A Graphical Editor for OSU v3.0

By Fangchen Lin

A research paper submitted in partial fulfillment of the requirements for the degree of Master of Science

Major Professor : Dr. T. G. Lewis
Minor Professor : Dr. T. Budd

Department of Computer Science
Oregon State University
Corvallis, Oregon 97331

March 16, 1992

Acknowledgements

I would like to express my appreciation to my major professor, Dr. T.G. Lewis, for giving me the opportunity to work on this project. His guidance, understanding, encouragement and helpful discussion has been instrumental to my progress. I would also like to thank my Minor Professor, Dr. Timothy Budd, for his helpful comments. I am grateful to Dr. Bruce D'Ambrosio for his willingness to participate on my Graduate Committee.

I thank Huan Chao Keh and Chung-Cheng Luo, who bring a clear direct and interactive discuss during developing this project. I thank other members of the OSU v3.0 development team: Chih Lai, Walter Wittel, Kangho Lee, Tong Li, Kee Yun Chan, and Hueii Huang for their discussion.

Lastly, I would like to express my gratitude to my parents, brothers in Taiwan for their support and encouragement, and, to Fu-Lung Chang, for his patience and encouragement to keep me going when the going got tough.

Table of Contents

Abstract 1
1. Introduction 2
1.1. Statement of the Problem 2
1.2. Approaches to the Problem 3
1.3. Results of the Study 5
2. Petri-Net-Based Object-Oriented Conceptual Model for Direct-Manipulation User Interface Systems 8
2.1. Formal Definition of a Petri Net 8
2.2. Direct-Manipulation User Interface Modeling with Annotated Petri Nets 11
2.3. From Net Model to an Executable Object-Oriented Model 18
2.4. Translation of Annotated Petri Nets into C++ Programs 27
3. Use of the Annotated Petri Net Editor 29
3.1. File Actions 31
3.2. Creating a Petri Net 34
3.3. Editing the Petri Net 44
3.4. Syntax Checks 47
3.5. Code Generation 50
4. Design and Implementation of the Petri Net Editor 51
4.1. User Interface Classes 53
4.2. Petri Net Storage Structure and Object Classes 58
4.3. Graphics Object Classes 70
4.4. PNGraphicsView Class 76
5. Experience with the Petri Net Editor 88
5.1. MiniDraw 88
5.2. Help System 93
5.3. Calculator 96
5.4. Record Query 103
5.5. Statistics 108
6. Conclusion 113
7. Appendix 115
8. References 127

Abstract

Development of graphical user interface (GUI) applications is difficult since the process can be both complicated and tedious. We propose a solution directed at reducing programming time and effort required to build a GUI application. Our solution is based on the Petri Network, the Oregon SpeedCode Universe (OSU) Application Framework, and the OSU Browser (v. 3.0). A Petri Network is a visual programming language which is used represent the sequencing of objects and messages. The Application Framework provides reusable components in the form of objects. The Browser provides a visual way to examine a system in search of reusable components.

A Petri Net editor was constructed which incorporates a code generator and browser. This editor uses direct-manipulation to simplify coding tasks, accepting specifications from the developer and generating the internal representations of the Petri Net. The internal representation is input to the Code Generator, thus generating an OSU Application Framework-based C++ program as output.

Using the Petri Net editor to generate four application programs ; 1) drawing program, 2) a help system, 3) a calculator, and 4) a record query system, it is estimated that programming time has been reduced by 90% and programming effort has been reduced by 79%.

1. Introduction

This report describes a method of visual programming based on Petri Network, the Oregon SpeedCode Universe 3.0 (OSU v3.0) Application Framework, and the OSU Browser. A Petri Net (PN) editor is described which provides a visual programming language for the expression of objects and message flows within a desired application. We describe an implementation of the PN editor for writing application programs for the Macintosh computer.

1.1. Statement of the Problem

The development of graphical user interface (GUI) applications poses the following types of programming problems:

1) GUI programmers must learn and use complex, low-level toolbox routines;

2) Highly interactive interfaces are among the hardest to create, since they must handle at least two asynchronous input devices, real-time feedback, multiple windows, and elaborate, dynamic graphics [Myers 90];

3) GUI programmers must handle all input events at a low level;

4) Intertwined interactions exist between the user interface and the application logic (i.e., change propagation); and

5) GUI programmers must rewrite common but tedious routines for each new application.

