622 buffer_in_use
1624 display_undefined
1625 display_overflow
1627 wrong_display_kind
1632 no_display
1636 no_buffer
1640 max_input_length
1644 procedure create_buffer
1652 procedure delete_buffer
1663 procedure clear_buffer
1668 procedure append
1677 procedure append_line
1686 procedure new_line
1695 function length_of
1701 function contents_of
1709 procedure display_text (buffer)
1726 procedure display_text (file)
1742 procedure erase_text
1752 procedure refresh_text
1759 function kind_of
1766 function get_buffer
1775 function get_file
1786 procedure display_alert
1793 procedure prompt_for_input

1826 package view_utilities
1852 procedure construct_view
1903 procedure refresh_view
1932 procedure filter_view

1946 package layout_algorithms
1953 procedure cyclic_horizontal
1969 procedure acyclic_horizontal
1986 procedure cyclic_vertical
2003 procedure acyclic_vertical

2025 type event_info (completed)
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    -- Imported information about the instantiating application's graph objects.
21    type graph_id is private; -- uniquely identifies the instantiator's graphs
22       with function designates_same_graph (graph_a, graph_b : graph_id)
23          return boolean is "=";
24
25    -- Imported information about the instantiating application's node objects.
26    type node_id is private; -- uniquely identifies the instantiator's nodes
27       type node_kind is (<>); -- differentiates between different kinds of nodes
28       with function designates_same_node (node_a, node_b : node_id)
29          return boolean is "=";
30
31    -- Imported information about the instantiating application's arc objects.
32    type arc_id is private; -- uniquely identifies the instantiator's arcs
33       type arc_kind is (<>); -- differentiates between different kinds of arcs
34       with function designates_same_arc (arc_a, arc_b : arc_id)
35          return boolean is "=";
with function hash_of_node (the_node : node_id;
    number_of_buckets : positive) return positive;
-- maps node IDs into integers in the range 1..number_of_buckets with
-- a linear (or near-linear) distribution.

with function hash_of_arc (the_arc : arc_id;
    number_of_buckets : positive) return positive;
-- maps arc IDs into integers in the range 1..number_of_buckets with
-- a linear (or near-linear) distribution.

package browser is

-- A reusable graphical browser, capable of browsing over any object
-- management system whose objects and the relationships among them can
-- be depicted as a directed graph. It supports application-defined
-- data types for unique node and arc identifiers. It also supports
-- application-defined data types for distinguishing between different
-- kinds of nodes and arcs. It supports the association of
-- application-defined attributes with individual nodes and arcs. And
-- lastly, it supports tuning of translations between objects in the
-- application domain and those in the browser domain.

-- Browser Global Types:

-- Events are user actions that are reported to the application by the
-- reusable browser. The following type defines the kinds of events
-- that are reported.
type event_kind is (position_select,
    command_select,
    subcommand_select,
    menu_item_select,
    node_menu_item_select,
    arc_menu_item_select,
    menu_cancel,
    node_select,
    arc_select,
    text_quit,
    string_input,
    input_cancel,
    browser_quit);

type event_info (kind : event_kind
15 February 1991

86 := event_kind'first); -- describes the event in detail
87
88 type event_ptr is access event_info; -- points to an event description
89
90 -- Browser Global Constants:
91
92 anticipated_arcs : constant := 100; -- optimized for this many by default
93 anticipated_nodes : constant := 100; -- optimized for this many by default
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 -- Types:
102
103 type callback_type is private; -- a handle for callback procedures
104
105 -- Constants:
106
107 no_callback : constant callback_type; -- a value corresponding to no
108 -- callback procedure; all objects of type callback_type are initialized
109 -- to this value by default.
110
111 -- Exceptions:
112
113 callback_undefined : exception; -- an attempt was made to call an undefined
114 -- callback procedure
115
116 -- Operations:
117
118 generic
119 with procedure the_procedure (to_process_event : in event_ptr);
120 package callback is
121 function proc_id return callback_type;
122 end callback;
123
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
...PROCEDURE OR A TASK. OTHERWISE, A PROGRAM_ERROR EXCEPTION MAY BE RAISED
WHEN IT IS INVOKED VIA "CALL".

procedure call (the_proc_id : in callback_type;
to_process_event : in event_ptr);
--
-- Synopsis: This procedure is used to invoke a callback procedure.
-- It calls the specified procedure, passing it the specified event to
-- process.
-- If the specified procedure has not been defined as a callback procedure,
-- callback_undefined is raised.

private
  type callback_rep;
type callback_type is access callback_rep;
no_callback : constant callback_type := null;
end callbacks;

---

package menus is
  -- Modal menus to be displayed (popped-up) by the browser, and from which
  -- user selections are to be immediately obtained.

  -- Types:
  type menu_type is private; -- abstract type for a menu
  type menu_title is new string; -- menu title to be displayed
  type menu_item is new string; -- item in a menu
  subtype item_index is positive; -- position of an item in a menu

  -- Constants:
  no_menu : constant menu_type; -- a value for an undefined menu;
  -- all objects of type menu_type are initialized to this value by
  -- default, and are set to this value when deleted.

  -- Exceptions:
  menu_undefined : exception; -- the specified menu has not been defined
  -- or is no longer defined
  menu_overflow : exception; -- there are insufficient resources available
  -- to create or expand the menu

Page 54
no_such_item : exception; -- there is no such item in the specified menu

-- Operations:
procedure create_menu (the_menu : in out menu_type;
              with_title : in menu_title;
              number_of_items : in positive);

-- Synopsis: This procedure creates a menu with the specified number
-- of items and gives it the specified title. The menu is initially
-- empty (i.e., all of the items are blank).
-- Menu_overflow is raised if there are insufficient resources available
-- to create the menu.

procedure delete_menu (the_menu : in out menu_type);

-- Synopsis: This procedure deletes the specified menu.
-- If the specified menu is not defined to begin with,
-- menu_undefined is raised.
-- WARNING: THIS OPERATION MAY LEAVE DANGLING REFERENCES, IF THE
-- APPLICATION HAS CREATED MULTIPLE ALIASES FOR THE MENU VIA ASSIGNMENT.

procedure set_item (in_menu : in menu_type;
              the_item : in item_index;
              to_value : in menu_item);

-- Synopsis: This procedure sets the specified item in the specified
-- menu to the specified value.
-- If the specified menu has not been defined or has been deleted,
-- menu_undefined is raised.
-- If there is no such item as the specified item in the menu,
-- no_such_item is raised.
-- Menu_overflow is raised if there are insufficient resources available
-- to add the item to the menu.

function get_item (from_menu : menu_type;
              the_item : item_index) return menu_item;

-- Synopsis: This function returns the specified item from the specified
-- menu.
-- If the specified menu has not been defined or has been deleted,
-- menu_undefined is raised.
-- If there is no such item as the specified item in the menu,
-- no_such_item is raised.
function title_of (the_menu : menu_type) return menu_title;
--
-- Synopsis: This function returns the title of the specified
-- menu.
-- If the specified menu has not been defined or has been deleted,
-- menu_undefined is raised.

function number_of_items_in (the_menu : menu_type) return positive;
--
-- Synopsis: This function returns the number of items in the specified
-- menu.
-- If the specified menu has not been defined or has been deleted,
-- menu_undefined is raised.

procedure set_action (the_menu  : in menu.type;
the_action : in callbacks.callback_type);
--
-- Synopsis: This procedure defines a callback procedure to be invoked
-- automatically when the user selects an item from the specified menu.
-- If the specified menu has not been defined or has been deleted,
-- menu_undefined is raised.

function get_action (the_menu : menu.type)
return callbacks.callback.type;
--
-- Synopsis: This function returns the previously defined callback
-- procedure for the specified menu.
-- If the specified menu has not been defined or has been deleted,
-- menu_undefined is raised.

private

package cmd_lists is

-- Non-modal (continually selectable) commands to be associated with
-- individual browser views, specifying operations that may be
-- performed in those views at any time. A two-level hierarchy
-- of commands is supported, consisting of top-level commands and their
-- associated subcommands (if any).
-- Types:

type cmd_list_type is private; -- abstract type for a list of commands

type cmd_item is new string; -- item in a command list

subtype cmd_index is positive; -- position of a command in a list

-- Constants:

no_cmd_list : constant cmd_list_type; -- a value for an undefined command list; all objects of type cmd_list_type are initialized to this value by default, and are set to this value when deleted.

