NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

VISUAL META-PROGRAMMING LANGUAGE GRAPHICAL USER INTERFACE FOR GENERATIVE PROGRAMMING

by

Steven Carpenter

September 2002

Thesis Advisor:	Mikhail Auguston
Co-Advisor:	Richard Riehle

REPORT DOCUMENTATION PAGE			*Form Approved OMB No. 0704-0188*
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** September 2002	**3. REPORT TYPE AND DATES COVERED** Master's Thesis	
4. TITLE AND SUBTITLE: Visual Meta-Programming Language Graphical User Interface for Generative Programming.			**5. FUNDING NUMBERS**
6. AUTHOR(S) Steven M Carpenter			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A			**10. SPONSORING / MONITORING AGENCY REPORT NUMBER**
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited			**12b. DISTRIBUTION CODE**

13. ABSTRACT *(maximum 200 words)*

A Visual Meta-Programming Language allows the user to see a graphic representation of the data flow between components. Like the visual programming concepts for common programming languages in use today, this language makes it easier to build software by putting together graphical elements that correspond to larger and more complex pieces of code. This research will develop the implementation of a visual meta-programming language graphical user interface for program generation. The objective is to create an interface that represents programming data flow using the visual meta-programming language, allows the user to add, modify, and delete elements of the program, and generates formatted output that can be used by generative programs to produce code. Areas of study will include efficient data structure design to capture the nature and characteristics of visual elements of the language and translation of visual design to a format suitable for use by other programs.

14. SUBJECT TERMS Meta-Programming, visual language, data-flow, graphical user interface			**15. NUMBER OF PAGES** 107
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UL

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

VISUAL META-PROGRAMMING LANGUAGE GRAPHICAL USER INTERFACE FOR GENERATIVE PROGRAMMING

Steven M Carpenter
Lieutenant, United States Navy
B.S., Oregon State University, 1994

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September 2002

Author:
Steven M Carpenter

Approved by:
Mikhail Auguston, Thesis Advisor

Richard Riehle, Co-Advisor

C.S. Eagle, Chairman
Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

A Visual Meta-Programming Language allows the user to see a graphic representation of the data flow between components. Like the visual programming concepts for common programming languages in use today, this language makes it easier to build software by putting together graphical elements that correspond to larger and more complex pieces of code.

This research will develop the implementation of a visual meta-programming language graphical user interface for program generation. The objective is to create an interface that represents programming data flow using the visual meta-programming language, allows the user to add, modify, and delete elements of the program, and generates formatted output that can be used by generative programs to produce code. Areas of study will include efficient data structure design to capture the nature and characteristics of visual elements of the language and translation of visual design to a format suitable for use by other programs.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

Sincere Appreciation to

Dr. Auguston & Mr. Riehle

For their patience and support

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. BACKGROUND OF VISUAL PROGRAMMING

Pictures have always had the ability to convey more meaning concisely to people than writing or speaking, consider the aphorism “ a picture is worth a thousand words”. Since pictures are so good at conveying information in a confined space, there has been a lot of interest in their use in computers. Their ability to show the user or programmer what is happening or how things are related rather than describing them in words is the primary goal in visual programming. Whether showing a beginning programmer how loops and branches occur or showing an experienced programmer the architecture of a system , proper selections and use of diagrams and pictures allows a faster understanding of complex interactions than reading text. despite the promise of this approach, bringing the power of pictures to bear in software has been a long and difficult process.

When using pictures or symbols, the biggest question is what do they mean. How does the symbol shown relate to something in real life? The two parts of this question are how do you represent logical objects visually and how can we represent visual objects logically.

Due to the problems inherent in differing interpretations of pictures based on the user’s background, there is a need to define a grammar and language of symbols that all people involved in the same project understand . Like any language, one needs to learn the words and associated meanings before one can communicate.This has been done through the development of both informal and formal languages of icons that have been selected based on their commonality of interpretation and their applicability to the problem domain.

Informal languages are loose alphabets of icons that change meaning depending on the application in use. They tend to form from common usage or developers preference and propagate via the applications that use the language. An excellent example of this type of language is the MS windows/office iconography. These languages also tend to be used primarily in information processing application. Images represent actions to be taken on the focus area associated with the images, usually a workspace displaying information.

Formal languages on the other hand tend to be designed for a particular purpose. They exhibit a strictly defined set of images with associated meaning. The rules for combining images are explicitly stated and are often domain specific. Developers formalize these languages to reduce ambiguity in the graphics and to increase the ability of the computer to understand the language. A good example of this would be the language of PSDL. [1]

1. Visual Information Processing

The differences between these types of visual languages are reflected in the two areas that have come to comprise the field of visual programming.[2] The first area to fall under the umbrella of visual programming is the field of visual information processing. Objects usually have an inherent visual representation that can easily be associated with a matching logical interpretation. The objects are graphical elements, but their implementation and usage is usually based on a textual programming language. Originally used to show file system structure, it has since expanded to include the selection and manipulation of objects graphically. It is now a relatively mature area exemplified by the Mac and Wintel interfaces. It follows the concept that icons represent actions and objects and the user interaction with the icons determines the behavior of the computer.

In the common example from most windowing systems, the graphical representation of the file manager directory display mirrors the directory structure of the hard drive. Clicking on a directory changes the active directory to what was selected and reflects the change with an open folder icon to indicate that it is open. The scissors icon on the toolbar represents a physical action of cutting out whatever is selected. These actions are coded in a traditional text based language like C interpreting the users actions with a mouse as method calls to the appropriate objects or as procedure calls with the identified parameters depending on whether an Object Oriented or Procedural approach was taken. The Applications that fit into the visual information processing definition include image processing, vision, office automation, and image communication.

A hybrid of this area and the second, visual-programming languages, currently dominates the general view of visual programming. Popularized as “visual basic or visual c++”, these are programs that use the iconographic paradigm to “program” GUI’s. In general they are based on a textual programming language with a library of GUI elements and rules on combining them. The programmer is provided with a palette of common and customizable icons that he can arrange on the screen to best capture the user-computer interaction necessary for whatever application he is programming. For example, in Visual Basic, the user could create a single window with a scroll bar on the right, a pull down selection box at the top, and a button labeled “Command” in the middle. Once the interface is arranged to his satisfaction, the program generates the basic source code for the graphics and interface modules from its library of component code. The programmer then has to modify the GUI code to interface with the functionality of the application as he normally would if programming the entire program himself in the text based language. In the example above he would have to write the procedure to be called when the button is pressed within the stub VB provides which links it to the button.

2. Visual Programming Languages

The other area of visual programming is that of visual programming languages. The concept here is that by using pictures at the correct level of abstraction, programs themselves can be described and created without the programmer having to write text based code at all. In this system objects do not necessarily have an inherent visual representation. Abstractions are represented by standardized symbols tied to the specific abstraction and defined by the language alphabet. These symbols vary from language to language and are usually dependent on the problem domain. Most of these languages are based on data or control flow graph notation and model the transformation of data or the paths of control though the program. Depending on the graphical language used the representation of a program as a graph can be easily manipulated to apply graph based model checking, complexity measurement and other analysis tools. Application areas for visual programming include general programming, teaching abstractions, modeling, and specification description. A few of the visual languages in use today include PSDL, Prograph[3], LabVIEW[4], and V.

B. PURPOSE

This product will implement a visual meta-programming language graphical user interface for program generation. The objective is to create an interface that represents programming data flow using the visual meta-programming language, allowing the user to add, modify, and delete elements of the program, and generate formatted output that can be used by generative programs to produce code. The program will be used to graphically represent programs providing a more intuitive and easily understood way of seeing functionality than straight code. From the representation created, output files can be saved and imported into a code generator that will write code implementing the program.

This interface will be used to aid in the development of the V language and notation and to provide input to the compiler for testing. It allows the user to quickly and easily generate and modify visual programs for use in testing the code generator. This ease of use will also allow for better evaluation of the usability of the V language and the effectiveness of its symbology.

II. OVERVIEW OF V VISUAL METAPROGRAMMING LANGUAGE

V is a visual programming language that uses dataflow notation to build a graphical representation of meta-programs that can then be used to generate ada code to implement its functionality. Developed by Mikhail Auguston to experiment with the representation of the dependancies between data and process[5], it lends itself well to use in the area of meta-programming. It expands on work done on other dataflow-based visual programming languages like labVIEW and Prograph.[6,7] It is fully scalable from abstract, high-level program definition to specific iterative constructs common to most programming languages. Since language processors in most cases are well represented in dataflow notation, this language was developed to use visual representation of dependencies between data flow items. It is intended to provide clear correspondence between source code (data flow diagram) and order of execution. The meta-programs that V is designed to produce are programs that manipulate other programs, particularly program generators, checkers, compilers, and code analyzers.

A. NOTATION

The notation for the V language as used by this project is derived from the basic icons listed by Auguston, Berzins, and Bryant in their paper Visual Meta-Programming Language[6]. These icons were modified slightly for ease of implementation with the Gtkada toolkit and are listed in figure 1.

1. Node Icons

Icons consist of nodes representing data or manipulation of data; connections, representing the paths or associations between nodes; and ports which are the places on a node available for a connection. Data boxes contain scalar or aggregate data values along with the associated type. Each has zero or more output ports as well as zero or more input ports with optional names that also appear in the expression representing the data value. Rules contain the name of the rule and any input or output ports required. Patterns contain the same information as data boxes and try to match their input to the pattern in order to pass it on. If the pattern does not match, the diagram fails. Switches have one or more input ports and an equal number of associated output ports for true and false. If

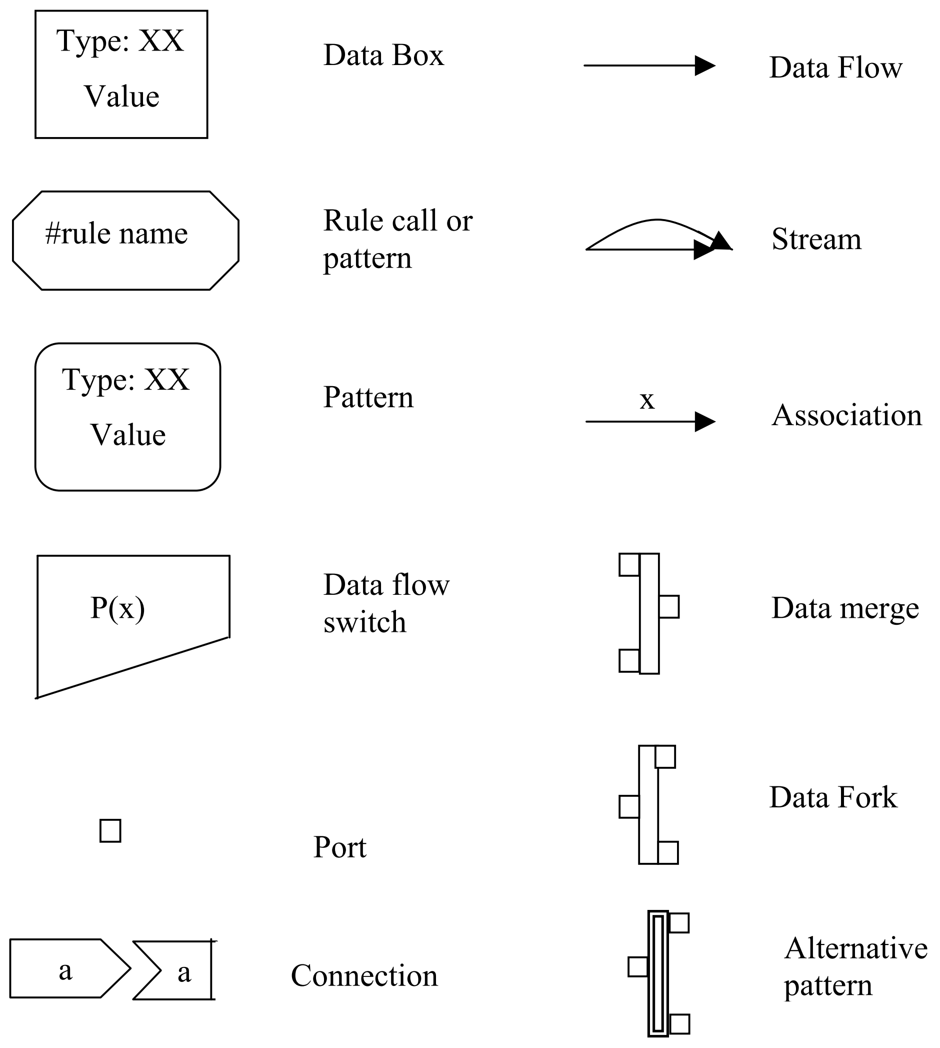


Figure 1. Basic icons for the V meta-programming language (After: Visual Meta-Programming Language[6])

the Boolean expression in the switch evaluates to true, the true output ports receive the data flow otherwise the false output ports transfer the flow. The input ports can be named and the names can appear in the expression. Connection nodes only have a single port associated with their type either input or output. These ports allow connections to cross the diagram without cluttering the screen. Data merge nodes have two or more input ports and one output port. Data forks are the inverse. The alternative pattern is a data fork that must have pattern nodes connected to its output ports.

2. Connection Icons

Each of the three connection icons must begin at an output port and end at an input port. The data flow connection represents simple data flow delivering one data item at a time. A stream contains an unbounded sequence of data items and can deliver as many as needed. An association contains an association name and creates a named link between two data objects or patterns.

THIS PAGE INTENTIONALLY LEFT BLANK

III. REQUIREMENTS AND DESIGN

A. REQUIREMENTS

Requirements for this project follow directly from the need for a GUI to demonstrate the effectiveness of the V language and were identified though discussions with the author of the V compiler. Initial non-functional requirements were to provide the interface on multiple OS platforms including Unix and Windows, and to provide for future work through ease of maintainability. Initial high level functional requirements for this project were to provide a Graphical Interface to the user allowing him to manipulate V programs on screen and output them in a format usable by the compiler.

1. Non-Functional Requirements

Two main non-functional requirements exist for this system. The first requirement is for cross platform compatibility. Typical users of the VIDE system are expected to work alone or in small groups on platforms ranging from Unix systems to Win98/WinNT/Win2000. Future platforms are expected to include future variants of Unix/Linux and Windows. To meet these needs, the program must be available for use on these systems with no modifications to the source code.

The second major non-functional requirement is to ensure the program is easily maintainable. The V language is still under evaluation and elements of it may change frequently. Because of this volatility in the language, the program must be easily modifiable to support changes in the language. Further work on the Integrated Development Environment will also require the program to be easily modifiable to support additions and changes to the functionality required of the program. Since maintaining and upgrading this system will be performed by individuals not involved in its initial development, it must be easy to read and maintain.

2. Functional Requirements

The two main functional requirements identified during the interview process were expanded to produce lower level requirements and a third set of future requirements was identified for consideration. The first requirement for a graphical user interface that allows the user to represent programs in the visual meta-programming language breaks down into several sub-requirements. The program must provide the user with a range of objects to choose from which represent data flow elements. The set of objects must allow the user to represent data, processes, and the dependencies between them. It must allow him to place the chosen elements on a workspace and connect them together in order to logically represent the desired functionality. All element objects in the workspace should be editable and movable. The system must display the objects with their relationships in a manor that minimizes confusion and interference. These requirements and their components are summarized in table 1.

REF#	DESCRIPTION	PRIORITY
1	Provide Graphical Interface to User	Critical
1.1	Display Graphical Information on Screen	Critical
1.1.1	Display Window Interface	Critical
1.1.1.1	Display Menu Items	Critical
1.1.1.1.1	Display File operations	Critical
1.1.1.1.2	Display Edit operations	Critical
1.1.1.1.3	Display View operations	Useful
1.1.1.1.4	Display Run operations	Critical
1.1.1.1.5	Display Window operations	Useful
1.1.1.1.6	Display Help operations	Useful
1.1.1.2	Display Tool Buttons for quick item selection	Critical
1.1.1.2.1	Display object buttons	Critical
1.1.1.2.2	Display flow buttons	Critical
1.1.1.3	Display Window	Critical
1.1.1.3.1	Display buttons	Critical
1.1.1.3.2	Display scroll bar	Critical
1.1.1.3.3	Display Title	Important
1.1.2	Display workspace for user	Critical
1.1.2.1	Draw objects in workspace	Critical
1.1.2.1.1	Draw shape	Critical
1.1.2.1.2	Draw ports	Critical
1.1.2.1.3	Draw info	Critical
1.1.2.2	Draw data flow items in workspace	Critical
1.1.2.2.1	Draw/make connections	Critical
1.1.2.2.2	Draw route	
1.1.3	Display cursor	Critical
1.2	The system must allow the user to edit the diagram.	Critical
1.2.1	The system must store user diagram for access	Critical
1.2.1.1	The system must store object's information internally	Critical
1.2.1.1.1	The system must store the type/shape of each object	Critical
1.2.1.1.2	The system must store the input ports and connections	Critical
1.2.1.1.3	The system must store the output ports and connections	Critical
1.2.1.1.4	The system must store the function of the object	Critical
1.2.1.1.5	The system must store the location of the object	Important
1.2.1.1.6	The system must manage the size of the object	Critical
1.2.1.2	The system must store the connections between objects	Critical
1.2.1.2.1	The system must store the type and shape of connection	Critical
1.2.1.2.2	The system must store the origin of the connection	Critical
1.2.1.2.3	The system must store the destination of the connection	Critical
1.2.1.2.4	The system must store waypoints describing the connection's route	Useful
1.2.1.3	The system must keep track of the environment of the diagram	Critical
1.2.1.3.1	The system must store the name of the diagram	Critical
1.2.1.3.2	The system must store the input ports of the diagram	Critical
1.2.1.3.3	The system must store the output ports of the diagram	Critical
1.2.1.3.4	The system must store the intermediate text file for conversion	Important

1.2.2	The system must allow the user to select elements for editing	Critical
1.2.2.1	The system must allow the user to select objects	Critical
1.2.2.2	The system must allow the user to select connections	Important
1.2.2.3	The system must allow the user to select groups of objects and/or connections	Useful
1.2.2.4	The system must allow the user to select ports	Critical
1.2.2.5	The system must allow the user to select waypoints	Useful
1.2.3	The system must allow the user to delete selected elements	Critical
1.2.3.1	The system must remove items selected for deletion from the screen	Critical
1.2.3.2	The system must remove references to deleted items from other objects	Critical
1.2.4	The system must allow the user to copy selected elements	Useful
1.2.4.1	The system must create duplicate objects of selected objects	Useful
1.2.4.2	The system must differentiate between the copies and originals	Useful
1.2.4.3	The system must show the new objects on the screen	Useful
1.2.5	The system must allow the user to move selected elements	Critical
1.2.5.1	The system must redisplay the element in its new location	Critical
1.2.5.2	The system must redistribute connected items to maintain consistency	Critical
1.2.6	The system must allow the user to change connections	Critical
1.2.6.1	The system must allow the user to disconnect one end of a connection from a port	Useful
1.2.6.2	The system must allow the user to reconnect a connection to other ports	Useful
1.2.7	The system must allow the user to change names and expressions of objects	Important
1.2.8	The system must allow the user to undo changes	Useful
1.2.8.1	The system must keep a history of changes	Useful
1.2.8.2	The system must be able to reset the last change	Useful

Table 1. Graphical User Interface Requirements

The second requirement for communication with the program generator or compiler also has several components. The system must be able to export the complete diagram in a format readable by the program generator. This format must maintain all dependencies and functionality of the displayed diagram with no distortion. The system must be able to store the program for future work and redisplay it without losing any accuracy. These requirements and their components are summarized in table 2.