The development of a PN editor does not provide solutions for all of these problems. However, it does address the restricted problem of sequencing linked structures for user interfaces, while providing the ability to describe all of the possible execution paths that a user may follow through the application interface: The problem addressed in this report is the provision of an object-oriented conceptual model and a visual programming language which can be used by programmers to construct GUI applications. The goals of this study are to reduce the time and effort required for a programmer using this system as compared to the same programmer using C++ as the only means of programming.

1.2. Approaches to the Problem

The approach adopted is based on the following assumptions:

1) Visual programming based on Petri Network eases program design, implementation, and reuse;

2) Graphical programming with Petri Network provides a high-level model for program integration, specification, modeling, design, validation, simulation, and rapid prototyping;

3) An application framework provides the skeletal structure for an application and reduces programming time and effort to implement the domain specific parts; and

4) A browser allows programmers to easily view and seek reusable components.

Petri Network was selected as the visual programming language because of their underlying mathematical structure which can be easily understood. The PN editor provides a capability for modeling high-level designs since, through the use of an object-oriented application framework, it is capable of modeling both the static and dynamic aspects of a GUI application at high levels of abstraction. In turn, the OSU 3.0 Application Framework (the OSU Framework) was selected as the basis for visual programming because of its small size and the fact that it is in many respects easier to use than *MacApp* [Keh 91]. Thus, this tool embodies most of the generic functionality required for the construction of a GUI application. The OSU Browser provides a visual programming assistance tool that can be invoked internally, from within a programming editor, to provide a graphical view of the class hierarchies for an entire application. Thus, the designer has only to exercise direct choice of appropriate messages and to paste them into the application, providing the basic means to both investigate and to reuse components.

1.3. Results of the Study

A Petri net editor (Figure 1.1.) was constructed, based on the following principles: 1) Integration with the Browser; 2) incorporation of a code generator; and 3) the use of direct manipulation to simplify the coding task. Using this editor, four applications were implemented: 1) help system, 2) calculator example, 3) record query example and 4) *MiniDraw*. The time and effort required to implement these four examples is summarized as follows.

	MiniDraw	Help System	Calculator	Query Record	Average
*lines of codes generated	529	327	285	283	356
*MM (effort)	0.58	0.35	0.30	0.30	0.38
*TDEV (time)	2.03	1.67	1.59	1.58	1.72
*total lines of codes	545	330	395	549	455
*MM (effort)	0.6	0.35	0.42	0.6	0.49
*TDEV (time)	2.05	1.68	1.81	2.06	1.9
*%saving in effort	96%	99%	71%	50%	79%
*%saving in time	99%	99%	87%	76%	90%
*number of places	7	7	1	5	5
*number of transitions	20	18	20	9	16.75
*number of arcs	66	26	38	25	38.75
*number of messages	50	0	61	30	35.25

As shown in the above table, this approach reduced programming time by a factor of 90 percent and the effort required to complete this task by approximately 79 percent. In addition, the Application Framework was used to implement the Petri Net editor itself.

The Petri net editor consists of 6,695 lines of code. Without the reused 33 objects and 262 methods (see Appendix A) from the OSU Application Framework, the size of code will be at least 10,000 lines of code.