-- Exceptions:

cmd_list_undefined : exception; -- the specified command list has not been defined or is no longer defined

cmd_list_overflow : exception; -- there are insufficient resources available to create or expand the command list

subcmds_already_exist : exception; -- a subcommand list already exists for the specified command

no_such_cmd : exception; -- there is no such command in the specified command list

-- Operations:

procedure create_cmd_list (the_cmd_list : in out cmd_list_type;
number_of_cmds : in positive);

-- Synopsis: This procedure creates a command list with room for the specified number of commands. The list is initially empty (i.e., all of the items are blank).

-- Cmd_list_overflow is raised if there are insufficient resources available to create the command list.

procedure create_subcmd_list (the_cmd_list : in cmd_list_type;
the_cmd : in cmd_index;
number_of_subcmds : in positive);

-- Synopsis: This procedure creates a subcommand list associated with the specified command and with room for the specified number of subcommands. The list is initially empty (i.e., all of the items...
-- are blank).
-- If the specified command list has not been defined or has been deleted,
-- cmd_list undefined is raised.
-- If there is no such command as the specified command in the command list,
-- no such cmd is raised.
-- If a subcommand list already exists for the specified command,
-- subcmds already exist is raised.
-- Cmd_list overflow is raised if there are insufficient resources available
-- to create the subcommand list.

procedure delete_cmd_list (the_cmd_list : in out cmd_list_type);
--
-- Synopsis: This procedure deletes the specified command list.
-- If the specified command list is not defined to begin with,
-- cmd_list undefined is raised.
-- WARNING: THIS OPERATION MAY LEAVE DANGLING REFERENCES, IF THE
-- APPLICATION HAS CREATED MULTIPLE ALIASES FOR THE COMMAND LIST VIA
-- ASSIGNMENT.

procedure delete_subcmd_list (the_cmd_list : in cmd_list_type;
the_cmd      : in cmd_index);

-- Synopsis: This procedure deletes the list of subcommands associated
-- with the specified command.
-- If the specified command list has not been defined or has been deleted,
-- cmd_list undefined is raised.
-- If there is no such command as the specified command in the command list,
-- no such cmd is raised.
-- If the specified subcommand list is not defined to begin with,
-- cmd_list undefined is raised.

procedure set_cmd (in_cmd_list : in cmd_list_type;
the_cmd    : in cmd_index;
to_value   : in cmd_item);
--
-- Synopsis: This procedure sets the specified command in the specified
-- command list to the specified value.
-- If the specified command list has not been defined or has been deleted,
-- cmd_list undefined is raised.
-- If there is no such command as the specified command in the command list,
-- no such cmd is raised.
-- Cmd_list overflow is raised if there are insufficient resources available
-- to add the command to the command list.

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.
function number_of_subcmds_of (the_cmd_list : cmd_list_type;
   the_cmd : cmd_index)
   return natural;

procedure set_action (the_cmd_list : in cmd_list_type;
   the_action : in callbacks.callback_type);

function get_action (the_cmd_list : cmd_list_type)
   return callbacks.callback_type;

package graphs is
   -- Graph structures defined by the application. A graph structure
   -- consists of a collection of nodes and the arcs connecting them.
   -- The reusable browser caches the application-defined unique identifier
   -- (ID) associated with each node and each arc, as well as the
   -- application-defined kind indication for each node and each arc.
The purpose of the graph structure is to translate between objects in the browser domain and those in the application domain, and to facilitate the construction and maintenance of views.

-- Types:

type graph_type is private; -- abstract type for a graph definition
type node_type is private; -- abstract type for a node
type node_name is new string; -- the textual name associated with a node
type arc_type is private; -- abstract type for an arc
type arc_name is new string; -- the textual name associated with an arc
type arc_direction is (one_way, two_ways);

-- Constants:

no_graph : constant graph_type; -- a value for an undefined graph;
   -- all objects of type graph_type are initialized to this value
   -- by default, and are set to this value when deleted.
nc.node : constant node_type; -- a value for an undefined node;
   -- all objects of type node_type are initialized to this value
   -- by default, and are set to this value when deleted.
no.arc : constant arc_type; -- a value for an undefined arc;
   -- all objects of type arc_type are initialized to this value
   -- by default, and are set to this value when deleted.

-- Exceptions:

graph_already_exists : exception; -- a graph with the specified ID already exists

graph_not_found : exception; -- the specified graph could not be found

graph_undefined : exception; -- the specified graph has not been defined or
   -- is no longer defined

graph_overflow : exception; -- there are insufficient resources available
   -- to create or expand the graph

node_already_exists : exception; -- a node with the specified ID already exists

node_undefined : exception; -- the specified node has not been defined or
   -- is no longer defined

node_not_found : exception; -- the specified node could not be found

node_has_references : exception; -- the specified node cannot be deleted,
   -- because it is referenced by one or more arcs
arc_already_exists : exception; -- an arc with the specified ID already exists
arc_undefined : exception; -- the specified arc has not been defined or is no longer defined
arc_not_found : exception; -- the specified arc could not be found

-- Graph Operations:

procedure create_graph (   the_graph : in out graph_type;
with_id : in graph.id;
estimated_arcs : in positive
:= anticipated_arcs;
estimated_nodes : in positive
:= anticipated_nodes);

-- Synopsis: This procedure creates a graph definition having the specified ID. The translation of objects from the application domain into objects in the browser domain is optimized for a graph having the estimated number of arcs and nodes.
-- Attempting to create a second graph definition with the same ID raises graph_already_exists.
-- Graph_overflow is raised if there are insufficient resources available to create the graph definition.

procedure destroy_graph (   the_graph : in out graph_type);

-- Synopsis: This procedure destroys the specified graph definition. All arcs and nodes are deleted from the graph, and the graph becomes undefined.
-- If the specified graph is not defined to begin with, graph_undefined is raised.
-- WARNING: THIS OPERATION MAY LEAVE DANGLING REFERENCES, IF THE APPLICATION HAS CREATED MULTIPLE ALIASES FOR THE GRAPH OR FOR ANY OF ITS NODES OR ARCS VIA ASSIGNMENT.

function is_defined (the_graph : graph_type) return boolean;

-- Synopsis: This function indicates whether or not the specified graph definition is currently defined (i.e., whether or not it currently exists).

function id_of (   the_graph : graph_type) return graph_id;

-- Synopsis: This function returns the ID of the specified graph.
function get_graph (with_id : graph_id) return graph_type;

-- Synopsis: This function returns the graph having the specified ID.
-- If there is no such graph, graph_not_found is raised.

procedure add_node (the_node : in out node_type;
                   with_id : in node_id;
                   with_name : in node.name := "";
                   of_kind : in node_kind := node_kind'first;
                   to_graph : in graph_type);

-- Synopsis: This procedure defines a node having the specified ID
-- and name, and distinguished as being of the specified kind, and adds
-- it to the specified graph definition.
-- Note that the node kind defaults to the base value of the node_kind
-- type. This is useful for applications where there is no notion of
-- node kind (i.e., where all nodes are of the same kind).
-- If the specified graph definition has not been defined or has been
-- destroyed, graph_undefined is raised.
-- Attempting to add a second node with the same ID to the same graph
-- raises node_already_exists.
-- Graph_overflow is raised if there are insufficient resources available
-- to add the node.

procedure delete_node (the_node : in out node_type;
                       from_graph : in graph_type);