REF#	**DESCRIPTION**	**PRIORITY**
2	The system must be able to import and export the complete diagram in a format readable by the program generator.	Critical
2.1	Export user diagram in text format.	Critical
2.1.1	Open output to write text.	Critical
2.1.2	Translate user program to text.	Critical
2.1.2.1	Map objects to text function descriptor.	Critical
2.1.2.2	Map connections to object parameters	Critical
2.1.3	Send text to output and close.	Critical
2.2	Import diagram from text format.	Important
2.2.1	Open file for input.	Important
2.2.2	Read text from file	Important
2.2.3	Generate program from text.	Important
2.2.3.1	Map text function to object	Important
2.2.3.2	Map text function parameters to object connections	Important
2.2.4	Display program.	Important

Table 2. Communication Requirements

In future versions, the system must be able to communicate with its clients, specifically the program generator. It must be able to graphically represent data values at each element while the program generator runs values through the diagram. It should be able to run at different speeds, from step by step, to the speed of the program generator. It should allow the user to inspect and change values at different points in the run. The system should be able to back up step by step while conducting a run. The system should allow the user to display and edit the text representation of the program. These considerations are summarized in Table 3.

REF#	**DESCRIPTION**	**PRIORITY**
3	The system must be able to interact directly with the program generator.	Useful
3.1	The system must be able to communicate to the generator	Useful
3.1.1	Start the generator	Useful
3.1.2	Send individual component data to the generator	Useful
3.1.3	Receive information from the generator	Useful
3.2	The system must be able to show debugging information from the generator	Useful
3.2.1	The system must show code generated by the generator	Useful
3.2.2	The system must allow user to inspect code and values at points in diagram	Useful
3.2.3	The system must allow the user to run the program	Useful
3.2.3.1	The user must be able to insert values for the program to use	Useful
3.2.3.2	The system must be able to step through the program	Useful
3.2.3.3	The system must be able to run the program to breakpoints	Useful
3.2.3.4	The system must be able to modify points in the run	Useful
3.2.3.5	The system must be able to back up during a step by step run	Useful
3.2.4	The system must be able to display and edit the text representation of the diagram	Useful

Table 3. Future Requirements

3. Use Cases

Once the requirements were determined, extended use cases were developed to examine system behavior in common usage. These use cases form the basis for design and provide a behavioral association with the requirements. Each use case contains a brief description of the behavior, the basic flow of events, a list of requirements the use case affects, and alternate flow when applicable.

USE CASE NAME:	**NEW DIAGRAM**		
DESCRIPTION:	This Use Case describes the flow of events involved in the creation of a new diagram.		
REQUIREMENTS:	**1.1.1.1.1, 1.1.1.3, 1.1.2.1.2, 1.2.1.3,**		
BASIC FLOW:			
	User		System
1	The user enters diagram name and number of input and output ports in the edit bar.	1	The system displays and stores the information.
2	The user selects the drop down file menu using the mouse or hotkey.	2	The system displays the drop down menu.
3	The user selects New from the menu using the mouse or hotkey	3a	The system clears the workspace
		3b	The system creates a new diagram with the specified name

			and number of ports
		3c	The system displays the name and ports in the workspace
ALTERNATE FLOW 1: NO NAME ENTERED			
	User		System
2	The user selects the drop down file menu using the mouse or hotkey.	2	The system displays the drop down menu.
3	The user selects New from the menu using the mouse or hotkey	3a	The system clears the workspace
		3b	The system creates a new diagram with the specified number of ports and the default name “Untitled”
		3c	The system displays the name and ports in the workspace
ALTERNATE FLOW 2: NO PORTS ENTERED			
	User		System
2	The user selects the drop down file menu using the mouse or hotkey.	1	The system displays the drop down menu.
3	The user selects New from the menu using the mouse or hotkey	3a	The system clears the workspace
		3b	The system creates a new diagram with the specified name, two input ports and one output port
		3c	The system displays the name and ports in the workspace
Preconditions:	The program must be running normally. No dialog boxes can be open.		
Postconditions:	No dialog boxes are open. A blank diagram with the specified number of input and output ports along with specified name is displayed in the workspace.		

Table 4. Use Case 01: New Diagram

USE CASE NAME:	**OPEN DIAGRAM**		
DESCRIPTION:	This use case describes the flow of events involved in opening a file and loading it’s diagram.		
REQUIREMENTS:	**1.1.1.1.1, 1.1.1.3, 1.1.2, 2.2**		
BASIC FLOW:			
	User		System
1	The user selects the drop down file menu using the mouse or hotkey.	1	The system displays the drop down menu.
2	The user selects Open from the menu using the mouse or hotkey	2	The system generates a file selection dialog box
3	The user enters or selects a filename and hits ok	3	The system checks for a valid filename
		4	The system saves the information and closes the dialog box
		5	The system opens the file
		6	The system reads the project name and ports
		7	The system clears the workspace
		8	The system creates a new diagram with the specified name and ports
		9	The system displays the name and ports in the workspace
		10	For each node in the file the system adds the corresponding node to the diagram
		11	For each connection in the file the system adds the corresponding connection to the diagram
		12	The system closes the file
ALTERNATE FLOW 1: CANCEL			
	User		System
1	The user selects the drop down file menu using the mouse or hotkey.	1	The system displays the drop down menu.
2	The user selects Open from the menu using the mouse or hotkey	2	The system generates a file selection dialog box
3	The user hits cancel on the dialog box	3	The system closes the dialog box
ALTERNATE FLOW 2: INVALID NAME ENTERED			
	User		System
1	The user selects the drop down file menu using the mouse or hotkey.	1	The system displays the drop down menu.
2	The user selects Open from the menu using the mouse or hotkey	2	The system generates a file selection dialog box
3	The user hits ok or enters a filename and hits ok	3a	The system checks for a valid filename
		3b	The system generates a pop-up box with the message “Invalid file name”

4	The user clicks on ok or hits enter on the pop-up box	4	The system returns to the file selection dialog box
5	The user continues from line 3 in the basic or alternate flow sequence		
ALTERNATE FLOW 3: ERROR IN FILE			
	User		System
1	The user selects the drop down file menu using the mouse or hotkey.	1	The system displays the drop down menu.
2	The user selects Open from the menu using the mouse or hotkey	2	The system generates a file selection dialog box
3	The user enters or selects a filename and hits ok	3	The system checks for a valid filename
		4	The system saves the information and closes the dialog box
	File	5	The system opens the file
6	The file can not be read	6a	The system displays the error message in a pop-up box requiring user to hit okay
		6b	The system closes the file and continues from line 1
			or
		6	The system reads the project name and ports
		7	The system clears the workspace
		8	The system creates a new diagram with the specified name and ports
		9	The system displays the name and ports in the workspace
10	The end of the file is corrupted	10	The system displays the error message in a pop-up box requiring user to hit okay
		10	The system closes the file and continues normal operation with the diagram current to the point where the file failed.

Preconditions:	The program must be running normally. No dialog boxes can be open. A file containing properly formatted data is available to be opened.
Postconditions:	No dialog boxes are open. A diagram containing all the information from the file is displayed in the workspace. The file is closed. The filename is saved.

Table 5. Use Case 02: Open Diagram

USE CASE NAME:	**SAVE DIAGRAM**
DESCRIPTION:	This Use Case describes the flow of events involved in saving a diagram to a file.
REQUIREMENTS:	**1.1.1.1.1, 1.1.1.3, 1.1.2, 2.1**
BASIC FLOW:	

	User		System
1	The user selects the drop down file menu using the mouse or hotkey.	1	The system displays the drop down menu.
2	The user selects Save from the menu using the mouse or hotkey	2	The system checks for a valid filename associated with the diagram
		3	The system opens the file
		4	The system writes the diagram name and port information to the file in the appropriate format
		5	For each node in the diagram the system writes the corresponding text representation to the file
		6	For each connection in the diagram the system writes the corresponding text representation in the file
		7	The system closes the file
ALTERNATE FLOW 1: NO FILE NAME EXISTS			
	User		System
1	The user selects the drop down file menu using the mouse or hotkey.	1	The system displays the drop down menu.
2	The user selects Save from the menu using the mouse or hotkey	2	The system checks for a valid filename associated with the diagram
		3	The system runs the Save As use case 04 from line 2
	ALTERNATE FLOW 2: ERROR IN FILE		
	User		System
1	The user selects the drop down file menu using the mouse or hotkey.	1	The system displays the drop down menu.
2	The user selects Save from the menu using the mouse or hotkey	2	The system checks for a valid filename associated with the diagram
	File	3	The system opens the file
4	The file has an error	4	The system closes the file

		5	The system displays the error message in a pop-up box requiring user to hit okay
		6	The system returns to the state defined by the preconditions
Preconditions:	The program must be running normally. No dialog boxes can be open. A filename is associated with the existing diagram. A diagram including at least a name, input and output ports is displayed in the workspace		
Postconditions:	No dialog boxes are open. The text representation of the diagram displayed in the workspace is saved in the file referred to by the filename. The file is closed. The filename is saved.		

Table 6. Use Case 03: Save Diagram

USE CASE NAME:	**SAVE DIAGRAM AS FILENAME**		
DESCRIPTION:	This Use Case describes the flow of events involved in saving a diagram to a file specified by the user.		
REQUIREMENTS:	**1.1.1.1.1, 1.1.1.3, 1.1.2, 2.1**		
BASIC FLOW:			
	User		System
1	The user selects the drop down file menu using the mouse or hotkey.	1	The system displays the drop down menu.
2	The user selects Save As from the menu using the mouse or hotkey	2	The system generates a file selection dialog box
3	The user enters or selects a filename and hits ok	3	The system checks for a valid filename
		4	The system saves the information and closes the dialog box
		5	The system associates the filename with the diagram
		6	The system continues from use case 03 Save diagram line 3
ALTERNATE FLOW 1: CANCEL			
	User		System
1	The user selects the drop down file menu using the mouse or hotkey.	1	The system displays the drop down menu.
2	The user selects Save As from the menu using the mouse or hotkey	2	The system generates a file selection dialog box
3	The user hits cancel on the dialog box	3	The system closes the dialog box and returns to operation in the state consistent with the precondition
ALTERNATE FLOW 2: INVALID NAME ENTERED			
	User		System
1	The user selects the drop down file menu using the mouse or hotkey.	1	The system displays the drop down menu.
2	The user selects Save As from the menu using the mouse or hotkey	2	The system generates a file selection dialog box
3	The user hits ok or enters a filename and hits ok	3a	The system checks for a valid filename
		3b	The system generates a pop-up box with the message "Invalid file name"
4	The user clicks on ok or hits enter on the pop-up box	4	The system returns to the file selection dialog box
5	The user continues from line 3 in the basic or alternate flow sequence		
ALTERNATE FLOW 3: ERROR IN FILE			
	User		System
1	The user selects the drop down file menu using the mouse or hotkey.	1	The system displays the drop down menu.
2	The user selects Save As from the menu using the mouse or hotkey	2	The system generates a file selection dialog box
3	The user enters or selects a filename and hits ok	3	The system checks for a valid filename
		4	The system saves the information and closes the dialog box
	File	5	The system opens the file
6	The file can not be written	6a	The system displays the error message in a pop-up box requiring user to hit okay
		6b	The system closes the file and continues normal operation in a state consistent with the preconditions
Preconditions:	The program must be running normally. No dialog boxes can be open. A diagram including at least a name, input and output ports is displayed in the workspace		
Postconditions:	No dialog boxes are open. The text representation of the diagram displayed in the workspace is saved in the file referred to by the filename. The file is closed. The filename is saved.		

Table 7. Use Case 04: Save Diagram As…

USE CASE NAME:	**EXIT SYSTEM**		
DESCRIPTION:	This Use Case describes the flow of events involved exiting the program		
REQUIREMENTS:	**1.1.1.1.1**		
BASIC FLOW:			
	User		System
1	The user selects the drop down file menu using the mouse or hotkey.	1	The system displays the drop down menu.
2	The user selects Exit from the menu using the mouse or hotkey	2	For each diagram open in the system create a pop-up box asking if the user wishes to save that diagram
3	The user hits ok	3a	The system runs use case 03: Save, from line 2 for each ok response from user
		3b	The system deletes the current diagram from the workspace and memory
		4	The system shuts down normally
ALTERNATE FLOW 1: NO SAVE			
	User		System
1	The user selects the drop down file menu using the mouse or hotkey.	1	The system displays the drop down menu.
2	The user selects Exit from the menu using the mouse or hotkey	2	For each diagram open in the system create a pop-up box asking if the user wishes to save that diagram
3	The user hits no	3	The system deletes the current diagram from the workspace and memory
		4	The system shuts down normally
ALTERNATE FLOW 2: CANCEL			
	User		System
1	The user selects the drop down file menu using the mouse or hotkey.	1	The system displays the drop down menu.
2	The user selects Exit from the menu using the mouse or hotkey	2	For each diagram open in the system create a pop-up box asking if the user wishes to save that diagram
3	The user hits cancel	3	The system returns to normal operation in accordance with the preconditions
ALTERNATE FLOW 3: WINDOW SHUTDOWN			
	User		System
1	The user selects the Close button on the window	1	The system continues from line 2 of the basic or alternate flows in this use case
Preconditions	The program must be running normally. No dialog boxes can be open.		
Postcondition	The system is no longer running.		

Table 8. Use Case 05: Exit

USE CASE NAME:	**CUT/COPY/DELETE**		
DESCRIPTION:	This use case describes the flow of events involved in cutting, copying or deleting a node.		
REQUIREMENTS:	**1.1.1.1.2, 1.1.1.3, 1.2.1.1, 1.2.1.3, 1.2.2, 1.2.3, 1.2.4**		
BASIC FLOW:			
	User		System
1	The user selects the drop down edit menu using the mouse or hotkey.	1	The system displays the drop down menu.
2	The user selects Copy from the menu using the mouse or hotkey	2	The system saves the contents of the selected list to the clipboard
ALTERNATE FLOW 1: DELETE			
	User		System
1	The user selects the drop down edit menu using the mouse or hotkey.	1	The system displays the drop down menu.
2	The user selects Delete from the menu using the mouse or delete key	2	The system copies all connections to elements of the selected list to the undo list
		3	The system deletes all connections to elements of the selected list from the workspace
		4	The system saves the selected list to the undo list
		5	The system deletes all items in the selected list from the workspace
		6	The system clears the selected list

ALTERNATE FLOW 2: CUT				
	User		System	
1	The user selects the drop down file menu using the mouse or hotkey.	1	The system displays the drop down menu.	
2	The user selects Cut from the menu using the mouse or hotkey	2	The system executes the basic flow from line 2	
		3	The system executes alternate flow 1 from line 2	
Preconditions:	Use case 06: select item must have successfully completed. The program must be running normally. No dialog boxes can be open. A diagram including at least input and output ports and a name is displayed in the workspace. The select button is selected on the toolbar. The item in the selected list and all its component parts are highlighted.			
Postconditions:	No dialog boxes are open. The select button is selected on the toolbar. The program is running normally. Copy: The item formally selected by the user and all its component parts are highlighted. The highlighted items are represented on the clipboard. The highlighted items are in the selected list. Cut: The formally highlighted items are represented on the clipboard. The formally highlighted items are not displayed in the workspace. The formally highlighted items are deleted from the diagram. The formally highlighted items and all links to them are represented on the undo list. The formally highlighted items are not in the selected list. Delete: The formally highlighted items are not displayed in the workspace. The formally highlighted items are deleted from the diagram. The formally highlighted items and all links to them are represented on the undo list. The formally highlighted items are not in the selected list			

Table 9. Use Case 06: Cut/Copy/Delete

USE CASE NAME:	**SELECT ITEM**			
DESCRIPTION:	This Use Case describes the flow of events involved in selecting an item for editing.			
REQUIREMENTS:	**1.1.1, 1.1.2, 1.2.2**			
BASIC FLOW:				
	User		System	
1	The user left clicks on an item in the workspace	1	The system checks the item for type	
		2	The system adds the item to the selected list	
		3	The system highlights all elements in the selected list	
ALTERNATE FLOW 1: ITEM IS A NODE				
	User		System	
1	The user left clicks on an item in the workspace	1a	The system checks the item for type	
		1b	The system adds all nodes and ports contained in the node clicked on to the selected list	
		2	The system continues with the basic flow from line 2	
ALTERNATE FLOW 2: A NODE IS ALREADY SELECTED				
	User		System	
1	The user left clicks on an item in the workspace	1a	The system empties the selected list	
		1b	The system continues with basic or alternate flow from line 1	
Preconditions:	The program must be running normally. No dialog boxes can be open. A diagram including at least input and output ports and a name is displayed in the workspace. The select button is selected on the toolbar			
Postconditions:	The program must be running normally. No dialog boxes can be open. A diagram including at least input and output ports and a name is displayed in the workspace. The select button is selected on the toolbar. The item selected by the user and all its component parts are highlighted.			