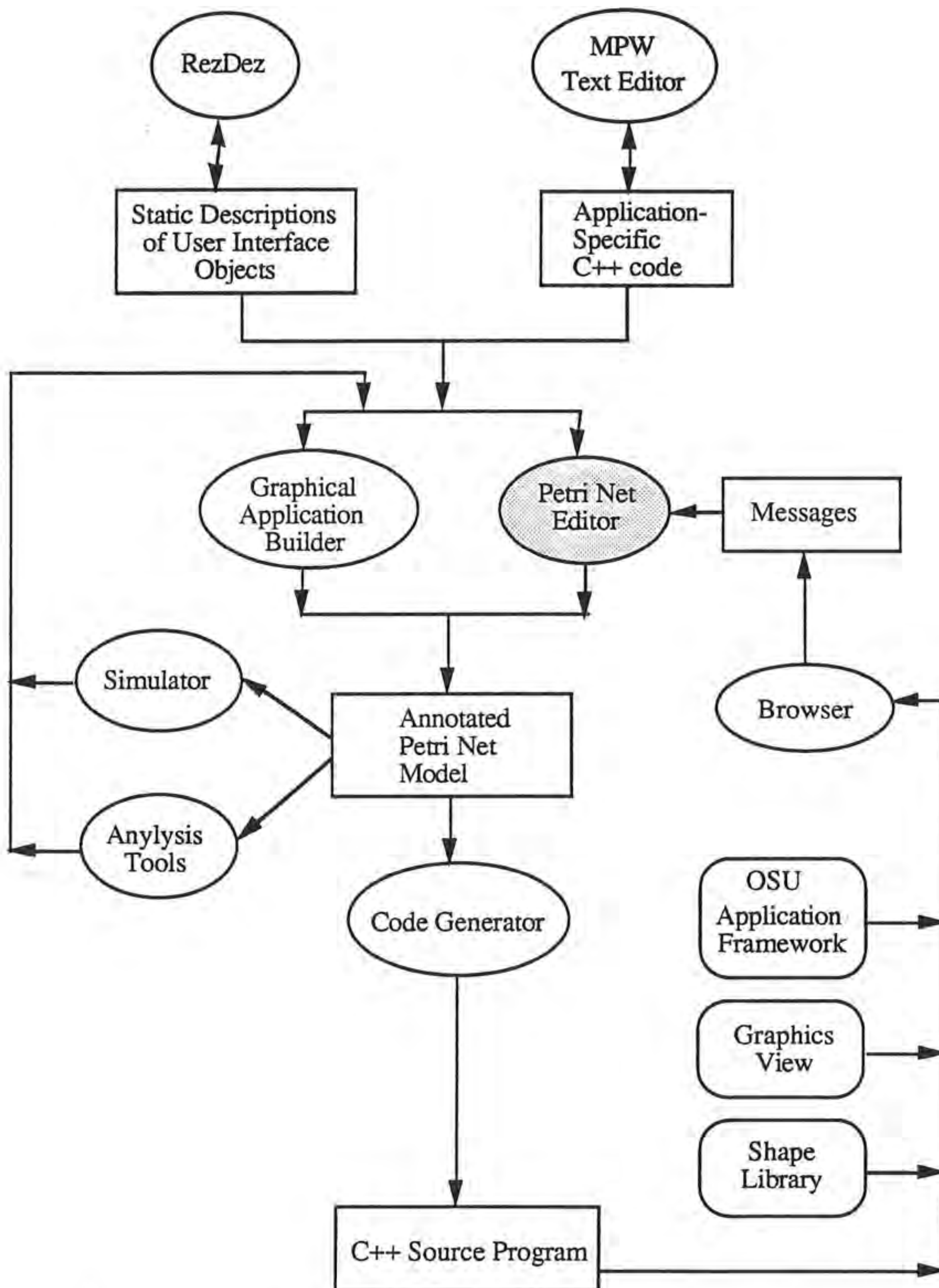


Figure 1.1. Architecture of OSU 3.0.

2. Petri-Net-Based Object-Oriented Conceptual Model for Direct-Manipulation User Interface Systems

In this chapter, we describe the use of an object-oriented conceptual model for the construction of a direct-manipulation GUI application. First, the formal Petri net structure and its rules are briefly described in Section 1. In Section 2, we illustrate the use of an annotated Petri net to represent the relationships that link individual user interface objects together in a GUI application. In Section 3, the process for the adaptation of an annotated Petri net for an object-oriented GUI application model, via the OSU Application Framework, is considered. Finally, a brief description of the logical structure of the annotated Petri net, as translated into C++ source code, is presented in Section 4.

2.1. Formal Definition of a Petri Net

Petri Net Structure

A Petri net is composed of four parts: a set of places, P, a set of transitions, T, an input function, I, and an output function, O. The input functions maps from a transition, tj, to a collection of places, $I(tj)$, identified as the input places of the transition. In turn, the output function maps a transition to a collection of places, $O(tj)$, identified as the output places of the transition [Peterson 81].

Petri Net Graphs

Most theoretical considerations of Petri nets are based on the formal definitions of Petri net structures provided above. However, a graphical representation for a Petri net structure is much more useful for illustrating the concepts of Petri net theory. A Petri net graph reflects two types of nodes, where circles represent places and bars represent transitions (Figure 2.1). Directed arcs (arrows) connect places and the transitions, some of which are directed from places to transitions and vice versa for the remainder. One type of arc, the input arc, is directed from the place pi to the transition tj to define this place as an input place of the transition; conversely, the output arc is directed from the transition tj to the place pi to define this place as an output place of the transition.