-- Synopsis: This procedure deletes the specified node from the specified
-- graph. The specified node is deleted and becomes undefined.
-- If the specified graph definition has not been defined or has been
-- destroyed, graph_undefined is raised.
-- If the specified node is not defined to begin with, node_undefined
-- is raised.
-- If there is no such node in the specified graph, node_not_found
-- is raised.
-- If the specified node is referenced by any arcs (as either a source
function is_defined (the_node : node_type) return boolean;
--
-- Synopsis: This function indicates whether or not the specified node is
-- currently defined.

function is_a_member_of( the_node : node_type;
the_graph : graph_type) return boolean;
--
-- Synopsis: This function indicates whether or not the specified node is
-- a member of the specified graph.
-- If the specified graph definition has not been defined or has been
-- destroyed, graph_undefined is raised.
-- If the specified node has not been defined or has been deleted,
-- node_undefined is raised.

function graph_of (the_node : node_type) return graph_type;
--
-- Synopsis: This function returns the graph that the node was originally
-- added to.
-- If the specified node has not been defined or has been deleted,
-- node_undefined is raised.
-- If the graph's definition has been destroyed, graph_undefined is
-- raised.

function has_references (the_node : node_type) return boolean;
--
-- Synopsis: This function indicates whether or not there are any arcs
-- currently referencing the specified node.
-- If the specified node has not been defined or has been deleted,
-- node_undefined is raised.

function id_of (the_node : node_type) return node_id;
--
-- Synopsis: This function returns the ID of the specified node (i.e.,
-- the ID that was specified when the node was defined).
-- If the specified node has not been defined or has been deleted,
-- node_undefined is raised.

function name_of (the_node : node_type) return node_name;
--
-- Synopsis: This function returns the name of the specified node (i.e.,
function kind_of (the_node : node_type) return node_kind;
--
-- Synopsis: This function returns the kind of the specified node (i.e.,
-- the node kind that was specified when the node was defined).
-- If the specified node has not been defined or has been deleted,
-- node_undefined is raised.

function get_node (with_id : node_id;
               from_graph : graph_type) return node_type;
--
-- Synopsis: This function returns the node having the specified ID
-- from the specified graph.
-- If the specified graph definition has not been defined or has been
-- destroyed, graph_undefined is raised.
-- If there is no such node in the specified graph, node_not_found
-- is raised.

generic
  with procedure visit (the_node : in node_type;
                      continue : out boolean);
procedure iterate_nodes (of_the_graph : in graph_type);
--
-- Synopsis: This procedure "visits" each node of the graph, executing
-- the specified "visit" procedure for each node. The iteration order
-- is not defined.
-- If the specified graph definition has not been defined or has been
-- destroyed, graph_undefined is raised.

-- Arc Operations:

procedure add_arc (the_arc : in out arc_type;
                 with_id : in arc_id;
                 with_name : in arc_name
                 := "";
                 from_node : in node_type;
                 to_node : in node_type;
                 directed : in arc_direction
                 := one_way;
                 of_kind : in arc_kind
                 := arc_kind'first;
                 to_graph : in graph_type);
--- Synopsis: This procedure defines an arc having the specified ID
--- and name, connecting the specified nodes in the implied direction(s),
--- and distinguished as being of the specified kind, and adds it to
--- the specified graph definition.
--- Note that if no arc direction is specified, the arc is assumed to be
--- unidirectional (emanating from from_node and terminating at to_node).
--- Note also that the arc kind defaults to the base value of the arc_kind
--- type. This is useful for applications where there is no notion of
--- arc kind (i.e., where all arcs are of the same kind).
--- If the specified graph definition has not been defined or has been
--- destroyed, graph_undefined is raised.
--- Attempting to add a second arc with the same ID to the same graph
--- raises arc_already_exists.
--- If either of the specified nodes has not been defined as a member of
--- the specified graph or has been deleted, node_undefined is raised.
--- Graph_overflow is raised if there are insufficient resources available
--- to add the arc.

procedure delete_arc (     the_arc   : in out arc_type;
                        from_graph : in graph_type);

--- Synopsis: This procedure deletes the specified arc from the specified
--- graph. The specified arc is deleted and becomes undefined.
--- If the specified graph definition has not been defined or has been
--- destroyed, graph_undefined is raised.
--- If the specified arc is not defined to begin with, arc_undefined
--- is raised.
--- If there is no such arc in the specified graph, arc_not_found
--- is raised.
--- WARNING: THIS OPERATION MAY LEAVE DANGLING REFERENCES, IF THE
--- APPLICATION HAS CREATED MULTIPLE ALIASES FOR THE ARC VIA ASSIGNMENT.

function is_defined (      the_arc  : arc_type) return boolean;
---
--- Synopsis: This function indicates whether or not the specified arc is
--- currently defined.

function is_a_member_of(   the_arc  : arc_type;
                        the_graph : graph_type) return boolean;
---
--- Synopsis: This function indicates whether or not the specified arc is
--- a member of the specified graph.
--- If the specified graph definition has not been defined or has been
--- destroyed, graph_undefined is raised.
function graph_of (the_arc : arc_type) return graph_type;
--
-- Synopsis: This function returns the graph that the arc was originally
-- added to.
-- If the specified arc has not been defined or has been deleted,
-- arc_undefined is raised.
-- If the graph's definition has been destroyed, graph_undefined is
-- raised.

function id_of (the_arc : arc_type) return arc_id;
--
-- Synopsis: This function returns the ID of the specified arc (i.e.,
-- the ID that was specified when the arc was defined).
-- If the specified arc has not been defined or has been deleted,
-- arc_undefined is raised.

function name_of (the_arc : arc_type) return arc_name;
--
-- Synopsis: This function returns the name of the specified arc (i.e.,
-- the name that was specified when the arc was defined).
-- If the specified arc has not been defined or has been deleted,
-- arc_undefined is raised.

function source_of (the_arc : arc_type) return node_type;
--
-- Synopsis: This function returns the source node of the specified
-- arc (i.e., the node that was specified as from_node when the arc
-- was defined).
-- If the specified arc has not been defined or has been deleted,
-- arc_undefined is raised.

function destination_of (the_arc : arc_type) return node_type;
--
-- Synopsis: This function returns the destination node of the specified
-- arc (i.e., the node that was specified as to_node when the arc
-- was defined).
-- If the specified arc has not been defined or has been deleted,
-- arc_undefined is raised.

function direction_of (the_arc : arc_type) return arc_direction;
--
-- Synopsis: This function returns the direction of the specified arc
function kind_of (the_arc : arc_type) return arc_kind;
--
-- Synopsis: This function returns the kind of the specified arc (i.e.,
-- the arc kind that was specified when the arc was defined).
-- If the specified arc has not been defined or has been deleted,
-- arc_undefined is raised.
--
function get_arc (with_id   : arc_id;
from_graph : graph_type) return arc_type;
--
-- Synopsis: This function returns the arc having the specified ID
-- from the specified graph.
-- If the specified graph definition has not been defined or has been
-- destroyed, graph_undefined is raised.
-- If there is no such arc in the specified graph, arc_not_found
-- is raised.
--
generic
with procedure visit (the_arc  : in arc_type;
continue : out boolean);
procedure iterate_arcs (of_the_graph : in graph_type);
--
-- Synopsis: This procedure "visits" each arc of the graph, executing
-- the specified "visit" procedure for each arc. The iteration order
-- is not defined.
-- If the specified graph definition has not been defined or has been
-- destroyed, graph_undefined is raised.
--
private

type graph_structure (node_hash_buckets : positive;
arc_hash_buckets  : positive); -- full type
-- declaration deferred to the package body

type graph_type is access graph_structure;
type node_info; -- full type declaration deferred to the package body
type node_type is access node_info;
type arc_info; -- full type declaration deferred to the package body
type arc_type is access arc_info;
no_graph : constant graph_type := null;
no_node : constant node_type := null;
no_arc : constant arc_type := null;
package views is

-- The user's view of application-defined graph structures, as presented
-- to the user by the reusable browser. This view abstraction enables
-- the application to control both the manner of depiction of the graph
-- structure and the semantics of user interactions with the graph
-- structure. In particular, it allows applications to control such
-- factors as which nodes and arcs are presented to the user, how they
-- are laid out on the display screen, how they are labelled, and what
-- actions are taken in response to their selection by the user.
-- Note that for some applications it may be desirable to present users
-- with several views of the same graph structure, or even views of
-- several different graph structures, simultaneously. Different views
-- may show different graphs, different subsets of the same graph, or
-- the same subset of the same graph but depicted differently.
-- Alternatively, different views of the same graph may be depicted
-- identically, but may respond differently to user actions.