Table 10. Use Case 07: Select

USE CASE NAME:	**PASTE**			
DESCRIPTION:	This Use Case describes the flow of events involved in pasting a node into the workspace			
REQUIREMENTS:	**1.1.1.1.2, 1.1.1.3, 1.1.2, 1.2.1.1, 1.2.1.3, 1.2.2, 1.2.3, 1.2.4**			
BASIC FLOW:				
	User		System	
1	The user selects the drop down edit menu using the mouse or hotkey.	1	The system displays the drop down menu.	
2	The user selects Paste from the menu using the mouse or hotkey	2	The system runs the use case 10: add item from line 3 using the information from the clipboard	

Preconditions:	Use case 07: cut/copy must have successfully completed. The program must be running normally. No dialog boxes can be open. A diagram including at least input and output ports and a name is displayed in the workspace. The select button is selected on the toolbar. One or more items are represented on the clipboard
Postconditions:	The program must be running normally. No dialog boxes can be open. A diagram including at least a name, input and output ports, and the contents of the clipboard is displayed in the workspace. The select button is selected on the toolbar. One or more items are represented on the clipboard

Table 11. Use Case 08: Paste

USE CASE NAME:	**SELECT NODE/CONNECTOR TYPE**		
DESCRIPTION:	This Use Case describes the flow of events involved in selecting the current active element type for adding elements to the workspace		
REQUIREMENTS:	**1.1.1.2, 1.1.1.3**		
BASIC FLOW:			
	User		System
1	The user left clicks on one of the tool bar buttons	1	The system toggles on the button in the toolbar corresponding to the button pressed
		2	The system toggles off the button in the toolbar corresponding to the active element
		3	The system replaces the active element type with the type selected
ALTERNATE FLOW 1: BUTTON ALREADY PRESSED			
	User		System
1	The user left clicks on the tool bar button already toggled on	1	The system resets the edit bar
Preconditions:	The program must be running normally. No dialog boxes can be open.		
Postconditions:	No dialog boxes are open. The button clicked on is toggled on. No other button is toggled on. The type associated with the button that was clicked is set as the active type.		

Table 12. Use Case 09: Select Node/Connector Type

USE CASE NAME:	**ADD NODE**		
DESCRIPTION:	This Use Case describes the flow of events involved in adding a node to the workspace		
REQUIREMENTS:	**1.1.1.2.1, 1.1.1.3, 1.1.2.1, 1.1.3, 1.2.1.1, 1.2.1.3**		
BASIC FLOW:			
	User		System
1	The user enters number of ports, expression, and other information as applicable into the edit bar	1	The system shows the information in the edit bar
2	The user left clicks on an open space in the workspace	2	The system creates a node with the appropriate type, ports and expression and adds it to the diagram
		3	The new node is displayed at the cursor location from the initial click with the rest of the workspace
ALTERNATE FLOW 1: CLICK ON ITEM			
	User		System
2	The user left clicks on an item in the workspace	2	The system creates a node with the appropriate type, ports and expression and adds it as a child to the node clicked on
		2	The system displays the new node inside its parent and resizes the parent as necessary
ALTERNATE FLOW 2: USER DOESN'T ENTER ALL ITEMS			
	User		System
2	The user left clicks on an open space in the workspace	2	The system creates a node with the appropriate type. The default number of ports and expression are used and the node is added to the diagram
		3	The new node is displayed at the cursor location from the initial click with the rest of the workspace
Preconditions:	The program must be running normally. No dialog boxes can be open. Use case 09 must have been successfully completed. A Node type button is toggled. A Node type is set as the active type. A diagram including at least input and output ports and a name is displayed in the workspace.		
Postconditions:	No dialog boxes are open. A diagram containing all the information in the precondition plus the node defined above is displayed in the workspace.		

Table 13. Use Case 10: Add Node

USE CASE NAME:	**ADD CONNECTION**		
DESCRIPTION:	This use case describes the flow of events involved in adding a connection between two ports.		
REQUIREMENTS:	**1.1.1.2.2, 1.1.1.3, 1.1.2.2, 1.1.3, 1.2.1.2, 1.2.1.3, 1.2.2.4, 1.2.2.5**		
BASIC FLOW:			
	User		System
1	The user left clicks on a port	1a	The system checks to make sure the port is available
		1b	The system starts a link from the clicked port to the cursor
2	The user clicks on one or more open areas on the workspace	2	For each click the system creates a waypoint at the clicked point and attaches the link to it forming a chain from the first port to the cursor
3	The user clicks on a port	3	The system finishes the connection between the two ports
		4	The system displays the entire workspace including the new connection
ALTERNATE FLOW 1: CLICK ON NON-PORT			
	User		System
1	The user left clicks on an open area in the workspace or on an node	1	The system ignores the click
Preconditions:	The program must be running normally. No dialog boxes can be open. Use case 09 must have been successfully completed. A Connector type button is toggled. A Connector type is set as the active type. A diagram including at least input and output ports and a name is displayed in the workspace.		
Postconditions:	No dialog boxes are open. A diagram containing all the information in the precondition plus the connection defined above is displayed in the workspace.		

Table 14. Use Case 11: Add Connection

B. DESIGN

1. Constraints

The non-functional requirements led to a design constraint on the language the system was to be implemented in. The requirement for maintainability, along with current state of practice indicated that an object-oriented approach should be taken. This allows for good separation of functionality, improving readability and logical consistency. Following the decision to take an object oriented approach was the decision to use an object-oriented language to implement the approach. An additional constraint on this decision came from the requirement for cross-platform compatibility. Java and GtkAda/GNAT support both the object-oriented approach and platform independent development and were considered for this project. Each provides the necessary graphic libraries to implement the user interface and drawing routines for the visual language.

Three factors influenced the selection of Ada for the language and GtkAda for the graphical toolkit. The first factor was the free availability of the software for windows and Linux/Unix. With little funding, the ability to freely download the GNAT Ada compiler and the GtkAda toolkit for both windows and Linux made the project quite affordable. The second factor was the strong typing and structure of Ada. These features reduced development time by reducing time spent locating potential programming errors. The last factor was the familiarity of the programmer with Ada and not with Java.

The final constraint on this project was resources. With a limited amount of time and only one programmer available, only the requirements listed as critical were selected for implementation.Other requirements were left for inclusion as time permitted or for follow-on work.

2. Architecture

In order to meet the extendibility and maintainability requirements, the model-view-presenter framework was chosen for this system.[8,9] This allows us to model the V language internally, separately from the representation of the view shown on the screen or printed out. Future changes to the model for efficiency or functionality are then partially independent of the representation on screen. It also allows for future work with multiple views of the represented diagram. For example, a text-based view of the diagram could easily be added without requiring any change to the existing model or view.

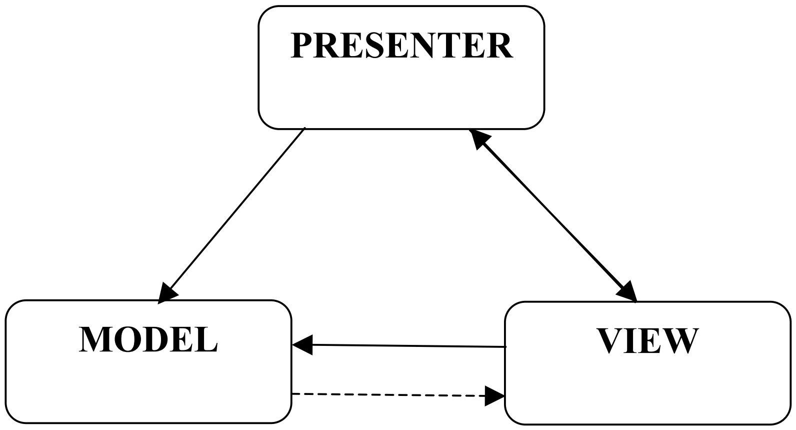


Figure 2. M-V-P relationship

In this program, the model contains the internal representation of the diagram under modification. Whenever it is modified, it notifies all of its views (at this time the only view implemented is the graphical one) that it has changed and each view then displays the changes as appropriate to the view. This separates the model from the rest of the program since it knows nothing about the other elements involved. It only has to respond to requests and changes in itself through its methods and then send notification to its list of observers that have registered an interest in the notification.

Each view acts as an observer of its associated model and allows for display of model information to the user as well as providing the user a way to interact with the model through the presenter. When the view is created, it registers with its model so that it can receive notification of changes. It then displays the model information in the format it is designed for and waits for the user or other outside actor to make changes. Actions on the screen are received by the view and passed to the presenter for interpretation. This allows the view to maintain cohesion by focusing on the display of information and leaving presentation and representation logic in other classes.

The presenter coordinates the creation of the view and manages presentation-related data separate from the model. Keeping track of inactive and active functionality in the presentation. When the user makes an action in the view, the presenter decides if it is an action on the model or just a display change. If it is change in the model, the presenter calls the appropriate methods in the model to make the change. If the change is in the way material should be presented to the user, the presenter calls a method in the view to make the appropriate change.

For this program, the model class and its associated classes represent the model in the MVP framework. The view is represented by the workspace class, and the presenter functionality resides in the main_window class and its associated helper classes. This delegation of responsibility allows modifications to be made to one area of functionality without causing major changes in the others. This interaction can be seen in figure 3.

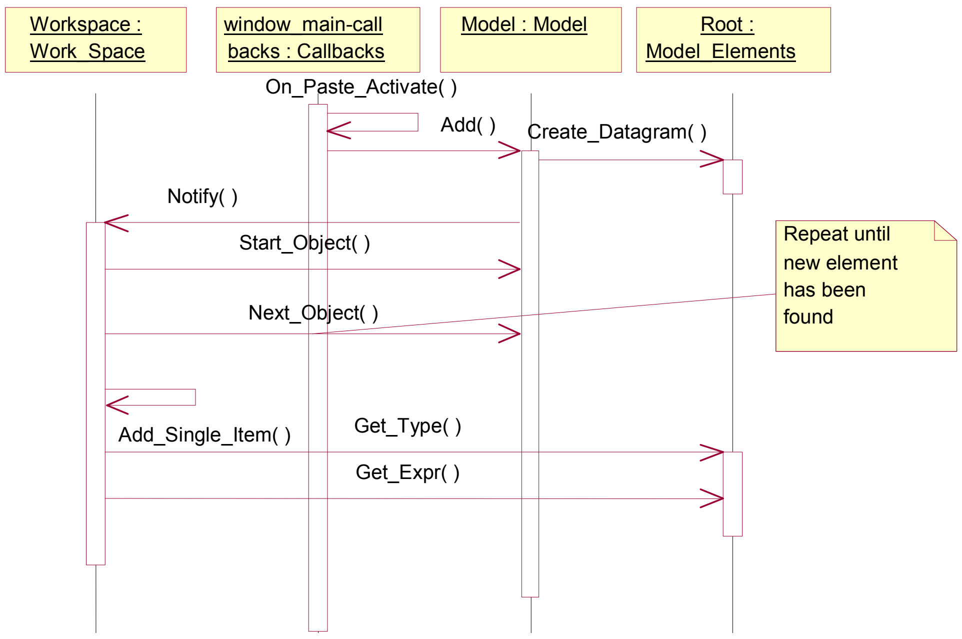


Figure 3. Sequence diagram for MVP interaction between classes in the V Graphical User Interface

IV. IMPLEMENTATION

A. ENVIRONMENT

The initial development platform for this project was Red Hat Linux 7.1, running on a Pentium II based desktop. Linux was the preferred development environment due to the drastically lower compile times experienced with it vs. the same machine running Windows98. Each completed build was also compiled on a Windows98 machine to verify compatibility. The program was written in Ada95 using the GNAT1.13p compiler. GtkAda v1.2.12 for Unix and v1.3.12 for Windows were the toolkits used for the graphics and windowing environment.[10,11,12] The initial GUI layout and main program elements were designed using GLADE 0.5.9, the GUI builder available with GtkAda. The program is known to run under Red Hat Linux 7.1, Windows98, WindowsNT, and Windows2000. It should also be able to run under the following platforms:

- Linux/x86
- Linux/sparc
- Linux/ppc
- Solaris/sparc
- Solaris/x86
- Dec Unix
- Tru64
- SGI IRIX 6.5
- HPUX
- NT 4.0
- Windows 2000
- AiX 4.3.2

- FreeBSD 3.2
- UnixWare 7.1

The GNU Visual Debugger (GVD), a graphical front-end for the GDB debugger provided with GNAT, was used for troubleshooting and unit testing.

B. PROGRAM

1. Model

The model section of the model-view-presenter framework is implemented in two main packages and a helper package. The central package to the model is the model package. This package provides the interface to the rest of the system and contains the diagram level information as well as a list of nodes and a list of connections. It also implements the observable interface from the observer pattern. Whenever the model changes, it notifies its observers about the change.

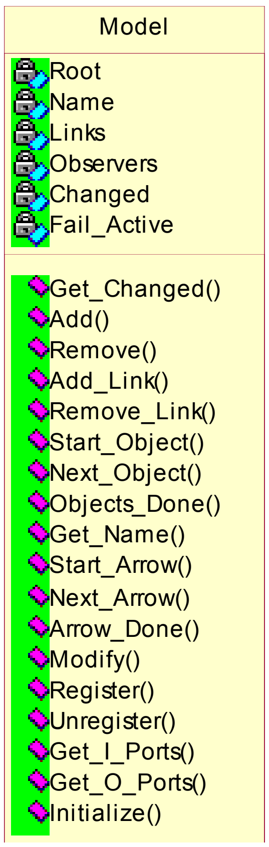


Figure 4. Model package

The second main package is the model_element package. This package defines the individual nodes and connections that make up the diagram. The nodes contain the node type, input and output connections, input port names, internal expressions, and children as appropriate to the node type. Connections contain the connection type, the connected nodes, the port numbers of the connected nodes, and an expression in the case of the association connector. The package also contains methods to read and to modify the attributes of the nodes and connectors.

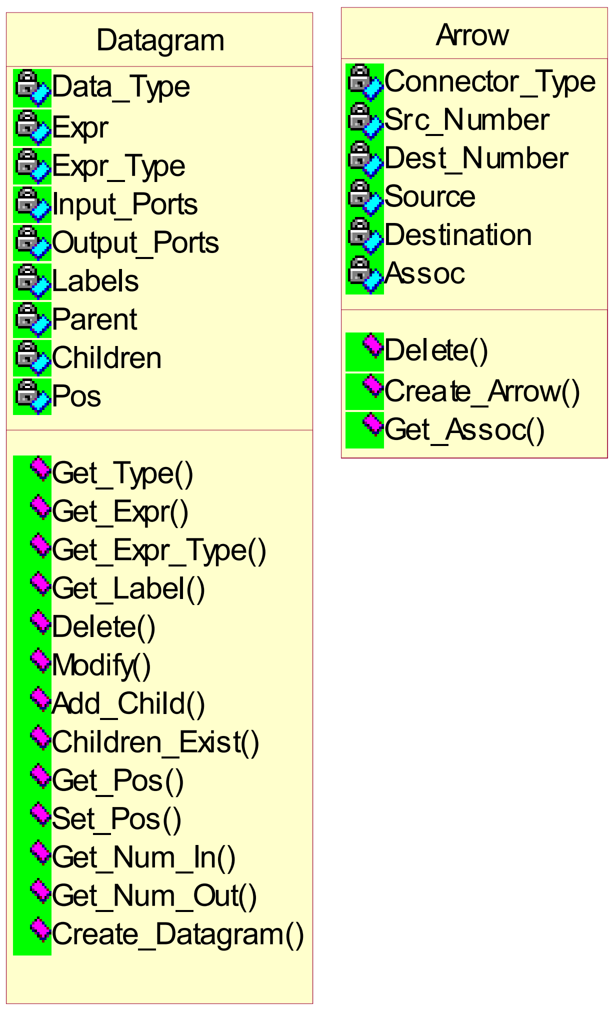


Figure 5. Model_elements package

The helper package print_out provides translation methods to produce formatted text output to a file, printer or screen. The format is compatible with the program generator that this project is designed to work with.

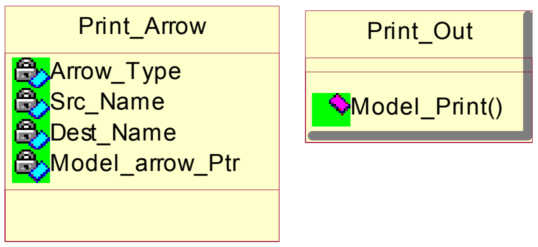


Figure 6. Print_out package

2. View

The packages that make up the diagram view include the workspace package and the helper package observer. The workspace package is one of the main packages in the system. It supports the graphical view of the model and forwards user input to the window_main package for presentation and model modification. It is based on the GtkAda canvas package[11], which provides basic canvas display, item movement, and connection routines. The workspace adds specific drawing routines for the model elements, item selection, highlighting, addition and removal. It also handles dynamic resizing of the items based on the number of children and ports. Each item in the workspace is associated with a model element and displays that element's information. The notify method lets the workspace know the model has been changed and that it needs to update itself.

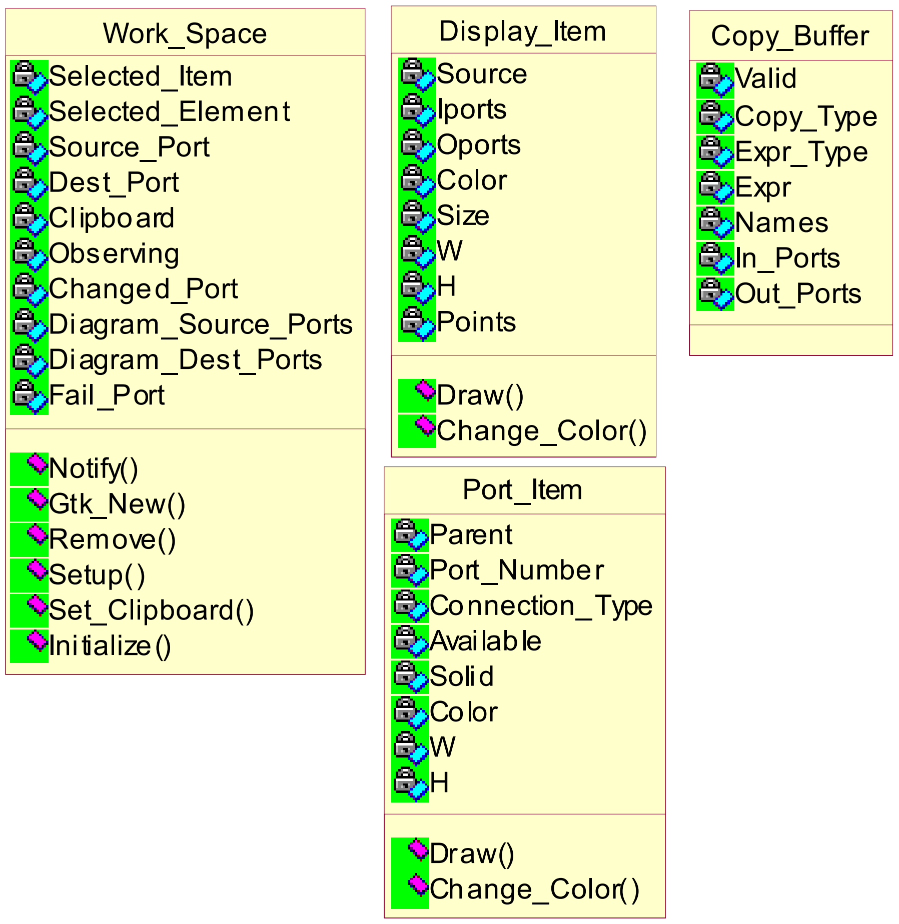


Figure 7. Workspace package

The observer package uses the façade pattern to provide the observer interface for interaction between the model and the views. Its methods allow the model to make notify calls to the façade, which then passes them on to the appropriate view. It also provides methods for the views to register and unregister with the model.