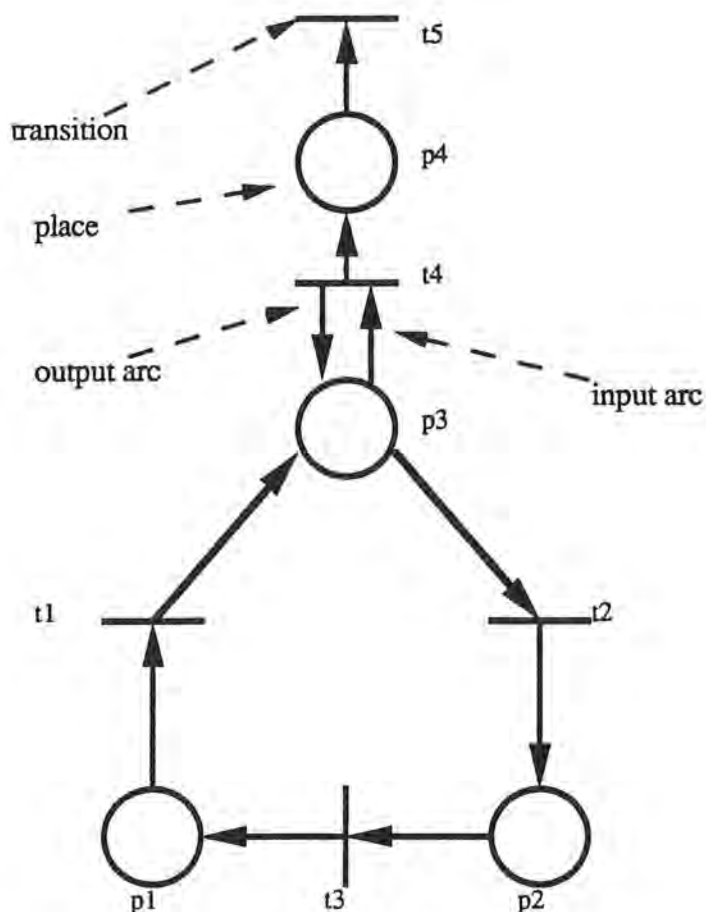


Figure 2.1 Petri net conceptualization.

Petri Net Markings

A marking, *u*, assigns tokens to places within the Petri net. In this sense, a *token* is a primitive concept, assigned to and regarded as residing within the places of a Petri net. The number and the positions of tokens may change during the execution of a Petri net.

Execution Rules for Petri Nets

The execution of a Petri net is controlled by the number and distribution of tokens in the Petri net, and is executed by *firing* transitions. A transition *fires* by

removing tokens from input places and creating new tokens for distribution to output places. A transition may fire when it is *enabled.* A transition is enabled if each of its input places has at least as many tokens in it as arcs from the place to the transition. Upon firing, a transition removes all of its enabling tokens from its input places, then deposits into each of its output places one token for each arc from the transition to the place. Transition firings can continue as long as at least one enabled transition exists.

2.2. Direct-Manipulation User Interface Modeling with Annotated Petri Nets

In this section, the use of an annotated Petri net to describe an application program sequence of operations, reflecting all of the possible execution paths for a direct-manipulation user interface system, is considered. To translate the model into a useful program, selected annotations is added to the formal definition of a Petri net. An example of an annotated Petri Net representation for a GUI application is shown in Figure 2.2.