-- Types:

type view_type is private; -- abstract type for a view of a graph
type view_title is new string; -- view title to be displayed
type node_type is private; -- abstract type for a view of a node
type node_label is new string; -- label to be displayed for a node
type arc_type is private; -- abstract type for a view of an arc
type arc_label is new string; -- label to be displayed for an arc
type attributes is new string; -- attributes to be displayed for node/arc
-- multiple lines of text may be separated by ASCII.LF characters
type coordinates_type is
record
  x_coordinate : natural := 0;
y_coordinate : natural := 0;
end record; -- specifies positions of nodes and arcs in the x-y plane
type dimensions_type is
record
  width : natural := 0; -- x dimension
15 February 1991

height : natural := 0; -- y dimension
end record; -- specifies dimensions of nodes and arcs

type orientation_type is (none, vertical, horizontal); -- specifies
-- orientation of layout for the view

-- Constants:

no_view : constant view_type; -- a value for an undefined view;
-- all objects of type view_type are initialized to this value
-- by default, and are set to this value when deleted.

no_node : constant node_type; -- a value for an undefined node;
-- all objects of type node_type are initialized to this value
-- by default, and are set to this value when deleted.

no_arc : constant arc_type; -- a value for an undefined arc;
-- all objects of type arc_type are initialized to this value
-- by default, and are set to this value when deleted.

no_attributes : constant attributes := ""
-- no attributes for node/arc

-- Exceptions:

view_undefined : exception; -- the specified view has not been defined or
-- is no longer defined

view_overflow : exception; -- there are insufficient resources available
-- to create or expand the view

view_is_displayed : exception; -- the specified view is currently displayed,
-- so it cannot be deleted

node_undefined : exception; -- the specified node has not been defined or
-- is no longer defined

node_not_found : exception; -- the specified node is not in the view

node_has_references : exception; -- cannot remove a node that is the
-- destination of at least one arc.

arc_undefined : exception; -- the specified arc has not been defined or
-- is no longer defined

arc_not_found : exception; -- the specified arc is not in the view

not_laid_out : exception; -- the view was not laid out before displaying it

-- View Operations:

procedure create_view (the_view : in out view_type;
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)
return cmd_lists.cmd_list_type;

-- Synopsis: This function returns the command list associated with
-- the specified view.
-- If the specified view is not defined or has been deleted, view_undefined
-- is raised.

function title_of (the_view: view_type) return view_title;

-- Synopsis: This function returns the title of the specified
-- view.
-- If the specified view is not defined or has been deleted, view_undefined
-- is raised.

function number_of_nodes_in (the_view : view_type) return natural;

-- Synopsis: This function returns the total number of nodes in the
-- specified view.
-- If the specified view has not been defined or has been deleted,
-- view_undefined is raised.

function number_of_arcs_in (the_view : view_type) return natural;

-- Synopsis: This function returns the total number of arcs in the
-- specified view.
-- If the specified view has not been defined or has been deleted,
-- view_undefined is raised.

procedure set.orientation (of_view  : in view.type;
to.value : in orientation_type);

-- Synopsis: This procedure sets the orientation of the layout of the
-- specified view to the specified value (i.e., vertical or horizontal).
-- If the specified view has not been defined or has been deleted,
-- view_undefined is raised.

procedure set_arrow_spacing (of_view  : in view.type;
to_value : in natural);

---

Page 72
15 February 1991

function get_arrow_spacing (of_view : view_type) return natural;
--
-- Synopsis: This function returns the arrow spacing value for the
-- specified view.
-- If the specified view has not been defined or has been deleted,
-- view_undefined is raised.

procedure set_outdated_flag (of_view  : in view_type;
   to.value : in boolean);
--
-- Synopsis: This procedure sets the outdated flag for the
-- specified view to the specified value, indicating whether or not the
-- view is currently up to date.
-- If the specified view is not defined or has been deleted,
-- view_undefined is raised.

function is_outdated (the_view : view_type) return boolean;
--
-- Synopsis: This function returns an indication of whether or not the
-- specified view is currently up to date.
-- If the specified view is not defined or has been deleted, view_undefined
-- is raised.

procedure set_topology_display (of.view : in view_type;
   to_value : in boolean);
--
-- Synopsis: This procedure sets the topology display flag for the
-- specified view to the specified value, indicating whether or not the
-- view's topology is to be displayed. If the view is currently displayed,
--- the topology display is immediately made visible or invisible (depending
--- on the specified value).
--- If the specified view is not defined or has been deleted,
--- view_undefined is raised.

function get_topology_display (of_view : view_type) return boolean;
---
--- Synopsis: This function returns the topology display flag for the
--- specified view.
--- If the specified view is not defined or has been deleted, view_undefined
--- is raised.

procedure display_view (the_view : in view_type);
---
--- Synopsis: This procedure displays the specified view. Only nodes and
--- arcs that have been layed out and are not suppressed are displayed.
--- If the view is already displayed, that display is revised to conform
--- with current layout and suppression settings.
--- If the specified view is not defined or has been deleted,
--- view_undefined is raised.
--- If the view has not been completely laid out (including specifying its
--- orientation), not_laid_out is raised.

function is_displayed (the_view : view_type) return boolean;
---
--- Synopsis: This function returns an indication of whether or not the
--- specified view is currently displayed.
--- If the specified view is not defined or has been deleted, view_undefined
--- is raised.

procedure search_view (the_view : in view_type;
                   for_node : in graphs.node_type);
---
--- Synopsis: This procedure searches the specified view for the specified
--- node and (if found) centers it in the display area.
--- If the specified view is not defined or has been deleted,
--- view_undefined is raised.
--- If the specified node is not defined or has been deleted,
--- graphs.node_undefined is raised.
--- If the specified node is not found in the specified view,
--- node_not_found is raised.

procedure search_view (the_view : in view_type;
                   for_arc : in graphs.arc_type);
---
-- Synopsis: This procedure searches the specified view for the specified arc and (if found) centers it in the display area.
-- If the specified view is not defined or has been deleted,
-- view_undefined is raised.
-- If the specified arc is not defined or has been deleted,
-- graphs.arc_undefined is raised.
-- If the specified arc is not found in the specified view,
-- arc_not_found is raised.

procedure erase_view (the_view : in view_type);
--
-- Synopsis: This procedure erases the display of the specified view.
-- If the specified view is not currently displayed, this procedure has no effect.
-- If the specified view is not defined or has been deleted,
-- view_undefined is raised.

procedure display_menu (the_menu : in menus.menu_type;
for_view : in view_type);
--
-- Synopsis: This procedure displays the specified modal menu, associating it with the specified view, and constrains user input exclusively to that menu. The menu is erased automatically after the user has made a selection.
-- If the specified menu is not defined or has been deleted,
-- menus.menu_undefined is raised.

procedure display_menu (the_menu : in menus.menu_type;
for_node : in node_type;
of_view : in view_type);
--
-- Synopsis: This procedure displays the specified modal menu, associating it with the specified node, and constrains user input exclusively to that menu. The menu is erased automatically after the user has made a selection.
-- If the specified node is not defined or has been deleted,
-- node_undefined is raised.
-- If the specified menu is not defined or has been deleted,
-- menus.menu_undefined is raised.

procedure display_menu (the_menu : in menus.menu_type;
for_arc : in arc_type;
of_view : in view_type);
--
-- Synopsis: This procedure displays the specified modal menu, associating
-- it with the specified arc, and constrains user input exclusively
-- to that menu. The menu is erased automatically after the user has
-- made a selection.
-- If the specified arc is not defined or has been deleted,
-- arc_undefined is raised.
-- If the specified menu is not defined or has been deleted,
-- menus.menu_undefined is raised.

generic
with procedure visit (the_view : in view_type;
continue : out boolean);

procedure iterate_views;
--
-- Synopsis: This procedure iterates through all views, and executes the
-- specified "visit" procedure for each view. The iteration order is not
-- defined. Iteration terminates when the "visit" procedure returns a
-- value of FALSE for the continue parameter.