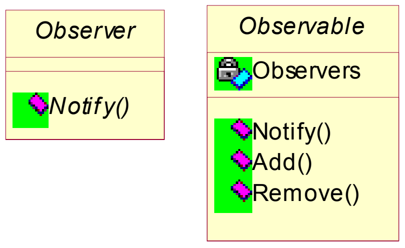


Figure 8. Observer_pattern package

3. Presenter

The rest of the packages in the system fall into the presenter area of the M-V-P framework. They provide the user interface allowing the user to make changes to the model and view and also provide feedback though the views on what functionality is available. The main program, package v_ide, was generated using the glade GUI developer. It provides the main program loop and top level window.

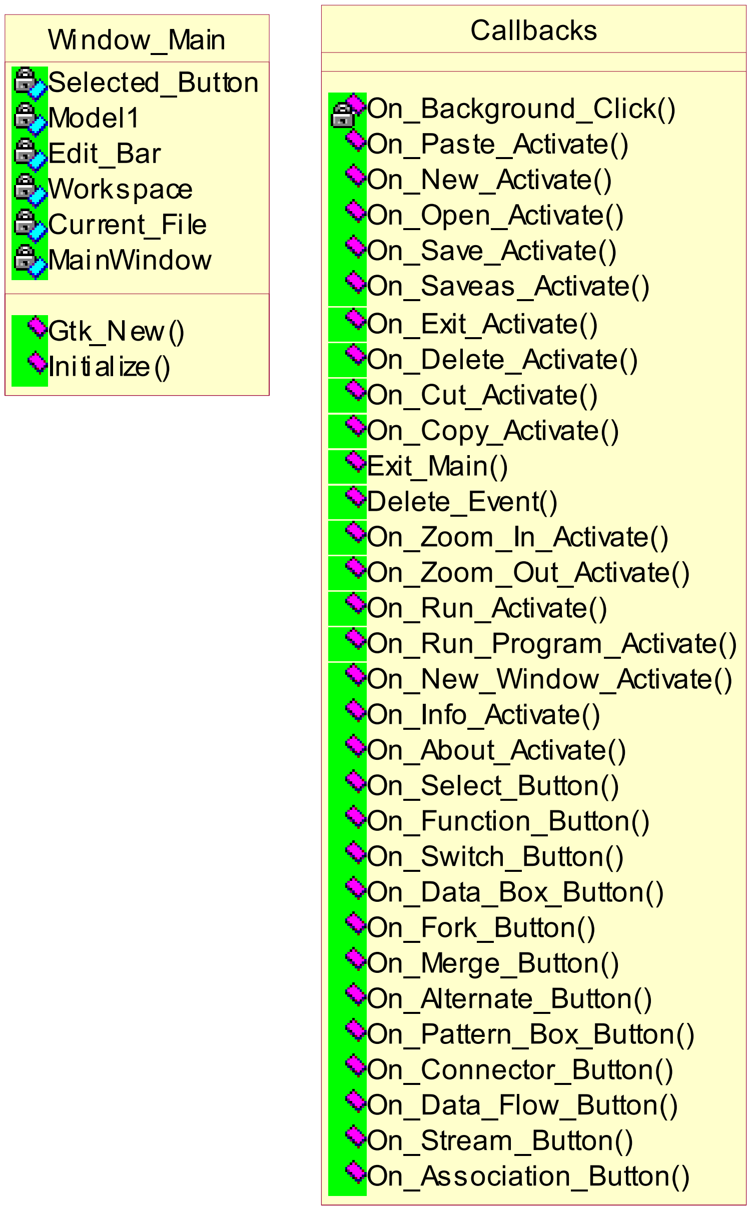


Figure 9. Window_main package

The second and largest package in the system is window_main. This package was partially developed using glade to provide the menu bar, tool button bar, and toplevel window functionality. Glade allows the user to arrange widgets such as buttons, menu items and window frames in the desired layout visually before generating the code to implement them. After this basic setup is generated, the programmer adds the appropriate functionality. In window_main and its associated callback package, this functionality includes the menu bar and its items, the tool button bar. The tool button bar identifies which functions are available in the workspace and menu bar and handles user input in the workspace in an appropriate manner.

Another package in the presenter is the edit bar. This package displays an edit bar to take user input for data to be added to the model. It provides this data to window_main whenever required by that package.

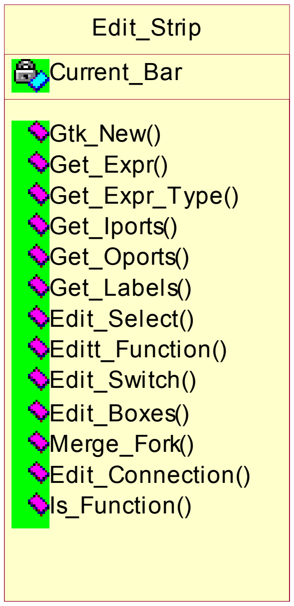


Figure 10. Edit_bar package

The final package in the project is the diagram package. This package provides the common data types used by the other packages. It describes the basic node and connector types available in the system.

C. USAGE

The graphical editor for the V language is made up of five main parts shown below in figure 11. The first part is the window containing the application. In the upper left is the program name, -VIDE, in the upper right are the standard window buttons for minimize, maximize, and exit. It is resizable to fit any dimensions desired by the user, although smaller sizes will not completely display some of the control bars.

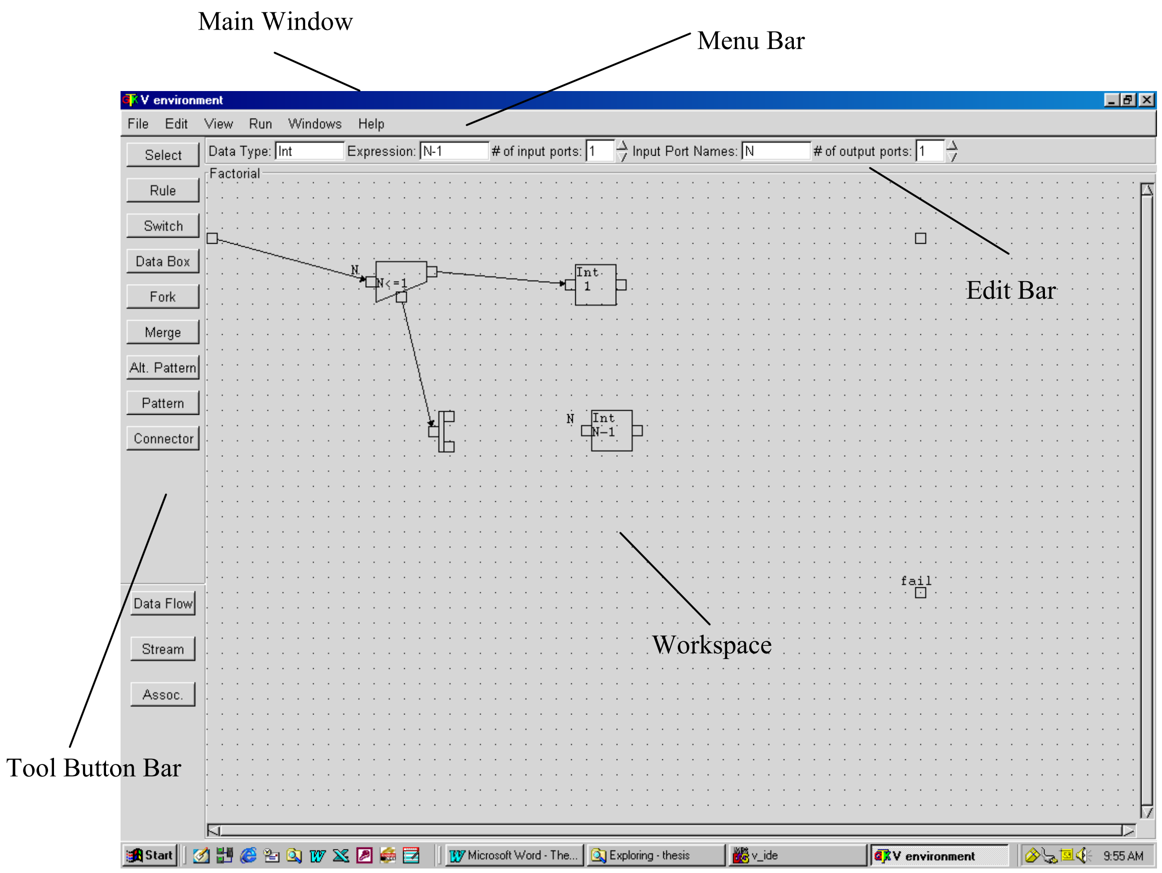


Figure 11. Graphical Editor Components

The next part of the application is the menu bar. The file menu contains options to create a new diagram, open an existing diagram, save a diagram and close the application. The edit menu allows the user to delete, cut, copy, or paste a selected node in an existing diagram. The view menu is not currently implemented but is reserved for changing the view of the current model, and for zoom control of the workspace. The run menu writes out the current model to a file with the same name as the diagram and an extension of .out. In future versions it is provided for control of the program generator and debugging. The window menu is not currently implemented and is reserved for working with multiple models. The help menu is not currently implemented.

The tool button bar provides quick access to the types of nodes and connectors available for use in the workspace. The select button allows the user to select nodes in the workspace so the user can move, edit, delete, cut or copy them. The node buttons allow the user to place a new node of the same type as the button in the workspace. Theconnection buttons allow the user to make a connection of the selected type between any two source and destination ports in the workspace.

The edit bar provides context sensitive entry items for the user to create or edit the nodes in the workspace. When a node or connection button is selected in the tool bar, the relevant data fields are displayed in the edit bar with the default values. When the select button is chosen in the tool bar, the diagram information is displayed for edit in the edit bar until a node or port is selected. When a node or port in the workspace is selected, the relevant fields are displayed in the edit bar for editing.

The workspace provides a place for the user to design a diagram in the V language. Nodes and connectors are added, edited, and deleted by clicking on the workspace with the appropriate information set in the control bars. Nodes can be rearranged in the workspace and the connections will automatically follow the nodes.

1. Creating a New Diagram

When the graphical editor starts up it does not have an active workspace and the edit bar contains the default information for a new diagram as shown in figure 12. To create a new diagram, the user enters the name, number of input ports, and number of output ports in the edit bar then selects new from the file menu. The workspace will become active as shown by the grid background, the diagram title in the upper left corner of the workspace, and the input, output, and fail ports along the edges of the diagram.

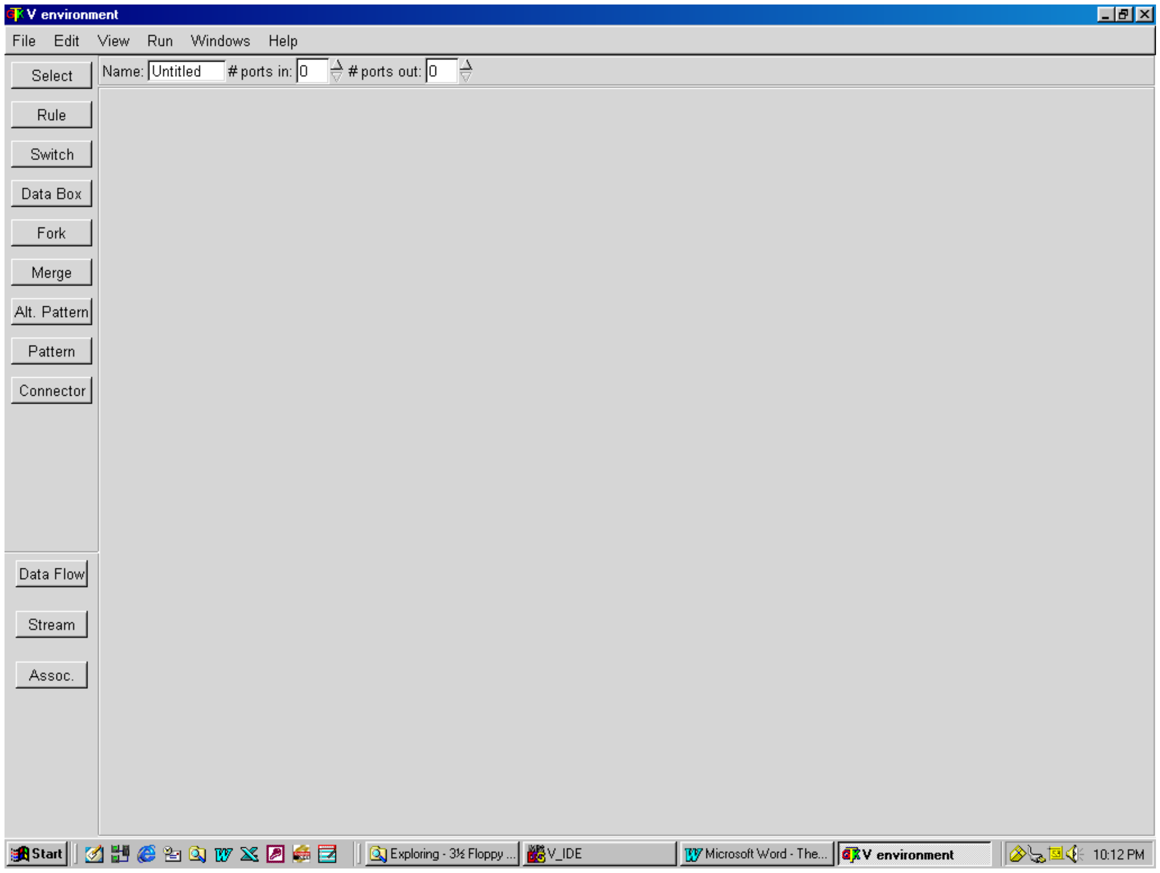


Figure 12. Initial state of the Graphical Editor after startup.

The example in figures 13 through 15 show the construction of the factorial function using the graphical editor and the V language. After the diagram is begun, the user clicks on the tool bar buttons to select the nodes needed for the factorial. After each tool button is selected, the edit bar information is entered and the user clicks on an empty area of the workspace. The new node is added to the diagram and appears where the user clicked.

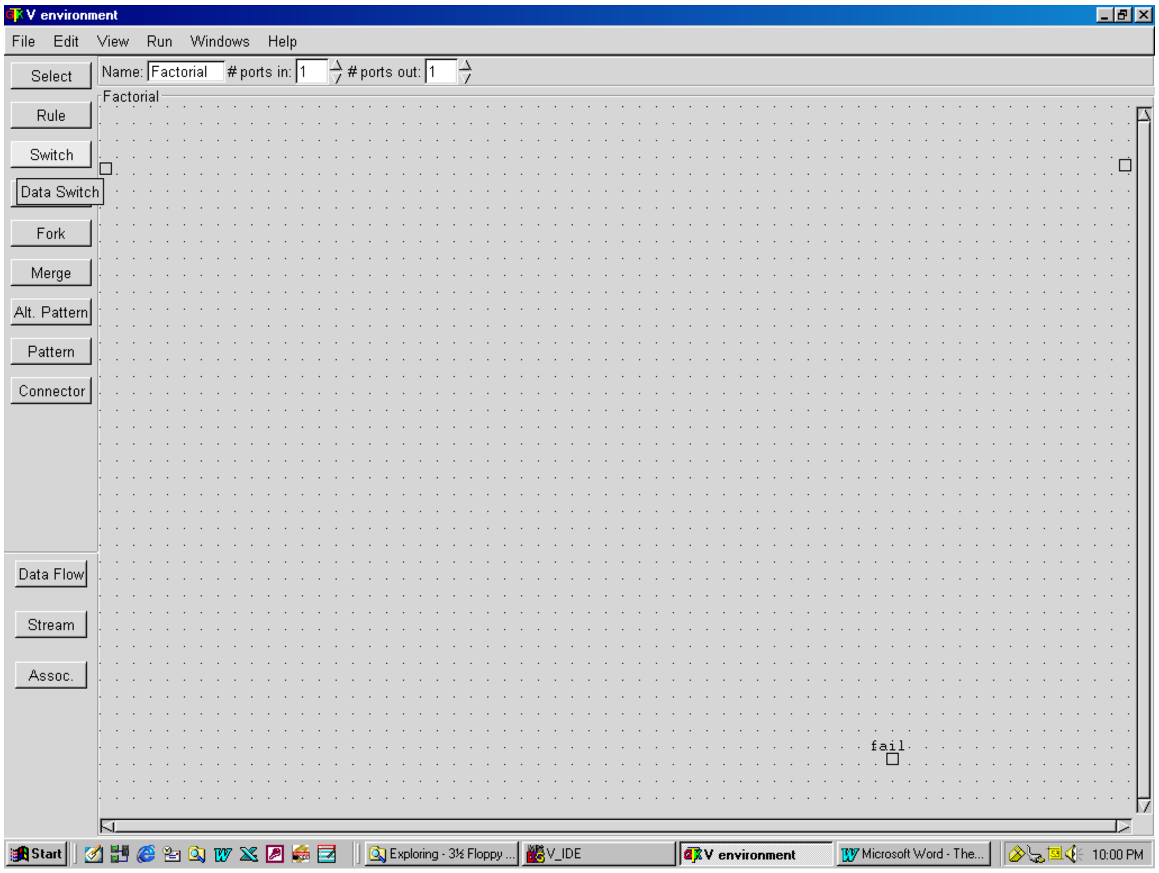
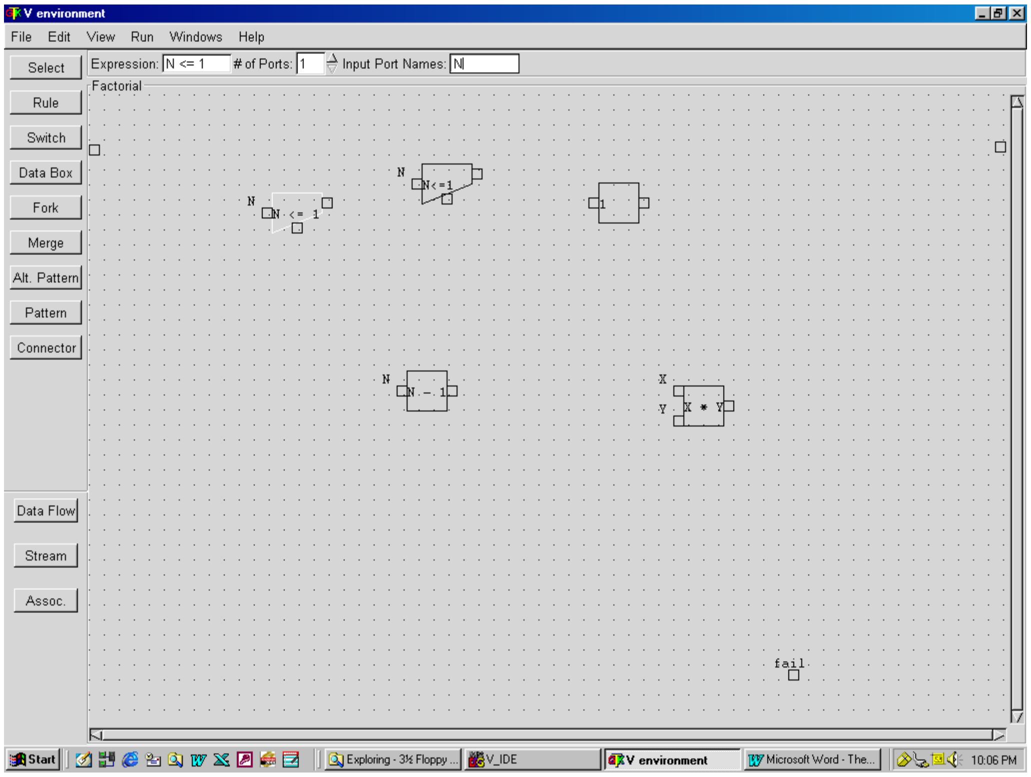


Figure 13. Creation of the Factorial Diagram

If an error is made, the relevant node is selected by clicking on the select button of the tool bar then on the node. The node is highlighted as shown in figure 14, and the edit bar contains the node's information. At this point the node can be deleted from the edit menu or the information changed and saved by clicking on the edit button at the end of the edit bar.