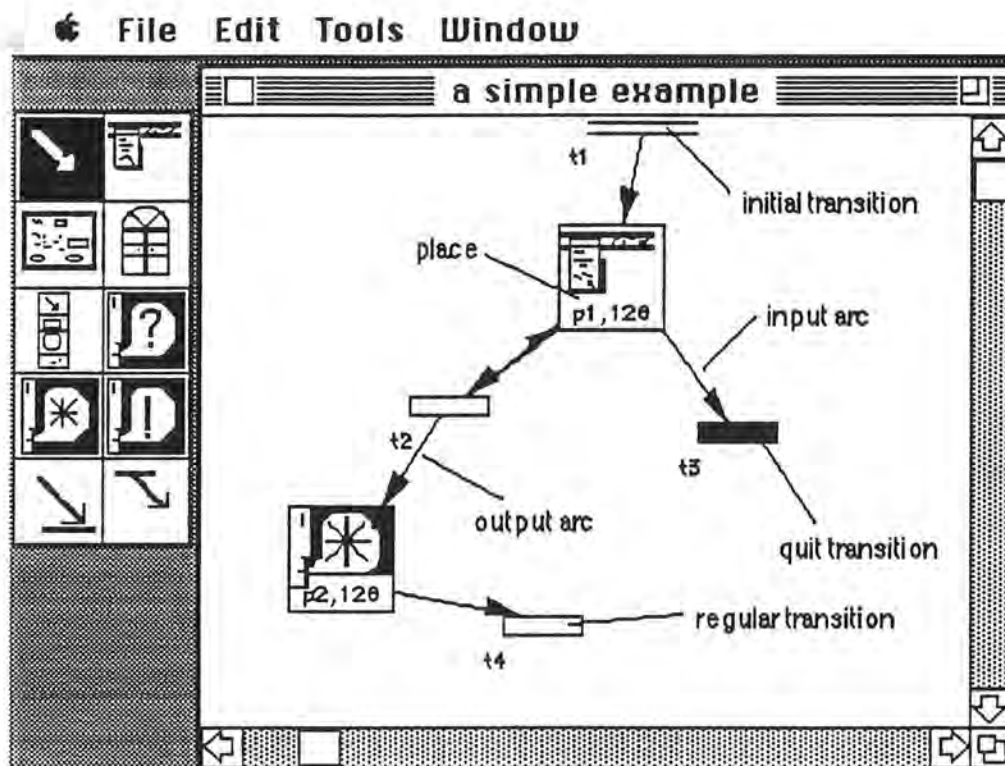


Figure 2.2 Petri net representation for a simple example.

For GUI applications, places are used to represent objects. From Figure 2.2, it may be noted that icons are used to represent places, thus each menu, dialog box, window, or alert has a unique icon. Transitions represent mouse actions performed on selectable areas within GUI objects, and are represented as rectangles in Figure 2.2. The input to a transition defines when the transition can be fired. Transition firing corresponds to user actions performed on one of the selectable GUI items and the actions undertaken by the application are described in the firing transition as output arcs. Following these user actions, the new application state (i.e., the application postcondition) is represented by places connected to the output arcs of the transition. Thus, during the execution of the net, current marking determines which user interface objects are placed upon the screen. The transitions enabled for current marking determine which items associated with which user interface objects are selectable.

When a user clicks on a selectable item, one of the enabled transitions is fired, resulting in placement of a new marking and effecting appropriate changes of the interface objects.

Places in the PN Editor

In applications, places represent object types. Each place can contain more than one token, and each token represents an instance of the place. Tokens are used to simulate the execution of a Petri net. For this purpose, a set of attributes is defined for each place type and tokens carry the attributes of the place instances which they represent. The precise appearances of user interface objects displayed on the screen are determined by the current values of the attributes of their corresponding tokens, any or all of which attributes can be modified or given an initial value at the time of transition firing.

Places are classified in two sets, either modal or modeless places. A modal place represents an interface object which will place the user in state or "mode" of being able to work only inside this user interface object [Apple 85]. For Macintosh user interfaces, modal dialogs and alerts constitute modal user interface objects. In turn, a modeless object does not require the user to respond before doing anything else [Apple 85].

In the PN editor, GUI object places are drawn in the net as icons, seven of which are used to designate user interfaces objects in the Macintosh system (Figure 2.3). Each place is labeled with a unique name (e.g., p1) and a resource ID. Resource IDs are used to retrieve static descriptions of user interface objects from an application's resource file. For the PN editor, menu, palette, and window places are modeless, whereas stop alert, caution, and note alert places are modal places; however,

the modes for dialog places can be specified by the user through the selection of radio buttons (Figure 2.4).