-- Node Operations:

procedure insert_node (the_view_node : in out node_type;
of_graph_node : in graphs.node_type;
into_view : in view_type;
label_as : in node_label := "";
set_attributes : in attributes := no_attributes;
set_action : in callbacks.callback_type := callbacks.no_callback);
--
-- Synopsis: This procedure creates a view node corresponding to the
-- specified graph node, indicating that the node is to be depicted with
-- the specified label and attributes and that the specified action is to
-- be taken upon node selection, and inserts it into the specified view.
-- Note that by default the nodes are depicted with no labels or attributes.
-- If the specified view has not been created or has been deleted,
-- view_undefined is raised.
-- View_overflow is raised if there are insufficient resources available
-- to insert the node into the view.
-- WARNING: THERE IS NO PROTECTION AGAINST INSERTING TWO VIEW NODES THAT
-- BOTH REFERENCE THE SAME GRAPH NODE INTO THE SAME VIEW; DOING SO MAY
-- PRODUCE UNEXPECTED RESULTS. IT IS THE APPLICATION'S RESPONSIBILITY
-- TO ENSURE THAT THIS DOES NOT HAPPEN.

procedure remove_node (the_node : in out node_type;
function is_defined (the_node : node_type) return boolean;

---

-- Synopsis: This function indicates whether or not the specified node is currently defined.

function is_a_member_of (the_node : node_type;
the_view : view_type) return boolean;

---

-- Synopsis: This function indicates whether or not the specified node is a member of the specified view.

function graph_node_of (the_node : node_type) return graphs.node_type;

---

-- Synopsis: This function returns the graph node that the view node is a view of.

function is_in_view (graph_node : graphs.node_type;
the_view : view_type) return boolean;

---

-- Synopsis: This function indicates whether or not the specified graph node is depicted in the specified view.

-- If the specified view has not been defined or has been removed,
-- node_undefined is raised.

-- 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.
function get_action (the_node : node_type)
    return callbacks.callback_type;

--
-- Synopsis: This function returns the previously defined callback
-- procedure for the specified node.
-- If the specified node has not been defined or has been deleted,
-- node_undefined is raised.

procedure set_position (of_node    : in node_type;
     to_position : in coordinates.type);
--
-- Synopsis: This procedure sets the x-y coordinates of the specified
-- node to the specified values.
-- If the specified node has not been defined or has been deleted,
-- node_undefined is raised.

function get_position (of_node : node_type) return coordinates.type;
--
-- Synopsis: This function returns the x-y coordinates of the specified
-- node.
-- If the specified node has not been defined or has been deleted,
-- node_undefined is raised.

function get_dimensions (of_node        : node_type;
     with_orientation : orientation.type)
return dimensions_type;
--
-- Synopsis: This function returns overall the x-y dimensions (i.e.,
-- width and height) for layout of the specified node, including its icon
-- and any labels or attributes, in the specified orientation.
-- If the specified node has not been defined or has been deleted,
-- node_undefined is raised.

function get_icon_dimensions (of_node : node_type) return dimensions_type;
--
-- Synopsis: This function returns the x-y dimensions (i.e., width and
-- height) of the icon representing the node, exclusive of any labels or
-- attributes.
-- If the specified node has not been defined or has been deleted,
-- node_undefined is raised.

procedure set_suppression (of_node : in node_type;
     to_value : in boolean);
--
function is_suppressed (the_node : node_type) return boolean;
--
-- Synopsis: This function indicates whether or not the node is to be suppressed from the display.
-- If the specified node has not been defined or has been deleted,
-- node_undefined is raised.

function number_of_arcs_from (the_node : node_type) return natural;
--
-- Synopsis: This function returns the number of arcs emanating from the specified node.
-- If the specified node has not been defined or has been deleted,
-- node_undefined is raised.

function number_of_arcs_to (the_node : node_type) return natural;
--
-- Synopsis: This function returns the number of arcs terminating at (i.e., directed to) the specified node.
-- If the specified node has not been defined or has been deleted,
-- node_undefined is raised.

generic
with procedure visit (the_node : in node_type;
                     continue : out boolean);
procedure iterate_nodes (of_the_view: in view_type);
--
-- Synopsis: This procedure iterates through all of the nodes in the specified view, and executes the specified "visit" procedure for each node. The iteration order is not defined. Iteration terminates when the "visit" procedure returns a value of FALSE for the continue parameter.
-- If the specified view has not been defined or has been deleted,
-- view_undefined is raised.

-- Arc Operations:

procedure insert_arc ( the_view_arc : in out arc_type;
of_graph_arc : in graphs.arc_type;
into_view : in view_type;
procedure remove_arc (the_arc : in out arc_type;
from_view : in view_type);

-- Synopsis: This procedure removes the specified arc from the specified
-- view. The specified arc is deleted and becomes undefined.
-- If the specified view has not been defined or has been
-- deleted, view_undefined is raised.
-- If the specified arc is not defined to begin with, arc_undefined
-- is raised.
-- If there is no such arc in the specified view, arc_not_found
-- is raised.
-- WARNING: THIS OPERATION MAY LEAVE DANGLING REFERENCES, IF THE
-- APPLICATION HAS CREATED MULTIPLE ALIASES FOR THE ARC VIA ASSIGNMENT.

function is_defined (the_arc : arc_type) return boolean;

-- Synopsis: This function indicates whether or not the specified arc is
-- currently defined.

function is_a_member_of( the_arc : arc_type;
the_view : view_type) return boolean;

-- 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 -- arc_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 vies 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_vies : vies_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

Page 82
function destination_of (the_arc : arc_type) return node_type;
--
-- Synopsis: This function returns the destination node of the specified
-- arc (i.e., the node that was specified as to_node when the arc
-- was defined).
-- If the specified arc has not been defined or has been deleted,
-- arc_undefined is raised.

function has_attributes (the_arc : arc_type) return boolean;
--
-- Synopsis: This function indicates whether or not any attributes have
-- been set for the specified arc.
-- If the specified arc has not been defined or has been deleted,
-- arc_undefined is raised.

procedure refresh_attributes (the_arc       : in arc_type;
                           new_attributes : in attributes);
--
-- 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
1489 -- automatically when the user selects the specified arc.
1490 -- If the specified arc has not been defined or has been deleted,
1491 -- arc_undefined is raised.
1492
1493 function get_action (the_arc : arc_type)
1494 return callbacks.callback.type;
1495 --
1496 -- Synopsis: This function returns the previously defined callback
1497 -- procedure for the specified arc.
1498 -- If the specified arc has not been defined or has been deleted,
1499 -- arc_undefined is raised.
1500
1501 procedure set_position (of_arc     : in arc_type;
1502 to_position : in coordinates_type);
1503 --
1504 -- Synopsis: This procedure sets the x-y coordinates of the specified
1505 -- arc to the specified values.
1506 -- If the specified arc has not been defined or has been deleted,
1507 -- arc_undefined is raised.
1508
1509 function get_position (of.arc : arc.type) return coordinates.type;
1510 --
1511 -- Synopsis: This function returns the x-y coordinates of the specified
1512 -- arc.
1513 -- If the specified arc has not been defined or has been deleted,
1514 -- arc_undefined is raised.
1515
1516 function get_dimensions (of.arc : arc.type;
1517 with_orientation : orientation.type)
1518 return dimensions.type;
1519 --
1520 -- Synopsis: This function returns overall the x-y dimensions (i.e.,
1521 -- width and height) for layout of the specified arc, including its icon
1522 -- and any labels or attributes, in the specified orientation.
1523 -- If the specified arc has not been defined or has been deleted,
1524 -- arc_undefined is raised.
function get_icon_dimensions (of_arc : arc_type) return dimensions_type;