To complete the factorial example, the connections are made by selecting data flow on the tool button bar then selecting the source and destination ports in the workspace. The final diagram is shown completed in figure 15.



Expression: N <= 1 # of Ports: 1 Input Port Names: N

Figure 14. Selected node is highlighted and ready for editing

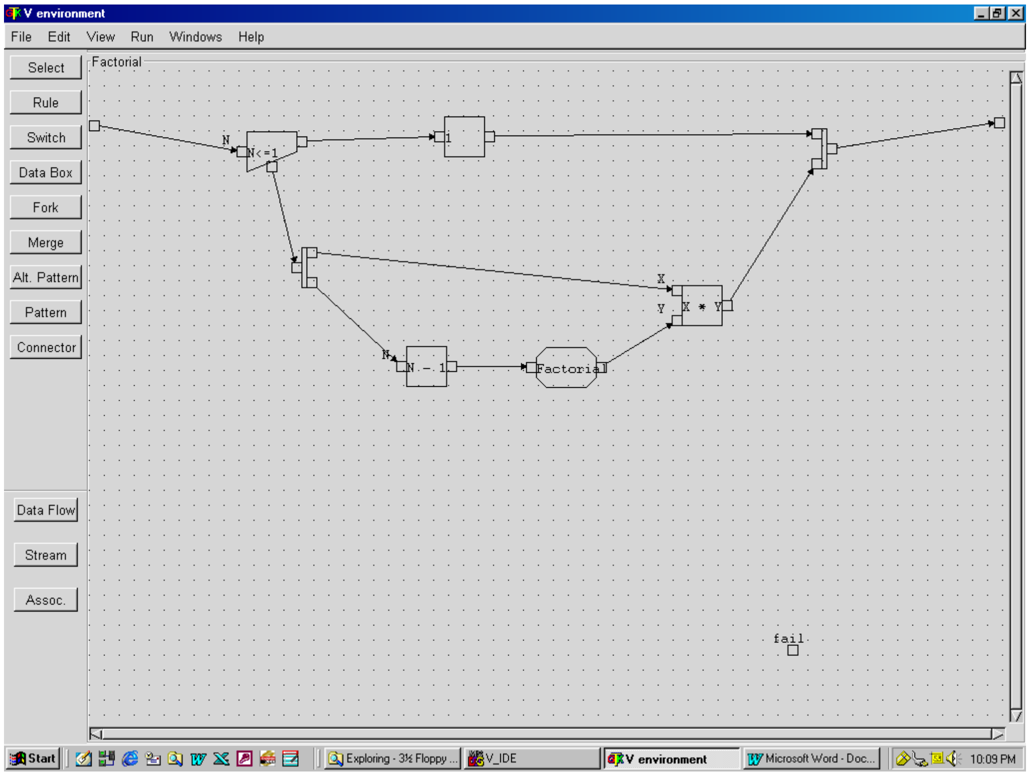


Figure 15. The completed factorial example diagram

V. CONCLUSIONS

A. SUMMARY

This project has implemented the basic functionality required for the graphical editor for the V language. It allows the user to see a graphic representation of the data flow between components. Using this program, the user is able to create a new diagram using the basic icons for the language, edit a diagram, create an output file from the diagram that is usable by a program generator to produce an Ada program, and save his work. We believe that this program will enable its users to evaluate the language and associated program generator faster and easier than through hand drawn examples.

B. RECOMMENDATIONS FOR FUTURE WORK

1. Program Generator Integration

One of the next steps in utilizing the V language, graphical editor, and program generator is integrating the program generator into an IDE with the editor. This will allow the user to run the program generator directly from the editor without saving the intermediate text file and switching to a separate command line program. The direct generation of Ada code from the IDE would be a time saver and would be required to support debugging and step though.

2. Additional Views

The graphical editor would benefit from the inclusion of multiple view of the diagram model. One use for this is an addition text view so the user could see and edit the text representation of the diagram directly alongside of the graphical view. A more interesting application would be views of parts of the model where a second workspace would show the diagram of a rule call in the primary diagram. In this manner, multiple workspaces could be linked together into a single program in the same way current text based programming IDEs display and link multiple files. Additional views would also be the quickest and easiest way to implement the history and undo/redo functionality suggested under future requirements

3. Extended Graphics

More work can be done with the GtkAda toolkit to support more accurate representations of the basic icons for the V language. The notation in this editor approximates the shapes for stream and association due to time constraints. Future work can accurately represent them as wide and medium arrows respectively. Other work includes finer granularity on size control for nodes, direct editing of node attributes in the workspace instead of the edit bar, and the inclusion of waypoints allowing multi-segmented connections to go around nodes.

4. Graph Based Automatic Layout

The use of graph theory to arrange nodes in the workspace is another promising area of development. The minimization of crossing data lines and the logical arrangement of the shapes on the screen can go far to reduce clutter and improve readability. One of the major problems with visual languages is that as the size of the drawing increases, the readability due to clutter goes down.

APPENDIX A. VISUAL DEVELOPMENT ENVIRONMENT CODE

A-1 DIAGRAM.ADS

```
--========================================================
-- FILE: diagram.ads
-- AUTHOR: Steven Carpenter
-- VERSION: 1.1
-- LAST MODIFIED: 23 September, 2002
-- DESCRIPTION: This file provides the node and connection types used
by the V graphical
--  editor system.
-- COMPILER: GNAT 3.13p
-- WARNINGS: None
--========================================================

package Diagram is

   -- Types of elements represented by V
   type Drawings is (None, Diagram_Call, Data_Switch, Data_Box,
                        Data_Fork, Data_Merge,Alt_Pattern, Rule,
                        Pattern_Box, Connector, Pattern_Group,
Data_Flow,
                        Open_Stream, Association, Port, Waypoint );

   -- Types of connections between diagrams
   subtype Connections is Drawings range Data_Flow..Association;

   -- Types of diagrams represented by V
   subtype Diagrams is Drawings range Diagram_Call..Pattern_Group;

end Diagram;
```

A-2 MODELS.ADS

```
--==========================================================
-- FILE: Models.ads
-- AUTHOR: Steven Carpenter
-- VERSION: 1.4
-- LAST MODIFIED: 23 September, 2002
-- DESCRIPTION: This package provides the procedures to create
--  and manipulate the internal representation of a V program.
--  It includes the set of datagrams representing the nodes in
--  the diagram as well as the set of connections between nodes.
--  It also implements the observable interface to notify its
--  observers of any changes to the model
-- COMPILER: GNAT 3.13p
-- WARNINGS: None
--==========================================================

with Observer_Pattern; use Observer_Pattern;
with Ada.Strings.Unbounded; use Ada.Strings.Unbounded;
with Model_Elements; use Model_Elements;
with Diagram;
with Glib; use Glib;

package Models is

   -- Container for Connectors
   type Arrow_List is array(1..50) of Arrow_Ptr;

   -- definition for observable interface
   type Model_Observers;
   type Model_Observers_Ptr is access all Model_Observers;




   -- Model class, contains diagram name, the root datagram
   -- containing all datagrams in the diagram, a switch denoting
   -- a connection tot he fail port, the list of connections,
   -- the most recently changed datagram, and the observable
   -- interface.
   type Model_Record is tagged
     record
        Name  : Unbounded_String;
        Root  : Datagram_Ptr;
        Fail_Active : Boolean;
        Links : Arrow_List;
        Changed : Datagram_Ptr;
        Observers : Model_Observers_Ptr;
     end record;

   type Model is access all Model_Record'Class;

   -- Model observer is the implementation of the observable
   -- interface for the model.
   type Model_Observers is new Observable_Record with
     record
        Observed_Model : Model;
     end record;


--==========================================================
-- PROCEDURE: Initialize
-- INPUT: A string for the model name, default is "Untitled"
--        An integer for the number of input ports, default is 2,
--        A string for the input port labels, default is empty
--        An integer for the number of output ports, default is 1
-- OUTPUT: A new instance of a model
-- PRECONDITIONS: None
-- DESCRIPTION: This procedure creates a new model
--=========================================================
   procedure Initialize ( New_Model : out Model;
                          New_Name  : String   := "Untitled";
                          Num_Iports : Integer := 2;
                          Inp_Labels : String  := "";
                          Num_Oports : Integer := 1);


--==========================================================
-- PROCEDURE: Add
-- INPUT: The current model to add a datagram to
--        The parent datagram of the new datagram
--        The type of the new datagram to add
--        A string for the expression of the new datagram
--        A string for the value type of the new datagram
--        An integer for the number of input ports
--        An integer for the number of output ports
--        A string for the input port labels
--        A position containing the x and y coordinates
-- OUTPUT: A reference to the datagram added to the model
-- PRECONDITIONS: A model must already exist
--                If adding a child, its parent must exist
-- DESCRIPTION: This procedure creates a new datagram and adds
--  it to the model.
--=========================================================
   procedure Add (Current_Model : access Model_Record;
                  Parent      : in out Datagram_Ptr;
                  Obj_Type    : in Diagram.Drawings;
                  Expression  : in String;
                  Expr_Type   : in String;
                  Number_Inp_Ports, Number_Out_Ports : in Integer;
                  Labels      : in String;
                  Xypos       : in Position;
                  New_Element : out Datagram_Ptr);



--==========================================================
-- PROCEDURE: Remove
-- INPUT: The current model to remove a datagram from
--        The datagram to remove from the model
-- OUTPUT: None
-- PRECONDITIONS: The model must exist
--                The item to be removed must exist
-- DESCRIPTION: This procedure removes a datagram from the model
--=========================================================
   procedure Remove(Current_Model : Model;
                    Item : Datagram_Ptr);



--==========================================================
-- PROCEDURE: Add_Link
-- INPUT: The current model to add a link to
--        The type of the connection to add
--        The datagram that is the source of the connection
--        The datagram that is the destination of the connection
--        The port number of the source port
--        The port number of the destination port
--        A string for the association of the connection
-- OUTPUT: None
-- PRECONDITIONS: The model must exist
--                The source and destination datagrams must exist
--                If the destination port is the fail port the
--                port number must be zero
-- DESCRIPTION: This procedure creates a link between two ports and
--  adds it to the model
--=========================================================
   procedure Add_Link  (Current_Model: access  Model_Record;
                        Obj_Type : in Diagram.Connections;
                        Src_Datagram, Dest_Datagram : in out
Datagram_Ptr;
                        Src_Port_Number, Dest_Port_Number : in Integer;
                        Link_Label : String );



--==========================================================
-- PROCEDURE: Remove_Link
-- INPUT: The current model to remove a connection from
--        A datagram connected to the connection
--        A variable denoting whether the datagram is a source
--        The port number of the connection to the datagram
-- OUTPUT: None
-- PRECONDITIONS: The model must exist
--                the link must exist
-- DESCRIPTION: This procedure removes a link between two
--  datagrams.
--=========================================================
   procedure Remove_Link( Current_Model : Model;
                          Sel_Datagram : Datagram_Ptr;
                          Is_Source : Boolean;
                          Port_Number : Integer);



--==========================================================
-- PROCEDURE: Modify
-- INPUT: The model to be modified
-- OUTPUT: None
-- PRECONDITIONS: The model must exist
--                the datagram to be modified must exist
-- DESCRIPTION: This procedure modifies one or more of
--  the attributes of a datagram in a model. The root
--  datagram containing the model information can also be
--  modified.
--=========================================================
   procedure Modify(Current_Model : Model;
                    The_Obj       : Datagram_Ptr;
                    Expression    : in String;
                    Expression_Type : in String;
                    Number_Iports : in Integer;
                    Input_Labels  : in String;
                    Number_Oports : in Integer);



--==========================================================
-- FUNCTION: Get_Name
-- INPUT: The model to be queried
-- OUTPUT: A string containing the name of the model
-- PRECONDITIONS: The model must exist
-- DESCRIPTION: This function gets the name of the diagram
--=========================================================
   function Get_Name(The_Model: in Model) return String;



--==========================================================
-- FUNCTION: Get_I_Ports
-- INPUT: The model queried
-- OUTPUT: An integer representing the number of input ports
-- PRECONDITIONS: The model must exist
-- DESCRIPTION: See Output
--=========================================================
   function Get_I_Ports(The_Model: in Model) return Integer;



--==========================================================
-- FUNCTION:  Get_O_Ports
-- INPUT: The model queried
-- OUTPUT: An integer representing the number of input ports
-- PRECONDITIONS: The model must exist
-- DESCRIPTION: See Output
--=========================================================
   function Get_O_Ports(The_Model: in Model) return Integer;



--==========================================================
-- FUNCTION: Get_Changed
-- INPUT: The model queried
-- OUTPUT: The last datagram changed
-- PRECONDITIONS: A datagram has been changed
-- DESCRIPTION: This function gets the last changed datagram
--=========================================================
   function Get_Changed(The_Model : in Model) return Datagram_Ptr;



--==========================================================
-- FUNCTION: Objects_Done
-- INPUT: The model under question
-- OUTPUT: True if the current datagram in the model is the last
--  one, False otherwise.
-- PRECONDITIONS: The model must exist
-- DESCRIPTION: This function checks to see if every element
--  that is a direct child of the root diagram has been visited
--=========================================================
   function Objects_Done(The_Model: in Model) return Boolean;

--==========================================================
-- PROCEDURE: Next_Object
-- INPUT: The model under question
-- OUTPUT: The next unvisited datagram in the model
-- PRECONDITIONS: The model must exist
-- DESCRIPTION: See output
--=========================================================
   procedure Next_Object (The_Model: in Model; Obj : out Datagram_Ptr
);


--==========================================================
-- PROCEDURE: Start_Object
-- INPUT: The model querried
-- OUTPUT: The root datagram
-- PRECONDITIONS: The model exists
-- DESCRIPTION: This procedure gets the root datagram of the diagram
--=========================================================
   procedure Start_Object (The_Model: in Model; Obj :out Datagram_Ptr
);


--==========================================================
-- FUNCTION: Arrows_Done
-- INPUT: The model queried
-- OUTPUT: True if all the arrows have been visisted
-- PRECONDITIONS: The model must exist
-- DESCRIPTION: This function checks to see if all arrows have been
visited
--========================================================
   function Arrows_Done(The_Model: in Model) return Boolean;



--=========================================================
-- PROCEDURE: Next_Arrow
-- INPUT: The model queried
-- OUTPUT: The next unvisited connection in the model
-- PRECONDITIONS: The model must exist
-- DESCRIPTION: See output
--========================================================
   procedure Next_Arrow (The_Model: in Model; Arrw : out Arrow_Ptr );



--=========================================================
-- PROCEDURE: Start_Arrow
-- INPUT: The model queried
-- OUTPUT: The first connection in the model
-- PRECONDITIONS: The model must exist
-- DESCRIPTION: See output
--========================================================
   procedure Start_Arrow (The_Model: in Model; Arrw :out Arrow_Ptr );



--=========================================================
-- PROCEDURE: Register
-- INPUT: The model registering an observer
--        An observer requesting notification of changes to the model
-- OUTPUT: None
-- PRECONDITIONS: The model must exist
-- DESCRIPTION: This prodedure implements the observable interface.
--  It adds the observer to the list of observers the model notifies
--  of every change.
--========================================================
   procedure Register( Subject: access Model_Record; Obs: in Observer);
```

```
--==========================================================
-- PROCEDURE: Unregister
-- INPUT: The model unregistering an observer
--        The observer requesting not to be notified of any more
changes
-- OUTPUT: None
-- PRECONDITIONS: The model must exist
-- DESCRIPTION: This procedure removes an observer from the list
--  of observers kept by the model
--=========================================================
   procedure UnRegister( Subject: access Model_Record; Obs: in
Observer);



end Models;
```