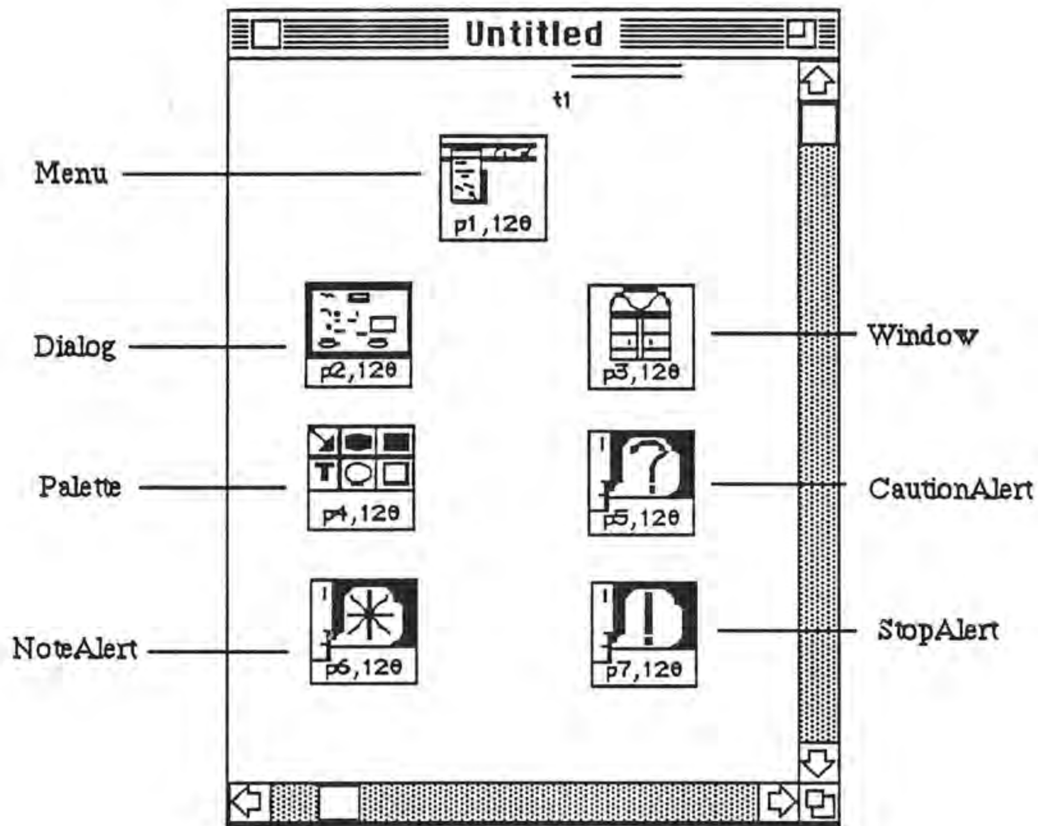


Figure 2.3 Icons for a Macintosh user interface.

Place Information

Place ID : p5

Resource Type : DIALOG

Resource ID : 128

Modal

Modeless

Instance Variables :

Cancel

OK

char *fRec
char *fName
char *fAddr
char *fPhone

Figure 2.4 Radio buttons for setting the mode of a dialog place.

Transitions in the PN Editor

Each transition represents a mouse action performed on a selectable area of an object place. The GUI object place for which the action is performed is called the owner of the transition and is connected to the transition by an input arc. Thus, a transition must be connected to at least one place by an input arc, each of which describes transition input conditions for enabling that specific transition. Therefore, an action cannot be performed until some objects exist. For example, the transition "Save" will not be performed in the "File" menu in the absence of an opened window displayed on the screen.

Within the net, transitions are drawn as boxes, each labeled with a unique name (e.g., t1). There are two types of special transitions, INIT and QUIT. The first is

represented as a double bar which, when fired, initiates (i.e., starts) an application. In turn, QUIT transitions are displayed as black rectangles, and the firing of a QUIT transition represents quitting an application. Note that INIT transitions do not have input places, whereas QUIT transitions lack output places.

Arcs in the PN Editor

Places and transitions are connected by directed arcs: Places to transitions by input arcs and transitions to places by output arcs. Input arcs are used to present the preconditions for an action to be performed on a GUI object. The preconditions for these actions consist of certain objects as well as the nonexistence of modal objects. For instance, a transition, representing the "Save" item of the "File" menu, cannot be selected in the absence of either a window or a "File" menu displayed on the screen. Moreover, the "Save" item cannot be selected if a modal object, such as a modal dialog box or an alert box, is displayed on the screen.