-- Synopsis: This function returns the x-y dimensions (i.e., width and
-- height) of the icon representing the arc, exclusive of any labels or
-- attributes.
-- If the specified arc has not been defined or has been deleted,
-- arc_undefined is raised.

procedure set_suppression (of_arc  : in arc_type;
to_value : in boolean);

-- Synopsis: This procedure sets the suppression flag for the
-- specified arc to the specified value, indicating whether or not
-- the arc is to be suppressed from the display.
-- If the specified arc has not been defined or has been deleted,
-- arc_undefined is raised.

function is_suppressed (the_arc : arc_type) return boolean;

-- Synopsis: This function indicates whether or not the arc is to be
-- suppressed from the display.
-- If the specified arc has not been defined or has been deleted,
-- arc_undefined is raised.

generic
    with procedure visit (the_arc  : in arc_type;
    continue : out boolean);
procedure iterate_arcs (of_the_view : in view_type);

-- Synopsis: This procedure iterates through all of the arcs in the
-- specified view, and executes the specified "visit" procedure for each
-- arc. The iteration order is not defined. Iteration terminates when
-- the "visit" procedure returns a value of FALSE for the continue
-- parameter.
-- If the specified view has not been defined or has been deleted,
-- view_undefined is raised.

generic
    with procedure visit (the_arc  : in arc_type;
    continue : out boolean);
procedure iterate_arcs_from (the_node : in node_type);

-- Synopsis: This procedure iterates through all of the arcs emanating
-- from the specified node, and executes the specified "visit" procedure

Page 85
**package** views;

**package** text is

```
 -- Abstract data types and utilities for text-based interactions.

-- Types:

type buffer_type is private;

type display_type is private;

type display_kind is (buffer_display, file_display);
```
-- Exceptions:

buffer_undefined : exception; — the specified buffer does not exist
buffer_overflow : exception; — insufficient resources are available to
-- create the buffer or to append to it
buffer_in_use : exception; -- the buffer contents are currently displayed

display_undefined : exception; -- the specified text display does not exist
display_overflow : exception; -- insufficient resources are available to
-- create the text display
wrong_display_kind : exception; -- operation not compatible with the
-- kind of display specified

-- Constants:

no_display : constant display_type; -- a value for an undefined text
-- display; all objects of type display_type are initialized to this
-- value by default, and are set to this value when erased.

no_buffer : constant buffer_type; -- a value for an undefined text
-- buffer; all objects of type buffer_type are initialized to this
-- value by default, and are set to this value when deleted.

max_input_length : constant := 100; -- max length of arbitrary string input

-- Buffer Operations:

procedure create_buffer (the_buffer : in out buffer_type;
  size : in positive);

-- Synopsis: This procedure creates a text buffer of the specified size
-- (in characters).
-- Buffer_overflow is raised if there are insufficient resources available
-- to create the buffer.

procedure delete_buffer (the_buffer : in out buffer_type);

-- Synopsis: This procedure deletes the specified text buffer.
-- If the specified text buffer does not exist in the first place,
-- buffer_undefined is raised.
-- If the contents of the specified text buffer are currently being
-- displayed, buffer_in_use is raised.
-- WARNING: THIS OPERATION MAY LEAVE DANGLING REFERENCES, IF THE
-- APPLICATION HAS CREATED MULTIPLE ALIASES FOR THE TEXT BUFFER VIA
procedure clear_buffer (the_buffer : in out buffer_type);

-- Synopsis: This procedure clears the specified text buffer.
-- If the specified text buffer does not exist, buffer_undefined is raised.

procedure append (the_buffer : in out buffer_type;
                   the_text  : in string);

-- Synopsis: This procedure appends the specified string to the
-- specified text buffer.
-- If the specified text buffer does not exist, buffer_undefined is raised.
-- If appending the specified text to the buffer would cause it to overflow,
-- buffer_overflow is raised.

procedure append_line (the_buffer : in out buffer_type;
                       the_text  : in string);

-- Synopsis: This procedure appends the specified string and an end-of-
-- line character to the specified text buffer.
-- If the specified text buffer does not exist, buffer_undefined is raised.
-- If appending this text to the buffer would cause it to overflow,
-- buffer_overflow is raised.

procedure new_line (the_buffer : in out buffer_type;
                    count     : in positive := 1);

-- Synopsis: This procedure appends the specified number of end-of-line
-- characters to the specified text buffer.
-- If the specified text buffer does not exist, buffer_undefined is raised.
-- If appending the specified number of end-of-line characters to the
-- buffer would cause it to overflow, buffer_overflow is raised.

function length_of (the_buffer : in buffer_type) return natural;

-- Synopsis: This function returns the length of the contents of the
-- specified text buffer, in characters.
-- If the specified text buffer does not exist, buffer_undefined is raised.

function contents_of (the_buffer : in buffer_type) return string;

-- Synopsis: This function returns the contents of the specified text
-- buffer.
-- If the specified text buffer does not exist, buffer_undefined is raised.
-- Display Operations:

procedure display_text (the_display : in out display_type;
from_buffer : in buffer_type;
use_title  : in string := " ";
quit_action : in callbacks.callback_type
:= callbacks.no_callback);

-- Synopsis: This procedure displays text from the specified buffer on
-- the screen, exhibiting the specified title, and returns a display object
-- that may later be used to erase or refresh the display. The text
-- remains displayed until explicitly erased by the application (via a
-- call to erase_text). The user may request that the text display be
-- erased by selecting its "QUIT" button. In that case, the application
-- is notified of the user's request via a text_quit event. The
-- application may specify a callback procedure (quit_action) to handle
-- the text_quit event.
-- If the specified text buffer does not exist, buffer undefined is raised.

procedure display_text (the_display : in out display_type;
from_file  : in string;
use_title  : in string := " ";
quit_action : in callbacks.callback_type
:= callbacks.no_callback);

-- Synopsis: This procedure displays text from the specified file (i.e.,
-- the file having the specified external name) on the screen, exhibiting
-- the specified title, and returns a display object that may later be used
-- to erase or refresh the display. The text remains displayed until
-- explicitly erased by the application (via a call to erase_text). The
-- user may request that the text display be erased by selecting its
-- "QUIT" button. In that case, the application is notified of the user's
-- request via a text_quit event. The application may specify a callback
-- procedure (quit_action) to handle the text_quit event.

procedure erase_text (the_display : in out display_type);

-- Synopsis: This procedure erases the specified text display from the
-- screen and then makes the display object undefined.
-- If the specified display object is initially undefined,
-- display undefined is raised.
-- WARNING: THIS OPERATION MAY LEAVE DANGLING REFERENCES, IF THE
-- APPLICATION HAS CREATED MULTIPLE ALIASES FOR THE DISPLAY OBJECT VIA
-- ASSIGNMENT.
procedure refresh_text (the_display : in display_type);

--

-- Synopsis: This procedure updates the specified text display, using
-- the latest contents of its associated buffer or file.
-- If the specified display object is undefined, display_undefined is
-- raised.

function kind_of (the_display : display_type) return display_kind;

--

-- Synopsis: This function indicates whether the text being displayed
-- is from file or a buffer.
-- If the specified display object is undefined, display_undefined is
-- raised.

function get_buffer (the_display : display_type) return buffer_type;

--

-- Synopsis: This function obtains the text buffer that is associated
-- with the specified display.
-- If the specified display object is undefined, display_undefined is
-- raised.
-- If the specified display is not of kind buffer_display,
-- wrong_display_kind is raised.

function get_file (the_display : display_type) return string;

--

-- Synopsis: This function obtains the name of the text file that is
-- associated with the specified display.
-- If the specified display object is undefined, display_undefined is
-- raised.
-- If the specified display is not of kind file_display,
-- wrong_display_kind is raised.