A-3 MODEL_ELEMENTS.ADS

```
--==========================================================
-- FILE: model_elements.ads
-- AUTHOR: Steven Carpenter
-- VERSION: 1.4
-- LAST MODIFIED: 24 September, 2002
-- DESCRIPTION: This package contains the definitions and
--  methods associated with datagrams and connections in the
--  V language.
-- COMPILER: GNAT 3.13p
-- WARNINGS: None
--==========================================================
with Observer_Pattern; use Observer_Pattern;
with Ada.Strings.Unbounded; use Ada.Strings.Unbounded;
with Diagram;

package Model_Elements  is

   -- The maximum size for an input port label
   -- Currently set to 5 to minimize space on the screen
   -- taken up by the label
   Max_Port_Label_Size : constant Natural := 5;

   -- Early Definition of connection type
   type Arrow_Record;
   type Arrow_Ptr is access all Arrow_Record;

   -- Type to hold list of connections in the datagram
   -- each index corresponds to a port number
   type Ports is array(Integer range <>) of Arrow_Ptr;
   type Port_Access is access Ports;

   -- Type to hold list of input port labels
   -- should be one for each input port
   type Label_List is array(Integer range <>) of
     String(1..Max_Port_Label_Size);
   type Port_Labels is access Label_List;

   -- coordinate type for recording the datagrams position
   type Position is array(1..2) of Integer;

   type Datagram_Record;
   type Datagram_Ptr is access all Datagram_Record;
   type Datagram_List is array(1..10) of Datagram_Ptr;

   -- Type that represents a node in a V diagram.
   -- Each node contains an access to its parrent as
   -- well as a list of its children.
   -- It also conatains a list of observers for future
   -- implementation of the observer pattern at this level
   type Datagram_Record is tagged
      record
        Data_Type    : Diagram.Drawings;
        Expr         : Unbounded_String;
        Expr_Type    : Unbounded_String;
        Input_Ports  : Port_Access;
```

```
        Output_Ports : Port_Access;
        Labels       : Port_Labels;
        Observers    : Notification_List;
        Parent       : Datagram_Ptr;
        Children     : Datagram_List;
        Pos          : Position;
      end record;

   -- Type that represents a connection between two
   -- datagrams. Contains an access to the two datagrams.
   -- Association connections also contain a string to
   -- hold the association. Other types hold the empty string
   type Arrow_Record  is tagged
     record
        Connector_Type  : Diagram.Connections;
        Src_Number      : Integer;
        Dest_Number     : Integer;
        Source          : Datagram_Ptr;
        Destination     : Datagram_Ptr;
        Assoc           : Unbounded_String;
     end record;



--==========================================================
-- PROCEDURE: Create_Datagram
-- INPUT: The type of the datagram
--              A string containing the expression in the datagram
--              A string containing the expression type in the datagram
--              The number of input ports as an integer
--              A string containing the input port labels
--          The number of output ports as an integer
--          The parent datagram for the new datagram
--          The position of the new datagram
-- OUTPUT: A new datagram with the fields based on inputs
-- PRECONDITIONS: The parent datagram must exist
-- DESCRIPTION: This procedure creates a new datagram with the
specified fields
--========================================================
   procedure Create_Datagram(Reference     : out Datagram_Ptr;
                             Obj_Type      : in Diagram.Drawings;
                             Expression    : in String;
                             Expression_Type : in String;
                             Number_Iports : Integer;
                             Input_Labels  : String;
                             Number_Oports : Integer;
                             Parent        : Datagram_Ptr;
                             Location      : Position );


--=========================================================
-- PROCEDURE: Create_Arrow
-- INPUT: The type of the connection to create
--         The port number in the source datagram
--         The port number in the destination datagram
--         The source datagram
--         The destination datagram
--         A string containing the association for association
connections
-- OUTPUT: A new connection with the specified fields.
-- PRECONDITIONS: The parent datagram must exist
-- DESCRIPTION: See output.
--========================================================
   procedure Create_Arrow( The_Arrow : out Arrow_Ptr;
                           Arrow_Type : in Diagram.Drawings;
                           Number_Iport : in Integer;
                           Number_Oport : in Integer;
                           Source : Datagram_Ptr;
                           Dest : Datagram_Ptr;
                           Link_Label  : in String);


--=========================================================
-- PROCEDURE: Modify
-- INPUT: The datagram to modify
--         A string containing the new expression
--         A string containing the new expression type
--         The new number of input ports for the datagram
--         A string containing the new input port labels
--         The new number of output ports for the datagram
-- OUTPUT: None
-- PRECONDITIONS: The datagram to modify must exist
-- DESCRIPTION: This procedure modifies an existing datagram to reflect
the
--  changes specified in the input fields.
--=========================================================
   procedure Modify(The_Obj         : Datagram_Ptr;
                    Expression      : in String;
                    Expression_Type : in String;
                    Number_Iports   : Integer;
                    Input_Labels    : String;
                    Number_Oports   : Integer);



--==========================================================
-- PROCEDURE: Delete
-- INPUT: The datagram to delete
-- OUTPUT: None
-- PRECONDITIONS: An existing datagram
-- DESCRIPTION: This procedure deletes a datagram
--=========================================================
   procedure Delete(The_Obj: Datagram_Ptr);



--==========================================================
-- PROCEDURE: Delete
-- INPUT: The connection to delete
-- OUTPUT: None
-- PRECONDITIONS: An existing connection
-- DESCRIPTION: This procedure removes the references to the connection
--  from the source and destination datagrams and deletes the
connection
--=======================================================
   procedure Delete(Arrow : Arrow_Ptr);


--========================================================
-- PROCEDURE: Add_Child
-- INPUT: A datagram to add a child datagram to
--         The datagram to add as a child
-- OUTPUT: None
-- PRECONDITIONS: The parent and child datagrams must exist
-- DESCRIPTION: This procedure adds a datagram as a child of another
--  datagram to show containment of the child by the parent
--=======================================================
   procedure Add_Child(The_Obj : in out Datagram_Ptr;
                       The_Child : in Datagram_Ptr);


--========================================================
-- FUNCTION: Children_Exist
-- INPUT: A datagram to check for children
-- OUTPUT: True if the datagram contains one or more children
--   False otherwise.
-- PRECONDITIONS: The datagram must exist
-- DESCRIPTION: This function checks to see if there are any children
in
--  the datagram's children list
--=======================================================
   function Children_Exist(The_Obj : in Datagram_Ptr) return Boolean;


--========================================================
-- PROCEDURE: Set_Pos
-- INPUT: The datagram to change the position of
--         The new X coordinate as an Integer
--         The new Y coordinate as an Integer
-- OUTPUT: None
-- PRECONDITIONS: The datagram must exist
-- DESCRIPTION: This procedure changes the position field of the
datagram
--========================================================
   procedure Set_Pos(The_Obj : Datagram_Ptr; X : Integer; Y : Integer);


--=========================================================
-- FUNCTION: Get_Expr
-- INPUT: The datagram to query
-- OUTPUT: A string containing the expression of the datagram
--             the default value is the empty string
-- PRECONDITIONS: The datagram must exist
-- DESCRIPTION: This function reads the expression field of the
datagram
--========================================================
   function Get_Expr(The_obj: in Datagram_Ptr ) return String;


--=========================================================
-- FUNCTION: Get_Expr_Type
-- INPUT: The datagram to query
-- OUTPUT: A string containing the expression type of the datagram
--             the default value is the empty string
-- PRECONDITIONS: The datagram must exist
-- DESCRIPTION: This function reads the expression type field of the
datagram
--========================================================
   function Get_Expr_Type(The_obj: in Datagram_Ptr ) return String;


--=========================================================
-- FUNCTION: Get_Type
-- INPUT: The datagram to query
-- OUTPUT: The type of the datagram
-- PRECONDITIONS: The datagram must exist
-- DESCRIPTION: See output
--=========================================================
   function Get_Type(The_Obj: in Datagram_Ptr ) return
Diagram.Drawings;


--==========================================================
-- FUNCTION: Get_Pos
-- INPUT: The datagram to query
-- OUTPUT: A position containing the X and Y coordinates of the
datagram
-- PRECONDITIONS: The datagram must exist
-- DESCRIPTION: See output
--=========================================================
   function Get_Pos( The_Obj : in Datagram_Ptr) return Position;


--==========================================================
-- FUNCTION: Get_Label
-- INPUT: The datagram to query
--          The number of the port in the datagram associated with
--          the desired label
-- OUTPUT: A string containing the label of the requested datagram
input port
--                 default value is the empty string
-- PRECONDITIONS: The datagram must exist
-- DESCRIPTION: See output
--=========================================================
   function Get_Label(The_Obj : in Datagram_Ptr;
                      Port_Number : Integer) return String;


--==========================================================
-- FUNCTION: Get_Num_In
-- INPUT: The datagram to query
-- OUTPUT: The number of input ports of the datagram
-- PRECONDITIONS: The datagram must exist
-- DESCRIPTION: See output
--=========================================================
   function Get_Num_In( The_Obj : in Datagram_Ptr) return Integer;



--==========================================================
-- FUNCTION: Get_Num_Out
-- INPUT: The datagram to query
-- OUTPUT: A number of output ports of the datagram
-- PRECONDITIONS: The datagram must exist
-- DESCRIPTION: See output
--=========================================================
   function Get_Num_Out( The_Obj : in Datagram_Ptr) return Integer;



--==========================================================
-- FUNCTION: Get_Assoc
-- INPUT: A variable of type Edit_Strip
-- OUTPUT: True if the edit strip is currently representing a
--  function, False otherwise.
-- PRECONDITIONS: An existing edit strip
-- DESCRIPTION: This function checks to see if edit_function was
--  the last edit command(non-get).
--=========================================================
   function Get_Assoc( The_Arrow : in Arrow_Ptr) return String;

end Model_Elements;
```

A-4 PRINT_OUT.ADS

```
--==========================================================
-- FILE: print_out.ads
-- AUTHOR: Steven Carpenter
-- VERSION: 1.1
-- LAST MODIFIED: 24 September, 2002
-- DESCRIPTION: This package contains the method to print the
--  diagram in a format readable by the program generator. It
--  contains types to keep track of port assignments in links and
--  datagrams during traversal of the model
-- COMPILER: GNAT 3.13p
-- WARNINGS: None
--=========================================================
with Models;
with Model_Elements;
with Diagram;

package Print_Out is

   -- Types to keep track of connections and their
   -- associated port assignments during the print
   type Print_Arrow_Record  is record
      Arrow_Type : Diagram.Connections;
      Src_Name, Dest_Name : String(1..4);
      Model_Arrow_Ptr : Model_Elements.Arrow_Ptr;
   end record;
   type Print_Arrow is access all Print_Arrow_Record;

   type Arrow_List is array(1..50) of Print_Arrow;


--=========================================================
-- PROCEDURE: Model_Print
-- INPUT: A model to print
-- OUTPUT: None
-- PRECONDITIONS: The model must exist
-- DESCRIPTION: This procedure iterates through every datagram in the
--  model and prints it to the file <<name>>.out. Port labels for
printing are
--  assigned as Px where x is an integer. Connections are printed after
--  datagrams so that port assignments can be matched to connections
--========================================================
   procedure Model_Print(Model_In : in Models.Model);

end Print_Out;
```

A-5 WORKSPACE.ADS

```
--=========================================================
-- FILE: work_space.ads
-- AUTHOR: Steven Carpenter based on extending the gtkada
--  interactive_canvas type. Port_Item and Display_Item are
--  based on the Display_Item type by E.Briot,J.Brobecker,
--  and A.Charlet in the testgtk create_canvas package
--  distributed with GtkAda. Methods Draw, Zoom_In, Zoom_Out,
--  and Remove_Link are also modified from the same package.
-- VERSION: 1.5
-- LAST MODIFIED: 24 September, 2002
-- DESCRIPTION: This package contains the types to create and
--  manipulate the workspace. It allows the manipulation of
--  items within the workspace representing datagrams and
--  connections in the model.
-- COMPILER: GNAT 3.13p
-- WARNINGS: None
--=========================================================
with Gtk.Frame;           use Gtk.Frame;
with Gtk.Handlers;        use Gtk.Handlers;
with Gtk.Scrolled_Window; use Gtk.Scrolled_Window;
with Gtk.Widget;          use Gtk.Widget;
with Gtkada.Canvas;       use Gtkada.Canvas;
with Ada.Strings.Unbounded; use Ada.Strings.Unbounded;
with Gdk.GC;
with Gdk.Font;
with Model_Elements;
with Models;
with Glib;
with Diagram;
with Observer_Pattern;
with Gdk.Types;
with Gdk.Pixmap;
with Gdk.Color;
with Gdk.Event;
```

```
package Work_Space is

   type Display_Item_Record;
   type Display_Item is access all Display_Item_Record'Class;
   type Port_Type is (Source, Destination);

   -- Type to represent ports on the screen. Contains a
   -- reference to the parent item, the width and height it
   -- takes up on the screen, its index number in its parent,
   -- whether it is a source or destination port, available to
   -- determine if it has a link or not, its own color indicating
   -- whether is is selected or not, and switch solid to indicate
   -- its fill type,
   type Port_Item_Record is new Canvas_Item_Record with record
      Parent          : Display_Item;
      W, H            : Glib.Gint;
      Port_Number     : Integer;
      Connection_Type : Port_Type;
      Available       : Boolean;
      Color           : Gdk.Color.Gdk_Color;
      Solid           : Boolean;
   end record;
   type Port_Item is access all Port_Item_Record'Class;

   -- Overridden procedure to allow the item to draw itself on
   -- the canvas.
   procedure Draw (Item    : access Port_Item_Record;
                   Canvas  : access Interactive_Canvas_Record'Class;
                   Dest    : Gdk.Pixmap.Gdk_Pixmap;
                   Xdest, Ydest : Glib.Gint);

   type Port_Array is array(1..10) of Port_Item;

   -- Type to represent datagrams on the screen. Contains a reference
   -- to the datagram it represents, arrays of input and output
   -- port_items to allow connections, Color to show selection
   -- status, size to allow resizing to accomidate children, the
   -- width and height it takes up on the screen, and the points
   -- describing its shape.
   type Display_Item_Record is new Canvas_Item_Record with record
      Source         : Model_Elements.Datagram_Ptr;
      Iports, Oports : Port_Array;
      Color          : Gdk.Color.Gdk_Color;
      Size           : Glib.Gint;
      W, H           : Glib.Gint;
      Points         : Gdk.Types.Gdk_Points_Array(1..8);
   end record;

   -- Overridden procedure to allow the item to draw itself on
   -- the canvas.
   procedure Draw (Item : access Display_Item_Record;
                   Canvas : access Interactive_Canvas_Record'Class;
                   Dest : Gdk.Pixmap.Gdk_Pixmap;
                   Xdest, Ydest : Glib.Gint);

   -- Type to hold the data elements in a clipboard. Contains all
   -- the data needed to create an identical datagram.
   type Copy_Element is record
      Valid : Boolean;
      Copy_Type : Diagram.Diagrams;
      Expr_Type : Unbounded_String;
      Expr : Unbounded_String;
      Names : Unbounded_String;
      In_Ports, Out_Ports : Integer;
   end record;

   -- Early definition of workspace
   type Work_Space_Record;
   type Work_Space_Access is access all Work_Space_Record'Class;

   --Type to implement the observer interface to allow the workspace
   -- to receive notification of changes to the model.
   type Model_Observer_Record is new Observer_Pattern.Observer_Record
with
```

```
      record
         Canvas : Work_Space_Access;
         Connected_Model  : Models.Model;
      end record;
   type Model_Observer is access all Model_Observer_Record'Class;

   -- Type to represent the workspace. Allows placement and motion of
   -- items on the screen. Contains attributes to hold both a selected
   -- screen item as well as its associated datagram, also holds the
   -- selected ports in the diagram for creating links, contains the
   -- observer link to register or unregister with the model, contains
   -- the clipboard, a port to allow deleting a link, and the root
   -- input, output and fail ports for the diagram/model
   type Work_Space_Record is new Interactive_Canvas_Record with record
      Selected_Element : Model_Elements.Datagram_Ptr;
      Selected_Item : Display_Item;
      Source_Port, Dest_Port : Port_Item;
      Observing : Model_Observer;
      Clipboard : Copy_Element;
      Changed_Port : Port_Item;
      Diagram_Source_Ports, Diagram_Dest_Ports : Port_Array;
      Fail_Port : Port_Item;
   end record;


   type Item_Color is (White, Black);

--==========================================================
-- PROCEDURE: Change_Color
-- INPUT: A canvas item to change color
--         The new color to change to, default is black
-- OUTPUT: None
-- PRECONDITIONS: The canvas item must exist
-- DESCRIPTION: This procedure changes the color of a canvas item or
--  any item descended from it that contains a color
--=========================================================
   procedure Change_Color( Item : access Canvas_Item_Record'Class;
                           New_Color : Item_Color := Black );


--==========================================================
-- PROCEDURE: Gtk_New
-- INPUT: None
-- OUTPUT: A new workspace
-- PRECONDITIONS: None
-- DESCRIPTION: This procedure creates a new workspace
--=========================================================
   procedure Gtk_New (Workspace : out Work_Space_Access);


--==========================================================
-- PROCEDURE: Initialize
-- INPUT: A workspace to initialize
-- OUTPUT: None
-- PRECONDITIONS: The workspace must have been created with Gtk_New
-- DESCRIPTION: This procedure Initializes the elements of the
workspace
--  so that it is ready to display items on the screen. It attaches the
--  "select item" and "zoomed" callbacks to the workspace.
--=========================================================
   procedure Initialize ( Workspace : access Work_Space_Record'Class);


--==========================================================
-- PROCEDURE: Setup
-- INPUT: The workspace to set up
--         The model that this workspace will act as a view for.
-- OUTPUT: None
-- PRECONDITIONS: The model must exist
--                             The workspace must have been initialized
-- DESCRIPTION: This procedure registers the workspace as an observer
--  on the model. It also creates the base view of the diagram the
model
--  will represent by putting the input, output and fail ports on the
--  screen with the diagram name and showing them to the user.
--=========================================================
   procedure Setup( Workspace :  Work_Space_Access;
                    Workspace_Model : Models.model);


--==========================================================
-- PROCEDURE: Notify
-- INPUT: The workspace to notify
--         The type of notification: added, changed, or deleted
-- OUTPUT: None
-- PRECONDITIONS: The workspace must exist and have registered with
--  a model. The model must have changed in some way.
-- DESCRIPTION: This procedure handles notification events from the
model.
--  Dependent on the notification type, this causes the workspace to
refresh
--  its view of the model by getting the changes and displaying them
properly
--=========================================================
   procedure Notify( Workspace   : access Model_Observer_Record;
                     Ntype : in Observer_Pattern.Notify_Type);


--==========================================================
-- PROCEDURE: Zoom_In
-- INPUT: A descendant of type canvas_item
-- OUTPUT: None
-- PRECONDITIONS: A zoomable canvas exists
-- DESCRIPTION: This procedure changes the zoom level of the canvas and
--  redraws all of the items on the screen in the new scale.
-- NOT CURRENTLY FUNCTIONAL
--=========================================================
   procedure Zoom_In(Canvas : access Work_Space_Record'Class);

--==========================================================
-- PROCEDURE: Zoom_Out
-- INPUT: A descendant of type canvas_item
-- OUTPUT: None
-- PRECONDITIONS: A zoomable canvas exists
-- DESCRIPTION: This procedure changes the zoom level of the canvas and
--  redraws all of the items on the screen in the new scale.
-- NOT CURRENTLY FUNCTIONAL
--==========================================================
   procedure Zoom_Out(Canvas : access Work_Space_Record'Class);



--===========================================================
-- PROCEDURE: Set_Clipboard
-- INPUT: The workspace containing the clipboard
--        The datagram type to add to the clipboard
--        A string containing the expression type to add to the
clipboard
--        A string containing the expression to add to the clipboard
--        A string containing the input port labels to add to the
clipboard
--        The number of input ports as an integer to add to the
clipboard
--        The number of output ports as an integer to add to the
clipboard
-- OUTPUT: None
-- PRECONDITIONS: The workspace must exist
-- DESCRIPTION: This procedure saves to the clipboard all the
information
--  necessary to create a new datagram from a copied datagram
--==========================================================
   procedure Set_Clipboard( Workspace : access Work_Space_Record'Class;
                            Item_Type: Diagram.Diagrams;
                           Item_Expr_Type : String;
                           Item_Expr : String;
                           Item_Names : String;
                           Item_In_Ports : Integer;
                           Item_Out_Ports : Integer);
```

```
end Work_Space;
```