When a transition is fired, certain application actions are taken, some of which may change the existence of screen objects or the attributes of existing objects. This situation is represented by output arcs connected to output places, with appropriate annotations inscribed on the output arcs. Thus, following the firing of a transition, the output places connected by output arcs are the currently existing objects. The inscribed annotations may include messages, predicates, and/or sequence numbers. Messages are sent from the place which owns the fired transition to the output places for that transition, and are presented in the following formats:

- MMM — call a member function, MMM, of the output place
- RRR::MMM — call a member function, MMM, of an object, RRR ('::' is a keyword.)

• VVV=MMM	call a member function, MMM, of the output place (a returned value for the member function is assigned to the variable VVV; '=' is a keyword.)
• VVV=RRR::MMM	call a member function, MMM, of an object, RRR (a returned value for the member function is assigned to the variable, VVV; '=' and '::' are keywords)
• **func**::FFF	call a defined function, FFF ('func' and '::' are keywords)
• RRR=**func**::FFF	call a defined function, FFF (a returned value for that function is assigned to a variable, RRR; note that FFF can also be the right-hand side of an assignment statement, such as a boolean value or an arithmetic expression; '=', 'func', and '::' are keywords)

The translation rules for messages are listed in Section 2.4. Predicates are boolean expressions whose values (either true or false) depend on the current state of the net, permitting the specification of conditional flows within the net. Sequence numbers are integer constants which can be used to determine the execution order of concurrently activated objects at the moment of firing a transition.

A transition may fire if it is enabled, which condition may be met as follows: 1) Each input place with an input arc has at least one token (i.e., the firing of a transition is conditioned by the presence of tokens in each of its input places); and 2) those modal places which do not represent ownership of the transition do not have any token. Subject to these conditions, a transition fires by: 1) removing one token from each of the input places connected by input arcs; 2) evaluating the boolean expressions of predicates attached to the output arcs of the transition; 3) interpreting the messages inscribed on the output arcs; and 4) adding tokens, according to the sequence numbers inscribed on output arcs, only to those output places which are in correspondence with

the true value of the boolean expressions. Rather than removing older tokens and adding new tokens, the firing of transitions which have both inputs and outputs from the same place may consist of the modification of the values of older tokens in accordance with messages inscribed on the output arcs. Since they are integral parts of the Petri net execution process, and are used for simulation, the firing process and the tokens cannot be represented by the PN editor. Rather, output or input arcs are represented in the editor as lines ending in arrowheads. Their annotations are inscribed on output arcs as given by the user through an entering statement in an edit-text box of a popped-up dialog box provided in the editor.

2.3. From Net Model to an Executable Object-Oriented Model

The process of using the PN editor to build an executable GUI application is accomplished by the derivation of a class of new concrete objects from existing object classes in the OSU Framework, then connecting these concrete object classes through a message interface. In this section, this process is illustrated with an example record query system (Figure 2.5, 2.6). For purposes of clarity, example explanatory notes are indicated in italics.

> *This system lets the user open a personal record file. A query dialog with three editText boxes and four buttons is used to display personal records in an opened file. Each time the "Prev" button is selected, the file backs up one record . Each time the "Next" button is pressed, the file advances one record. To update the record, the user types in new data and selects the "Change" button to confirm the update.*

Figure 2.5 File menu for the record query example.

Figure 2.6 Dialog displaying record contents.

The first step is to determine the generic user interface object classes that are to be subclassed to make up the user interface portion of the application. A generic user interface object class is chosen from the OSU Framework. This is done by drawing a typed user interface place (Figure 2.7).

Five places: two menu places, a window place, a dialog place, and a note alert place are drawn.