-- Miscellaneous Text Utilities:

procedure display_alert (the_alert_msg : in string);

--

-- Synopsis: This procedure displays an alert message (e.g., an error
-- message), which remains displayed until acknowledged by the user.
-- The user must acknowledge the alert message before any further
-- selections can be made.

procedure prompt_for_input (the_prompt : in string := " ",
                          input_length : in positive)
--- Synopsis: This procedure prompts the user to type in a string of at
most the specified input length, and causes the specified action
routine to be invoked once the user has responded. The application
is notified of the user’s response via a string_input event. The
application may define a callback procedure (the_action) to handle
the string_input event. The user must respond to the prompt before
any further selections can be made.

private
  type buffer_structure (size : positive); — full type declaration deferred
  — to the package body
  type buffer_type is access buffer_structure;
  no_buffer : constant buffer_type := null;
  type display_info (kind : display.kind); — full type declaration deferred
  — to the package body
  type display_type is access display_info;
  no_display : constant display_type := null;
end text;

package view_utilities is
  — Generic utility routines to facilitate the construction and maintenance
  — of views.
  —
  generic
  — Node selection function and functions to supply representation
  — and behavioral characteristics for each node.
  with function include_node (the_node : graphs.node_type)
    return boolean;
  with function label_for_node (the_node : graphs.node_type)
    return views.node_label;
  with function attributes_for_node (the_node : graphs.node_type)
    return views.attributes;
  with function action_for_node (the_node : graphs.node_type)
    return callbacks.callback_type;
  — Arc selection function and functions to supply representation
with function include_arc (the_arc : graphs.arc_type)
    return boolean;
with function label_for_arc (the_arc : graphs.arc_type)
    return views.arc_label;
with function attributes_for_arc (the_arc : graphs.arc_type)
    return views.attributes;
with function action_for_arc (the_arc : graphs.arc_type)
    return callbacks.callback_type;

procedure construct_view (the_view : in out views.view_type;
    of_graph : in graphs.graph_type;
    with_cmd_list : in cmd_lists.cmd_list_type;
    with_title : in views.view.title;
    estimated_arcs : in positive
        := anticipated_arcs;
    estimated_nodes : in positive
        := anticipated_nodes);

-- Synopsis: This procedure automates the construction of a view of a
-- graph. It scans all nodes and arcs of the specified graph, applies
-- the "include" function to determine whether or not to include each one,
-- and inserts the included arcs and nodes into the view.
-- The "label_for", "attributes_for" and "action_for" functions are applied
-- to each included node and arc to obtain the required representation
-- and behavioral characteristics.
-- The view is created with the specified command list and 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.

generic

-- Node selection function and functions to supply representation
-- and behavioral characteristics for each node.

with function include_node (the_node : graphs.node_type)
    return boolean;
with function label_for_node (the_node : graphs.node_type)
    return views.node_label;
with function attributes_for_node (the_node : graphs.node_type)
    return views.attributes;
with function action_for_node (the_node : graphs.node_type)
    return callbacks.callback_type;
1886  -- Arc selection function and functions to supply representation
1887  -- and behavioral characteristics for each arc.
1888
1889  with function include_arc (the_arc : graphs.arc_type)
1890  return boolean;
1891  with function label_for_arc (the_arc : graphs.arc_type)
1892  return views.arc_label;
1893  with function attributes_for_arc (the_arc : graphs.arc_type)
1894  return views.attributes;
1895  with function action_for_arc (the_arc : graphs.arc_type)
1896  return callbacks.callback_type;
1897
1898  -- Layout algorithm for re-layout.
1899
1900  with procedure layout (the_view : in views.view_type);
1901
1902  procedure refresh_view (the_view : in views.view_type);
1903
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;
procedure filter_view (the_view : in views.view_type);
--
-- Synopsis: This procedure filters an existing view, by suppressing
-- the display of some or all of its nodes and arcs. The nodes are
-- filtered first, then the arcs. Any node that is not explicitly
-- suppressed is unsuppressed by default. Any arc that is not
-- explicitly suppressed is unsuppressed if and only if neither its
-- source nor destination node has been suppressed. The filtering
-- is not apparent to the user until the view is redisplayed (via a
-- call to the views.display_view procedure).

end view.utilities;

---

package layout.algorithms is
--
-- Algorithms for laying out a view of a directed graph.
--
-- Constants:

procedure cyclic_horizontal (the_view : in views.view_type;
  x_pad   : in natural := 30;
  y_pad   : in natural := 10);
--
-- Synopsis: This procedure lays out the specified view horizontally,
-- using an algorithm accommodates cycles. The algorithm is based on
-- an algorithm presented in the following paper:
-- Robins, G., "The ISI Grapher: a Portable Tool for Displaying Graphs
-- Pictorially," ISI/RS-87-196, USC/Information Sciences Institute,
-- reprinted from the Proceedings of Symbolikka '87, Helsinki, Finland,
-- It has been optimized somewhat, however, and has been modified
-- substantially to accommodate cycles and to lay out arcs as well as nodes.
-- The x_pad and y_pad parameters specify the minimum spacing (in pixels)
-- between node and arc depictions in the X and Y directions, respectively.

procedure acyclic_horizontal (the_view : in views.view_type;
  x_pad   : in natural := 30;
  y_pad   : in natural := 10);
--
-- Synopsis: This procedure lays out the specified view horizontally
-- as well, but uses an algorithm that does not handle cycles. This is
-- essentially the same algorithm as is used for the cyclic_horizontal
procedure cyclic_vertical (the_view : in views.view_type;
  x_pad   : in natural := 10;
  y_pad   : in natural := 30);

-- Synopsis: This procedure lays out the specified view vertically,
-- using an algorithm that accommodates cycles. The algorithm is based on
-- an algorithm presented in the following paper:
-- Robins, G., "The ISI Grapher: a Portable Tool for Displaying Graphs
-- Pictorially," ISI/RS-87-196, USC/Information Sciences Institute,
-- reprinted from the Proceedings of Symboliikka '87, Helsinki, Finland,
-- It has been rewritten to produce a vertical layout, rather than a
-- horizontal layout. It has also been optimized somewhat, and has been
-- modified to accommodate cycles and to lay out arcs as well as nodes.
-- The x_pad and y_pad parameters specify the minimum spacing (in pixels)
-- between nodes and arc depictions in the X and Y directions, respectively.

procedure acyclic_vertical (the_view : in views.view_type;
  x_pad   : in natural := 10;
  y_pad   : in natural := 30);

-- Synopsis: This procedure lays out the specified view vertically
-- as well, but uses an algorithm that does not handle cycles. This is
-- essentially the same algorithm as is used for the cyclic_vertical
-- layout procedure, but without the modifications to accommodate cycles.
-- The x_pad and y_pad parameters specify the minimum spacing (in pixels)
-- between nodes and arc depictions in the X and Y directions, respectively.
-- Because it does not have to check for cycles, this algorithm is faster
-- than the algorithm used for the cyclic_vertical procedure. Therefore,
-- this procedure should be preferred for applications that can guarantee
-- that the view does not contain a cycle.
-- WARNING: IF USED ON A VIEW CONTAINING A CYCLE, THIS PROCEDURE WILL
-- EITHER HANG OR RAISE A STORAGE_ERROR EXCEPTION.

end layout_algorithms;
type event_info (kind : event_kind := event_kind'first) is -- describes the event in detail

record
  view : views.view_type;
  case kind is
  when position_select =>
    position : views.coordinates_type;
  when command_select =>
    command : cmd_lists.cmd_index;
  when subcommand_select =>
    topcommand : cmd_lists.cmd_index;
    subcommand : cmd_lists.cmd_index;
  when menu_item_select =>
    menu : menus.menu_type;
    item : menus.item_index;
  when node_menu_item_select =>
    n_node : views.node_type;
    n_menu : menus.menu_type;
    n_item : menus.item_index;
  when arc_menu_item_select =>
    a_arc : views.arc_type;
    a_menu : menus.menu_type;
    a_item : menus.item_index;
  when menu_cancel =>
    null;
  when node_select =>
    node : views.node_type;
  when arc_select =>
    arc : views.arc_type;
  when text_quit =>
    display : text.display_type;
  when string_input =>
    input : string (1..text.max_input.length);
    length : natural;
  when input_cancel =>
    null;
  when browser_quit =>
    null;
  end case;
end record;
procedure initialize (main_commands : in cmd_lists.cmd_list_type);