A-6 OBSERVER_PATTERN.ADS

```
--==========================================================
-- FILE: observer_pattern.ads
-- AUTHOR: Steven Carpenter
-- VERSION: 1.2
-- LAST MODIFIED: 23 September, 2002
-- DESCRIPTION: THis package provides the observer and
--  observable interfaces for the observer pattern. The
--  class being observed must extend the observable_record
--  to include a reference to itself. The observers must
--  overide the notify(observer) method to point to thier
--  own notify method
-- COMPILER: GNAT 3.13p
-- WARNINGS: None
--==========================================================
package Observer_Pattern is

   type Notify_Type is ( Added, Changed, Deleted );

   -- Type of observer, to be extended by the observer
   type Observer_Record is abstract tagged null record;
   type Observer is access all Observer_Record'Class;

   type Notification_List is array(1..3) of Observer;

   -- Type for the observed class, must be extended to include
   -- the reference to the observed class.
   type Observable_Record is abstract tagged
      record
         Observers : Notification_List;
      end record;
   type Observable is access all Observable_Record'Class;

--==========================================================
-- PROCEDURE: Notify
-- INPUT: An observer,
--        The type of notification
-- OUTPUT: None
-- PRECONDITIONS: None
-- DESCRIPTION: This procedure calls the appropriate methods
--  in the observer to let it know that there is a change
--=========================================================
   procedure  Notify(  Obs  :  access  Observer_Record'Class;  Nt  :  in
Notify_Type);


--==========================================================
-- PROCEDURE: Notify
-- INPUT: An observable object
--        The type of notification
-- OUTPUT: None
-- PRECONDITIONS: None
-- DESCRIPTION: This procedure allows the model to notify
--  all of its registered views that it has changed. It
--  calls notify on each observer in its notification list
--=========================================================
   procedure Notify(Obs : in Observable; Nt : Notify_Type);


--==========================================================
-- PROCEDURE: Add
-- INPUT: An observable
--        An observer
-- OUTPUT: None
-- PRECONDITIONS: None
-- DESCRIPTION: This procedure adds an observer to the notification
--  list of the target observable. It is called from the concrete
--  observable classes register method
--=========================================================
   procedure Add(Target : in out Observable; Obs : in Observer);
```

```
--==========================================================
-- PROCEDURE: Remove
-- INPUT: An observable
--        An observer
-- OUTPUT: None
-- PRECONDITIONS: None
--  DESCRIPTION:  This  procedure  removes  an  observer  from  the
notification
--  list of the target observable. It is called from the concrete
--  observable class's unregister method
--=========================================================
   procedure Remove( Target : in out Observable; Obs : in Observer);


end Observer_Pattern;
```

A-7 WINDOW_MAIN.ADS

```
--==========================================================
-- FILE: window_main.ads
-- AUTHOR: Steven Carpenter using Glade 0.5.9
-- VERSION: 1.4
-- LAST MODIFIED: 24 September, 2002
-- DESCRIPTION: This package creates the main window including
--  the menu bar and tool bar. It also holds the edit strip,
--  model, and workspace.
-- COMPILER: GNAT 3.13p
-- WARNINGS: None
--==========================================================
with Gtk.Window; use Gtk.Window;
with Gtk.Handlers;
with Gtk.Table; use Gtk.Table;
with Gtk.Menu_Bar; use Gtk.Menu_Bar;
with Gtk.Object; use Gtk.Object;
with Gtk.Menu_Item; use Gtk.Menu_Item;
with Gtk.Menu; use Gtk.Menu;
with Gtk.Scrolled_Window; use Gtk.Scrolled_Window;
with Gtk.Viewport; use Gtk.Viewport;
with Gtk.Frame; use Gtk.Frame;
with Gtk.Packer; use Gtk.Packer;
with Gtk.Separator; use Gtk.Separator;
with Gtk.Alignment; use Gtk.Alignment;
with Gtk.Vbutton_Box; use Gtk.Vbutton_Box;
with Gtk.Hbutton_Box; use Gtk.Hbutton_Box;
with Gtk.Button; use Gtk.Button;
with Gtk.Pixmap; use Gtk.Pixmap;
with Gtk.Widget;
with Gtk.Box;
with Diagram;
with Gtkada.Canvas;
with Work_Space;
with Models;
with Edit_Strip;

with Ada.Strings.Unbounded; use Ada.Strings.Unbounded;
package Window_main is

   -- Callback connect handlers for menu items
   package Menu_Item_Callback is new
     Gtk.Handlers.Callback (Gtk_Menu_Item_Record);

   -- Callback connect handlers for button items
   package Button_Callback is new
     Gtk.Handlers.Callback (Gtk_Button_Record);

   -- The main window record. Contains all attributes to hold
   -- all menu items, buttons, frames and picture icons for the
   -- main screen. Separated into three sections, the first
   -- contains all the menu items, the second contains the tool
   -- button items, and the third sets up the workspace and
   -- edit strip area.
   type Window_main_Record is new Gtk_Window_Record with record
      CurrentFile : Unbounded_String;
      Table1 : Gtk_Table;
      ----------------------
      -- Menu bar section --
```

```
----------------------
Menubar1 : Gtk_Menu_Bar;
File1 : Gtk_Menu_Item;
File1_Menu : Gtk_Menu;
New2 : Gtk_Menu_Item;
Open1 : Gtk_Menu_Item;
Save1 : Gtk_Menu_Item;
Saveas1 : Gtk_Menu_Item;
Exit1 : Gtk_Menu_Item;
Edit1 : Gtk_Menu_Item;
Edit1_Menu : Gtk_Menu;
Delete :  Gtk_Menu_Item;
Cut1 : Gtk_Menu_Item;
Copy1 : Gtk_Menu_Item;
Paste1 : Gtk_Menu_Item;
View1 : Gtk_Menu_Item;
View1_Menu : Gtk_Menu;
Zoom_In : Gtk_Menu_Item;
Zoom_Out : Gtk_Menu_Item;
Run : Gtk_Menu_Item;
Run_Menu : Gtk_Menu;
Run_Program : Gtk_Menu_Item;
Windows1 : Gtk_Menu_Item;
Windows1_Menu : Gtk_Menu;
New_Window1 : Gtk_Menu_Item;
Help1 : Gtk_Menu_Item;
Help1_Menu : Gtk_Menu;
Info1 : Gtk_Menu_Item;
Separator1 : Gtk_Menu_Item;
About1 : Gtk_Menu_Item;

------------------------
-- Button Bar section --
------------------------
-- The currently selected button
Selected_Button : Diagram.Drawings;
```

```
Vbuttonbox1 : Gtk_Vbutton_Box;
Select_Button : Gtk_Button;
Pixmap1 : Gtk_Pixmap;
Function_Button : Gtk_Button;
Pixmap2 : Gtk_Pixmap;
Switch_Button : Gtk_Button;
Pixmap3 : Gtk_Pixmap;
Data_Box_Button : Gtk_Button;
Pixmap4 : Gtk_Pixmap;
Fork_Button : Gtk_Button;
Pixmap5 : Gtk_Pixmap;
Merge_Button : Gtk_Button;
Pixmap6 : Gtk_Pixmap;
Alternate_Button : Gtk_Button;
Rule_Button : Gtk_Button;
Pixmap7 : Gtk_Pixmap;
Pattern_Box_Button : Gtk_Button;
Pixmap8 : Gtk_Pixmap;
Connector_Button : Gtk_Button;
Pattern_Group_Button : Gtk_Button;
Pixmap9 : Gtk_Pixmap;
Hseparator1 : Gtk_Hseparator;
Vbuttonbox2 : Gtk_Vbutton_Box;
Data_Flow_Button : Gtk_Button;
Pixmap10 : Gtk_Pixmap;
Stream_Button : Gtk_Button;
Pixmap11 : Gtk_Pixmap;
Association_Button : Gtk_Button;
Pixmap12 : Gtk_Pixmap;

--------------------
-- Canvas section --
--------------------
Work_Box    : Gtk.Box.Gtk_Vbox;
Edit_Frame  : Gtk_Frame;
Edit_Box    : Gtk.Box.Gtk_Hbox;
Edit_Bar    : Edit_Strip.Edit_Strip;
```

```
      Button_Frame : Gtk_Frame;
      Edit_Button : Gtk_Button;
      CanvasFrame : Gtk_Frame;
      Scrolledwindow1 : Gtk_Scrolled_Window;
      Workspace   : Work_Space.Work_Space_Access;

      -- The current Model
      Model1 : Models.Model;

   end record;
   type Window_main_Access is access all Window_main_Record'Class;


--==========================================================
-- PROCEDURE: Gtk_New
-- INPUT: None
-- OUTPUT: The main window for the graphical editor
-- PRECONDITIONS: None
-- DESCRIPTION: This procedure creates a new window with a menu bar,
--  Tool bar on the left side, edit strip under the menu bar, and an
area for
--  a workspace.
--=========================================================
   procedure Gtk_New (mainWindow : out Window_main_Access);


--==========================================================
-- PROCEDURE: Initialize
-- INPUT: The main window to initialize
-- OUTPUT: None
-- PRECONDITIONS: The window must have been created with Gtk_New
-- DESCRIPTION: This procedure creates and attaches all the widgets
--  necessary to implement the main window. It connects to all the
callbacks
--  in the callbacks package.
--=========================================================
   procedure Initialize (mainWindow : access Window_main_Record'Class);
   mainWindow : Window_main_Access;

end Window_main;
```

A-8 WINDOW_MAIN-CALLBACKS.ADS

```
--========================================================
-- FILE: window_main-callbacks.ads
-- AUTHOR: Steven Carpenter using glade 0.5.9
-- VERSION: 1.2
-- LAST MODIFIED: 24 September, 2002
-- DESCRIPTION: This package contains all the callbacks
--  for the buttons and menu-items declared in window_main
--  It also contains procedures used by these callbacks to
--  effect the model, edit strip and workspace.
-- COMPILER: GNAT 3.13p
-- WARNINGS: None
--========================================================

with Gtk.Arguments;
with Gtk.Widget; use Gtk.Widget;

package Window_main.Callbacks is

--========================================================
-- FUNCTION: Delete_Event
-- INPUT: The main window of the program
-- OUTPUT: False
-- PRECONDITIONS:
-- DESCRIPTION: This function sets the mainwindow to respond to delete
--  events generated by the upper right close button in the window
--=======================================================
   function Delete_Event (Object : access Window_main_Record'Class)
                       return Boolean;



--========================================================
-- PROCEDURE: Exit_Main
-- INPUT: The main window of the program
-- OUTPUT: None
-- PRECONDITIONS: The main window must exist
-- DESCRIPTION: This procedure closes the main window and shuts down
--  the program
--========================================================
   procedure Exit_Main
     (Object : access Window_main_Record'Class);


--=========================================================
-- PROCEDURE: On_New_Activate
-- INPUT: The menu item selected by the user
-- OUTPUT: None
-- PRECONDITIONS: The main window must exist
-- DESCRIPTION: This procedure creates a new model and workspace
--  with information from the edit strip. Any previous model and
workspace
--  are removed.
--========================================================
   procedure On_New_Activate
     (Object : access Gtk_Menu_Item_Record'Class);


--=========================================================
-- PROCEDURE: On_Open_Activate
-- INPUT: The menu item selected by the user
-- OUTPUT: None
-- PRECONDITIONS: The main window must exist
-- DESCRIPTION: This procedure creates a file selection dialog to get
the
--  required file name from the user. The selected file is then opened
and
--  should be read into the program
--========================================================
   procedure On_Open_Activate
     (Object : access Gtk_Menu_Item_Record'Class);
```

```
--==========================================================
-- PROCEDURE: On_Save_Activate
-- INPUT: The menu item selected by the user
-- OUTPUT: None
-- PRECONDITIONS: The main window must exist
-- DESCRIPTION: This procedure saves the model to the filename already
--  selected by the user. If no filename has been selected, the
On_saveas
--  procedure is run.
--=========================================================
   procedure On_Save_Activate
     (Object : access Gtk_Menu_Item_Record'Class);


--==========================================================
-- PROCEDURE: On_Saveas_Activate
-- INPUT: The menu item selected by the user
-- OUTPUT: None
-- PRECONDITIONS: The main window must exist
-- DESCRIPTION: This procedure creates dialog box to ask the user for a
file
--  name to save the program as. It then saves the file as the selected
filename
--=========================================================
   procedure On_Saveas_Activate
     (Object : access Gtk_Menu_Item_Record'Class);


--==========================================================
-- PROCEDURE: On_Exit_Activate
-- INPUT: The menu item selected by the user
-- OUTPUT: None
-- PRECONDITIONS: The main window must exist
-- DESCRIPTION: This procedure closes the program
--=========================================================
   procedure On_Exit_Activate
```

```
     (Object : access Gtk_Menu_Item_Record'Class);



--==========================================================
-- PROCEDURE: On_Delete_Activate
-- INPUT: The menu item selected by the user
-- OUTPUT: None
-- PRECONDITIONS: The workspace must exist
-- DESCRIPTION: This procedure removes elements from the model. If a
--  datagram has been selected in the workspace, the datagram  and all
related
-- connections are deleted from the model. If a port has been selected,
any
-- connection involving that port is removed from the model
--=========================================================
   procedure On_Delete_Activate
     (Object : access Gtk_Menu_Item_Record'Class);



--==========================================================
-- PROCEDURE: On_Cut_Activate
-- INPUT: The menu item selected by the user
-- OUTPUT: None
-- PRECONDITIONS: The workspace must exist
--                        A selected item must exist in the workspace
-- DESCRIPTION: This procedure copies the data from a selected element
--  to the workspace clipboard and deletes the element.
--=========================================================
   procedure On_Cut_Activate
     (Object : access Gtk_Menu_Item_Record'Class);



--==========================================================
-- PROCEDURE: On_Copy_Activate
-- INPUT: The menu item selected by the user
-- OUTPUT: None
-- PRECONDITIONS: The workspace must exist
--                       A selected item must exist in the workspace
-- DESCRIPTION: This procedure copies the data from a selected element
--  to the workspace clipboard
--=======================================================
   procedure On_Copy_Activate
     (Object : access Gtk_Menu_Item_Record'Class);



--========================================================
-- PROCEDURE: On_Paste_Activate
-- INPUT: The menu item selected by the user
-- OUTPUT: None
-- PRECONDITIONS: The workspace must exist
--                       There must be data in the clipboard
-- DESCRIPTION: This procedure creates a new datagram from data in the
--  clipboard and adds it to the model.
--=======================================================
   procedure On_Paste_Activate
     (Object : access Gtk_Menu_Item_Record'Class);



--========================================================
-- PROCEDURE: On_Zoom_In_Activate
-- INPUT: The menu item selected by the user
-- OUTPUT: None
-- PRECONDITIONS: The workspace must exist
-- DESCRIPTION: This procedure calls the zoom_in procedure on the
--  current workspace.
--=======================================================
   procedure On_Zoom_In_Activate
     (Object : access Gtk_Menu_Item_Record'Class);



--========================================================
-- PROCEDURE: On_Zoom_Out_Activate
-- INPUT: The menu item selected by the user
-- OUTPUT: None
-- PRECONDITIONS: The workspace must exist
-- DESCRIPTION: This procedure workspace calls the zoom_out procedure
on the
--  current workspace.
--=========================================================
   procedure On_Zoom_Out_Activate
     (Object : access Gtk_Menu_Item_Record'Class);


--==========================================================
-- PROCEDURE: On_Run_Activate
-- INPUT: The menu item selected by the user
-- OUTPUT: None
-- PRECONDITIONS: The main window must exist
-- DESCRIPTION: This procedure runs the print-out function
--=========================================================
   procedure On_Run_Activate
     (Object : access Gtk_Menu_Item_Record'Class);


--==========================================================
-- PROCEDURE: On_New_Window1_Activate
-- INPUT: The menu item selected by the user
-- OUTPUT: None
-- PRECONDITIONS: The main window must exist
-- DESCRIPTION: This procedure is a stub for future development
--=========================================================
   procedure On_New_Window1_Activate
     (Object : access Gtk_Menu_Item_Record'Class);


--==========================================================
-- PROCEDURE: On_Info1_Activate
-- INPUT: The menu item selected by the user
-- OUTPUT: None
-- PRECONDITIONS: The main window must exist
-- DESCRIPTION: This procedure displays the popup notifying the
--  user that there is no help
--========================================================
   procedure On_Info1_Activate
     (Object : access Gtk_Menu_Item_Record'Class);



--=========================================================
-- PROCEDURE: On_About1_Activate
-- INPUT: The menu item selected by the user
-- OUTPUT: None
-- PRECONDITIONS: The main window must exist
-- DESCRIPTION: This procedure displays the popup with
--  information on the author and version of the program
--========================================================
   procedure On_About1_Activate
     (Object : access Gtk_Menu_Item_Record'Class);

      ------------------------
      -- Button Bar section --
      ------------------------

--=========================================================
-- PROCEDURE: On_Select_Button_Clicked
-- INPUT: The button clicked on by the user
-- OUTPUT: None
-- PRECONDITIONS: The main window must exist
-- DESCRIPTION: This procedure changes the edit strip
--  to show the current model/diagram information and
--  adds a accept chages button to allow modification of
--  the diagram. As long as this button remains selected, the
--  edit strip will show the current information of the
--  selected datagram and the accept changes button for
--  modification
--========================================================
   procedure On_Select_Button_Clicked
     (Object : access Gtk_Button_Record'Class);
```

```
--==========================================================
-- PROCEDURE: On_Function_Button_Clicked
-- INPUT: The button clicked on by the user
-- OUTPUT: None
-- PRECONDITIONS: The main window must exist
-- DESCRIPTION: This procedure changes the edit strip to
--  provide diagram call inputs to create diagram calls
--  on the screen. While this button is selected, any click
--  on the workspace will place a diagram call there.
--=========================================================
   procedure On_Function_Button_Clicked
     (Object : access Gtk_Button_Record'Class);

--==========================================================
-- PROCEDURE: On_Switch_Button_Clicked
-- INPUT: The button clicked on by the user
-- OUTPUT: None
-- PRECONDITIONS: The main window must exist
-- DESCRIPTION: This procedure changes the edit strip to
--  provide switch inputs to create a switch
--  on the screen. While this button is selected, any click
--  on the workspace will place a switch there.
--=========================================================
   procedure On_Switch_Button_Clicked
     (Object : access Gtk_Button_Record'Class);

--==========================================================
-- PROCEDURE: On_Data_Box_Button_Clicked
-- INPUT: The button clicked on by the user
-- OUTPUT: None
-- PRECONDITIONS: The main window must exist
-- DESCRIPTION: This procedure changes the edit strip to
--  provide data box inputs to create a data box
--  on the screen. While this button is selected, any click
--  on the workspace will place a data box there.
--=========================================================
   procedure On_Data_Box_Button_Clicked
```

```
     (Object : access Gtk_Button_Record'Class);

--=========================================================
-- PROCEDURE: On_Fork_Button_Clicked
-- INPUT: The button clicked on by the user
-- OUTPUT: None
-- PRECONDITIONS: The main window must exist
-- DESCRIPTION: This procedure changes the edit strip to
--  provide fork inputs to create a fork
--  on the screen. While this button is selected, any click
--  on the workspace will place a fork there.
--========================================================
   procedure On_Fork_Button_Clicked
     (Object : access Gtk_Button_Record'Class);

--=========================================================
-- PROCEDURE: On_Merge_Button_Clicked
-- INPUT: The button clicked on by the user
-- OUTPUT: None
-- PRECONDITIONS: The main window must exist
-- DESCRIPTION: This procedure changes the edit strip to
--  provide merge inputs to create a merge
--  on the screen. While this button is selected, any click
--  on the workspace will place a merge there.
--========================================================
   procedure On_Merge_Button_Clicked
     (Object : access Gtk_Button_Record'Class);

--=========================================================
-- PROCEDURE: On_Alternate_Button_Clicked
-- INPUT: The button clicked on by the user
-- OUTPUT: None
-- PRECONDITIONS: The main window must exist
-- DESCRIPTION: This procedure changes the edit strip to
--  provide alternate pattern inputs to create an alternate
--  pattern on the screen. While this button is selected, any click
--  on the workspace will place an alternate pattern there.
```

```
--=======================================================
   procedure On_Alternate_Button_Clicked
     (Object : access Gtk_Button_Record'Class);

--========================================================
-- PROCEDURE: On_Rule_Button_Clicked
-- INPUT: The button clicked on by the user
-- OUTPUT: None
-- PRECONDITIONS: The main window must exist
-- DESCRIPTION: This procedure changes the edit strip to
--  provide rule inputs to create a rule
--  on the screen. While this button is selected, any click
--  on the workspace will place a rule there.
--=======================================================
--     procedure On_Rule_Button_Clicked
--       (Object : access Gtk_Button_Record'Class);

--========================================================
-- PROCEDURE: On_Pattern_Box_Button_Clicked
-- INPUT: The button clicked on by the user
-- OUTPUT: None
-- PRECONDITIONS: The main window must exist
-- DESCRIPTION: This procedure changes the edit strip to
--  provide pattern box inputs to create a pattern box
--  on the screen. While this button is selected, any click
--  on the workspace will place a pattern box there.
--=======================================================
   procedure On_Pattern_Box_Button_Clicked
     (Object : access Gtk_Button_Record'Class);

--========================================================
-- PROCEDURE: On_Pattern_Group_Button_Clicked
-- INPUT: The button clicked on by the user
-- OUTPUT: None
-- PRECONDITIONS: The main window must exist
-- DESCRIPTION: This procedure changes the edit strip to
--  provide pattern group inputs to create a  pattern group
--  on the screen. While this button is selected, any click
--  on the workspace will place a  pattern group there.
--========================================================
--     procedure On_Pattern_Group_Button_Clicked
--       (Object : access Gtk_Button_Record'Class);

--=========================================================
-- PROCEDURE: On_Connector_Button_Clicked
-- INPUT: The button clicked on by the user
-- OUTPUT: None
-- PRECONDITIONS: The main window must exist
-- DESCRIPTION: This procedure changes the edit strip to
--  provide connector inputs to create a connector
--  on the screen. While this button is selected, any click
--  on the workspace will place a connector there.
--========================================================
   procedure On_Connector_Button_Clicked
     (Object : access Gtk_Button_Record'Class);

--=========================================================
-- PROCEDURE: On_Data_Flow_Button_Clicked
-- INPUT: The button clicked on by the user
-- OUTPUT: None
-- PRECONDITIONS: The main window must exist
-- DESCRIPTION: This procedure clears the edit strip
--  While this button is selected, a data flow connector
--  may be placed between any two available source and
--  destination ports on  the diagram
--========================================================
   procedure On_Data_Flow_Button_Clicked
     (Object : access Gtk_Button_Record'Class);

--=========================================================
-- PROCEDURE: On_Stream_Button_Clicked
-- INPUT: The button clicked on by the user
-- OUTPUT: None
-- PRECONDITIONS: The main window must exist
-- DESCRIPTION: This procedure clears the edit strip
--  While this button is selected, a stream connector
--  may be placed between any two available source and
--  destination ports on the diagram
--=========================================================
   procedure On_Stream_Button_Clicked
     (Object : access Gtk_Button_Record'Class);

--==========================================================
-- PROCEDURE: On_Association_Button_Clicked
-- INPUT: The button clicked on by the user
-- OUTPUT: None
-- PRECONDITIONS: The main window must exist
-- DESCRIPTION: This procedure changes the edit strip to
--  provide the association input.  While this button is
--  selected, an association connector may be placed between
--  any two available source and destination ports on the diagram
--=========================================================
   procedure On_Association_Button_Clicked
     (Object : access Gtk_Button_Record'Class);

end Window_main.Callbacks;
```