-- Synopsis: This procedure displays the browser application's main
-- window and the specified command list. If the main window is already
-- displayed, this procedure has no effect.
-- If the specified command list has not been defined or has been deleted,
-- cmd_lists.cmd_list_undefined is raised.

procedure browse (event : out event_info);

-- Synopsis: This procedure activates the browser, thereby allowing the
-- user to interact with the display. Ideally, the application would
-- define actions procedures (callbacks) for all display objects, which
-- would be automatically invoked by the browser in response to user
-- actions. In this case, the browse procedure would never exit. This
-- is the preferred style of interaction, since it avoids interference
-- with the window system. If, however, it is desired that the browser
-- return control to the application in response to certain user actions,
-- the application need only refrain from defining action procedures for
-- those actions. If the user performs some action for which there is
-- no action procedure defined, the browse procedure exits and returns
-- an event indicating the nature of the user action.

procedure quit;

-- Synopsis: This procedure erases and destroys all windows created
-- by the browser application and terminates its connection with the
-- underlying window system. If this procedure is called from within a
-- callback procedure, the browse procedure will exit upon completion of
-- the callback and will return a browser_quit event.
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.

![Browser Screen Layout](image)

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 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.

<table>
<thead>
<tr>
<th>Name</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>bshell</td>
<td>application</td>
</tr>
<tr>
<td>bshell_bb</td>
<td>shell</td>
</tr>
<tr>
<td>bshell_bb*vp</td>
<td>bulletin board</td>
</tr>
<tr>
<td>bshell_bb<em>vp</em>arc_&lt;arc_kind&gt;</td>
<td>command</td>
</tr>
<tr>
<td>bshell_bb<em>vp</em>arc_atrib_&lt;arc_kind&gt;</td>
<td>ascii string</td>
</tr>
<tr>
<td>bshell_bb<em>vp</em>arc_label_&lt;arc_kind&gt;</td>
<td>label</td>
</tr>
<tr>
<td>bshell_bb<em>vp</em>node_&lt;node_kind&gt;</td>
<td>command</td>
</tr>
<tr>
<td>bshell_bb<em>vp</em>node_atrib_&lt;node_kind&gt;</td>
<td>ascii string</td>
</tr>
<tr>
<td>bshell_bb<em>vp</em>node_label_&lt;node_kind&gt;</td>
<td>label</td>
</tr>
<tr>
<td>bshell_main_bb</td>
<td>bulletin board</td>
</tr>
<tr>
<td>bshell_main_bb*popup_alert_text_psh</td>
<td>transient shell</td>
</tr>
<tr>
<td>bshell_main_bb*popup_alert_text_bb</td>
<td>bulletin board</td>
</tr>
<tr>
<td>bshell_main_bb<em>popup_alert_text_bb</em>popup_alert_text_quit</td>
<td>command</td>
</tr>
<tr>
<td>bshell_main_bb<em>popup_alert_text_bb</em>popup_alert_text_bb*popup_alert_text</td>
<td>ascii string</td>
</tr>
<tr>
<td>bshell_main_bb*popup_dialog_psh</td>
<td>transient shell</td>
</tr>
<tr>
<td>bshell_main_bb<em>popup_dialog_psh</em>popup_dialog</td>
<td>dialog</td>
</tr>
<tr>
<td>*popup_str_text_sh</td>
<td>top level shell</td>
</tr>
<tr>
<td><em>popup_str_text_sh</em>popup_str_text_bb</td>
<td>bulletin board</td>
</tr>
<tr>
<td><em>popup_str_text_sh</em>popup_str_text_bb*popup_str_text</td>
<td>ascii string</td>
</tr>
</tbody>
</table>
The following file listing is from the Browser file, which is the resource file for the sample application distributed with the Reusable Graphical Browser.

```
# Resource file for the Reusable Graphical Browser sample application
#
# RGB resource file name = Browser
# RGB resource directory path = @XAPPLRESDIR
#
# bitmap file path (must be set to directory containing bitmap files)
#----------------------------------------
*bitmapFilePath: /usr/zoo/chen/stars/code/application/bitmaps
#
# initial size of the shell
bshell.width: 350
```
# initial size of the view port
#
bsshell*bsshell_bb*vp.width: 300
bsshell*bsshell_bb*vp.height: 400

# initial size of the shell's bulletin board
#
##bsshell*bsshell_bb.width: 700
##bsshell*bsshell_bb.height: 500
bsshell*bsshell_bb.width: 800
bsshell*bsshell_bb.height: 600

# fix up the arcs
#
##bsshell*bsshell_bb*vp.arc_STRUCTURAL.borderWidth: 0 -- perm set to 0 in code
bsshell*bsshell_bb*vp.arc_STRUCTURAL.internalHeight: 0
bsshell*bsshell_bb*vp.arc_STRUCTURAL.internalWidth: 0

# show the arcs using bitmaps
#
##bsshell*bsshell_bb*vp.arc_STRUCTURAL.sensitive: false
##bsshell*bsshell_bb*vp.arc_STRUCTURAL.bitmap: point.xbm
bsshell*bsshell_bb*vp.arc_STRUCTURAL.bitmap: smalltriangle.xbm

# set dimensions of the arcs' attribute widgets
#
bsshell*bsshell_bb*vp.arc_attrib_STRUCTURAL.height: 30
bsshell*bsshell_bb*vp.arc_attrib_STRUCTURAL.width: 150

# set dimensions of the arcs' label widgets
#
##bsshell*bsshell_bb*vp.arc_label_STRUCTURAL.height:20
##bsshell*bsshell_bb*vp.arc_label_STRUCTURAL.height:40
bsshell*bsshell_bb*vp.arc_label_STRUCTURAL.border_width: 1

# show the nodes using bitmaps
#
bsshell*bsshell_bb*vp.node_FILE.bitmap: bigsquare.xbm
bsshell*bsshell_bb*vp.node_DIRECTORY.bitmap: bigcircle.xbm
bsshell*bsshell_bb*vp.node_OTHER.bitmap: bigquestion.xbm

# set dimensions of the nodes
#

# set dimensions of the nodes' attributes widgets
#-------------------------------------------
bshell*bbshell_bb*vp*node_ATTRIB_FILE.height: 50
bshell*bbshell_bb*vp*node_ATTRIB_FILE.width: 100
bshell*bbshell_bb*vp*node_ATTRIB_DIRECTORY.height: 50
bshell*bbshell_bb*vp*node_ATTRIB_DIRECTORY.width: 100

# set dimensions of the nodes' label widgets
#----------------------------------------
bshell*bbshell_bb*vp*node_LABEL_FILE.borderWidth: 0
bshell*bbshell_bb*vp*node_LABEL_DIRECTORY.borderWidth: 0

# set dimensions of the popup text window
#-------------------------------------
*popup_str_text_sh.width: 160
*popup_str_text_sh.height: 85
*popup_str_text.width: 150
*popup_str_text.height: 85

#
*popup_file_text_sh.width: 150
*popup_file_text_sh.height: 85
*popup_file_text.width: 150
*popup_file_text.height: 85

#
*popup_alert_text_psh.width: 300
*popup_alert_text_psh.height: 50
*popup_alert_text.width: 300
*popup_alert_text.height: 50

# set dimensions of the popup dialog window
#-------------------------------------
*popup_dialog_psh.width: 600
*popup_dialog_psh.height: 90
*popup_dialog.width: 600
*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