A-9 EDIT_STRIP.ADS

```
--==========================================================
-- FILE: diagram.ads
-- AUTHOR: Steven Carpenter
-- VERSION: 1.2
-- LAST MODIFIED: 23 September, 2002
-- DESCRIPTION: This file handles the edit strip functions of the
--  V graphical editor. It creates the strip with the proper labels
--  and input segments based on . It responds to commands to change
--  the strip to match the currently selected button. It also provides
--  the ability to show default values based on inputs to the calls.
--  Output abilities include the ability to read any of the values
--  entered by the user in the input segments.
-- COMPILER: GNAT 3.13p
-- WARNINGS: None
--==========================================================

with Gtk.Frame; use Gtk.Frame;
with Gtk.Box; use Gtk.Box;
with Gtk.Label; use Gtk.Label;
with Gtk.Adjustment; use Gtk.Adjustment;
with Gtk.Spin_Button; use Gtk.Spin_Button;
with Gtk.Button; use Gtk.Button;
with Gtk.GEntry; use Gtk.GEntry;
with Glib; use Glib;

package Edit_Strip is

   type Edit_Strip_Record is new Gtk_Frame_Record with null record;
   type Edit_Strip is access all Edit_Strip_Record'Class;


--==========================================================
-- PROCEDURE: Gtk_New
-- INPUT: None
-- OUTPUT: A new instance of an edit strip
-- PRECONDITIONS: None
-- DESCRIPTION: This procedure creates a new edit strip
--=========================================================
   procedure Gtk_New( Frame : out Edit_Strip );


--==========================================================
-- FUNCTION: Is_Function
-- INPUT: A variable of type Edit_Strip
-- OUTPUT: True if the edit strip is currently representing a
--  function, False otherwise.
-- PRECONDITIONS: An existing edit strip
-- DESCRIPTION: This function checks to see if edit_function was
--  the last edit command(non-get).
--=========================================================
   function Is_Function(Bar : access Edit_Strip_Record'Class)
                        return Boolean;



--==========================================================
-- FUNCTION: Get_Expr_Type
-- INPUT:  A variable of type Edit_Strip
-- OUTPUT: The string representing the expression type
-- PRECONDITIONS:  An existing edit strip
-- DESCRIPTION: This funtion returns either a user entered string
--  or the default string if nothing is entered in the expression type
--  field. The default string is either the empty string or the
--  current expression type if the edit strip was given one. The
--  meaning of the string is based on the diagram type. For a pattern
--  box or data box, this is the data type. For all other diagram
--  types this function returns the empty string.
--=========================================================
   function Get_Expr_Type(Bar : access Edit_Strip_Record'Class)
                          return String;



--==========================================================
-- FUNCTION: Get_Expr
-- INPUT: A variable of type Edit_Strip
-- OUTPUT:  The string representing the expression
-- PRECONDITIONS: An existing edit strip
-- DESCRIPTION:  This funtion returns either a user entered
--  string or the default string if nothing is entered in the
--  expression field. The default string is either the empty string
--  or the current expression if the edit strip was given one. The
--  meaning of the string is based on the diagram type. For a
--  pattern box or data box, this is the data or pattern value.
--  For rules/functions it is the function name. For switches it
--  is the boolean expression. For a connection it is the
--  connector name, and for an association, it is the association.
--  For all other diagram types this function returns the empty string.
--=========================================================
   function Get_Expr(Bar : access Edit_Strip_Record'Class)
                     return String;



--==========================================================
-- FUNCTION:  Get_Iports
-- INPUT: A variable of type Edit_Strip
-- OUTPUT: The number of input ports the user specified or default
-- PRECONDITIONS: An existing edit strip
-- DESCRIPTION: This function provides the number of input ports
--  required by a diagram. If the current edit strip is for a
--  rule/function, box, fork/merge/alt, or switch, it returns the
--  user entered value or the current value. For all others it returns
--  one.
--=========================================================
   function Get_Iports(Bar : access Edit_Strip_Record'Class)
                       return Gint;



--==========================================================
-- FUNCTION: Get_Labels
-- INPUT: A variable of type Edit_Strip
-- OUTPUT: A string containing labels
-- PRECONDITIONS: An existing edit strip
-- DESCRIPTION: This function returns the user entered value of
--  the labels field or the default value for boxes and switches. For
--  all others it returns the empty string.
--=========================================================
   function Get_Labels(Bar : access Edit_Strip_Record'Class)
                       return String;



--==========================================================
-- FUNCTION: Get_Oports
-- INPUT: A variable of type Edit_Strip
-- OUTPUT: The number of output ports the user specified or default
-- PRECONDITIONS: None An existing edit strip
-- DESCRIPTION: This function provides the number of output ports
--  required by a diagram. If the current edit strip is for a
--  rule/function, box, or fork/merge/alt it returns the user entered
--  value or the current value. For all others it returns one.
--=========================================================
   function Get_Oports(Bar : access Edit_Strip_Record'Class)
                       return Gint;



--==========================================================
-- PROCEDURE: Edit_Select
-- INPUT: A variable of type Edit_Strip
-- OUTPUT: None
-- PRECONDITIONS: An existing edit strip
-- DESCRIPTION: This procedure changes the input edit strip to
--  show nothing. The previous widgets inhabiting the strip are
removed.
--=========================================================
   procedure Edit_Select( Bar : access Edit_Strip_Record'Class);



--==========================================================
-- PROCEDURE: Edit_Function
-- INPUT: A variable of type Edit_Strip,
--        A string for the diagram name, default is "Untitled",
--        A Gfloat for the number of input ports, default is zero,
--                 Minimum value is zero, maximum is ten.
--        A Gfloat for the number of output ports, default is zero.
--                 Minimum value is zero, maximum is ten.
-- OUTPUT: None
-- PRECONDITIONS: An existing edit strip
-- DESCRIPTION: This procedure changes the input edit strip to
--  show input fields for a diagram call. The previous widgets
--  inhabiting the strip are removed. New widgets are added for
--  entering a name(String), input ports, and output
ports(spinbuttons).
--=========================================================
   procedure Edit_Function(Bar : access Edit_Strip_Record'Class;
                           Fcall_User_Name : String := "Untitled";
                           Fcall_User_Ip   : Gfloat := 0.0;
                           Fcall_User_Op   : Gfloat := 0.0);



--==========================================================
-- PROCEDURE: Edit_Switch
-- INPUT: A variable of type Edit_Strip,
--        A string for the expression, default is the empty string,
--        A string for the port labels, default is the empty string,
--        A Gfloat for the number of input ports, default is one.
--                 Minimum value is one, maximum is ten.
-- OUTPUT: None
-- PRECONDITIONS: An existing edit strip
-- DESCRIPTION: This procedure changes the input edit strip to
--  show input fields for a switch. The previous widgets inhabiting
--  the strip are removed. New widgets are added for entering an
--  expression(String), input ports(spinbutton), and input
--  labels(String).
--=========================================================
   procedure Edit_Switch( Bar : access Edit_Strip_Record'Class;
                           Switch_User_Expr  : String := "";
                           Switch_User_Names : String := "";
                           Switch_User_Ports : Gfloat := 1.0);



--==========================================================
-- PROCEDURE: Edit_Boxes
-- INPUT: A variable of type Edit_Strip
--        A string for the value type, default is the empty string,
--        A string for the value, default is the empty string,
--        A Gfloat for the number of input ports, default is one.
--                 Minimum value is zero, maximum is ten.
--        A string for the port labels, default is the empty string,
--        A Gfloat for the number of input ports, default is one.
--                 Minimum value is zero, maximum is ten.
-- OUTPUT: None
-- PRECONDITIONS: An existing edit strip
-- DESCRIPTION: This procedure changes the input edit strip to
--  show input fields for data or pattern boxes. The previous widgets
--  inhabiting the strip are removed. New widgets are added for
--  entering a type(String), value(String), input ports(spinbutton),
--  input labels(String), and output ports(spinbutton).
--=========================================================
   procedure Edit_Boxes( Bar : access Edit_Strip_Record'Class;
                                 Dbox_User_Type  : String := "";
                                 Dbox_User_Expr  : String := "";
                                 Dbox_User_Ip    : Gfloat := 1.0;
                                 Dbox_User_Names : String := "";
                                 Dbox_User_Op    : Gfloat := 1.0);


--==========================================================
-- PROCEDURE: Merge_Fork
-- INPUT: A variable of type Edit_Strip,
--        A Gfloat for the number of ports, default is two.
--                 Minimum value is two, maximum is ten.
-- OUTPUT: None
-- PRECONDITIONS: An existing edit strip
-- DESCRIPTION: This procedure changes the input edit strip to
--  show input fields for merge, fork, and alternate diagrams. The
--  previous widgets inhabiting the strip are removed. New widgets
--  are added for entering the number of ports(spinbutton).
--=========================================================
   procedure Merge_Fork( Bar : access Edit_Strip_Record'Class;
                                 Fork_User_Ports : Gfloat := 2.0);


--==========================================================
-- PROCEDURE: Edit_Connection
-- INPUT: A variable of type Edit_Strip.
-- OUTPUT: None
-- PRECONDITIONS: An existing edit strip
-- DESCRIPTION: This procedure changes the input edit strip to
--  show input fields for connections or associations. The previous
--  widgets inhabiting the strip are removed. New widgets are added
--  for entering an expression(String).
--=======================================================
   procedure Edit_Connection( Bar : access Edit_Strip_Record'Class);

end Edit_Strip;
```

LIST OF REFERENCES

[1] Luqi,V.Berzins, R.Yeh, "A Prototyping Language for Real-Time Software", IEEE *Transactions on Software Engineering*, October 1988, pp1409-1423

[2] S.-K.Chang, *Principles of Visual Programming Systems,* Prentice Hall, Englewood Cliffs, New Jersey, 1990.

[3] P.T.Cox, F.R.Gilles, T.Pietrzykowski, "Prograph", *in Visual Object-Oriented Programming, Concepts and Environments,*(ed. M.Burnett, A.Goldberg, T.Lewis), Manning 1995, pp45-66

[4] E.Baroth, C.Hartsough, "Visual Programming in the Real World", *in Visual Object-Oriented Programming, Concepts and Environments,*(ed. M.Burnett, A.Goldberg, T.Lewis), Manning 1995, pp21-42

[5] M.Auguston, "The V Experimental Visual Programming Language", Computer Science Department, *Technical report NMSU-CSTR-9611,* New Mexico State University, October,1996

[6] M.Auguston, V.Berzins, B.Bryant, "Visual Meta-Programming Language", *Proceedings of OOPSLA 2001 Workshop on Domain-Specific Visual Languages,* Tampa, Florida, October 14, 2001, pp.69-82

[7] M.Auguston, "Visual Meta-Program Notation", *Proceeding of the Monterey Workshop "Engineering Automation for Software Intensive System Integration",* Monterey, California, June 18-22, 2001.

[8] A.Bower, B.McGlashan, "Twisting the Triad", http://www.object-arts.com/Papers/TwistingTheTriad.PDF

[9] M.Potel, "MVP: Model-View-Presenter The Taligent Programming Model for C++ and Java", ftp://www6.software.ibm.com/software/developer/library/mvp.pdf, 1996

[10] E.Briot, J.Brobecker, A.Charlet, *GtkAda User's Guide,* Version 1.2.12, Document revision level 1.102.2.1, 2001.

[11] E.Briot, J.Brobecker, A.Charlet, *GtkAda Reference Manual,* Version 1.2.12, Document revision level 1.15, 2001.

[12] Ada Core Technologies, Inc, *GNAT Reference Manual, GNAT the GNU Ada 95 Compiler,* Version 1.13p, Document revision level 1.135, 2000.

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Fort Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Professor Dan Boger
   Naval Postgraduate School
   Monterey, California

4. RADM Winsor Whiton
   Naval Security Group Command
   Fort Mead, Maryland

5. Dr. Mikhail Auguston
   Computer Science Department
   New Mexico State University
   Las Cruces, New Mexico

6. Richard_Riehle
   Naval Postgraduate School
   Monterey, California