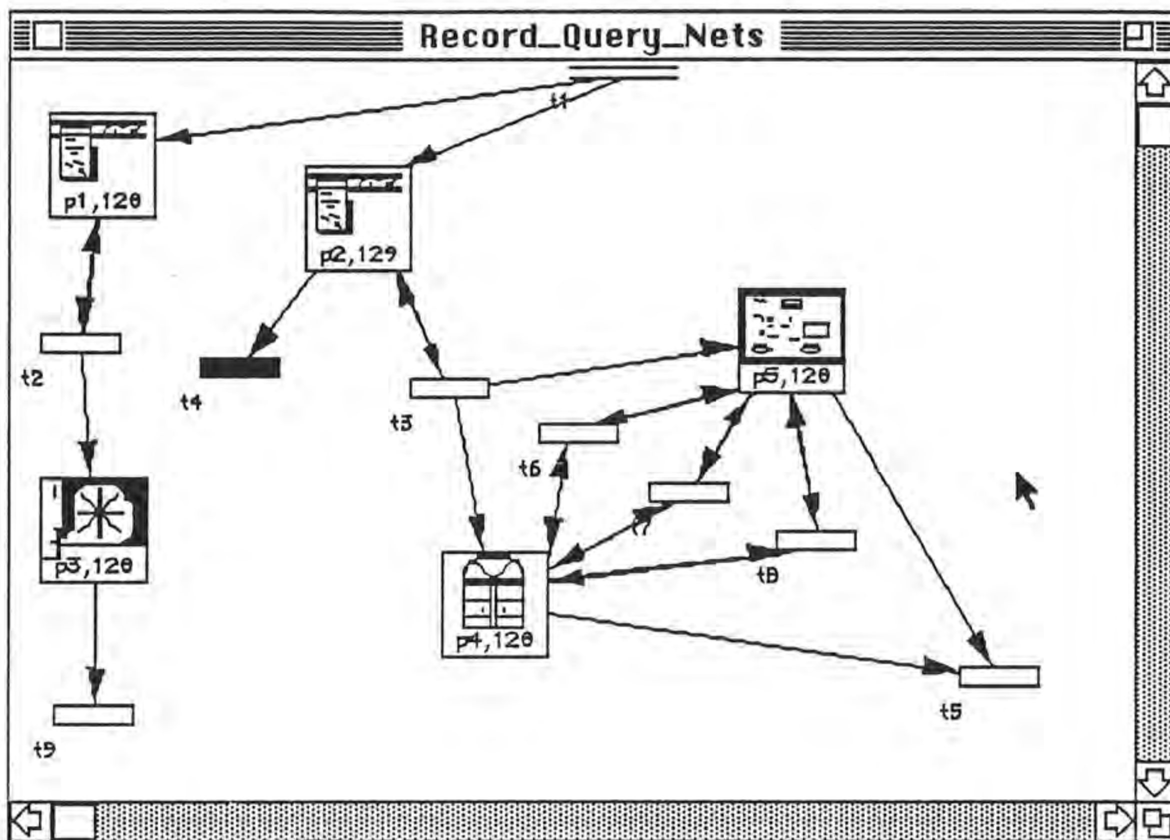


Figure 2.7 Petri net representation for a record query application.

The value of an instance variable is then specified, determining the visual characteristics of the selected user interface object class. This is done by entering a resource ID in an information dialog box brought up while a place is drawn (Figure 2.8).

> *After the drawing of a place, PN editor brings up a dialog asking for the information, e.g., the resource ID of the dialog.*

Place Information

Place ID : p5

Resource Type : DIALOG

Resource ID : 128

◉ Modal

○ Modeless

Cancel

Instance Variables :

OK

char *fRec
char *fName
char *fAddr
char *fPhone

Figure 2.8 Dialog box for request of dialog place details.

The second step is to derive new concrete classes from the generic user interface object classes selected above through the processes of subclassing and inheritance. That is, when a generic user interface object class has to be specialized into a subclass, it is customized by adding new application-specific data (i.e., instance variables) and behaviors (i.e., methods). Adding new application-specific data to a generic user interface object class can be done by declaring the variables (Figure 2.8).

There is an editText box for accepting the declarations of instance variables to store new application-specific data. For the new dialog class, four string-type variables, fRec, fName, fAddr, and fPhone, are declared: fRec stores the entire record string obtained from the opened file. After analyzing the string, three tokens are derived from the string and stored in fName, fAddr, and fPhone.

The mouse-selectable areas in the user interface object which will be used to initiate action messages to add new behaviors are then determined. Transitions are drawn for each user interface place, connecting each place to its owned transitions through input arcs.

> *For a query dialog* (Figure 2.6), *there are four buttons which will initiate action messages when they are clicked with the mouse; there are four transitions connected by input arcs from the dialog place* (Figure 2.7). *Each represents a button in the query dialog. Following creation of a transition, representing the "Next" button of the query dialog, the PN editor then pops up an information dialog* (Figure 2.9), *which asks for the attributes of a transition. In this dialog box, the user specifies the item number of the "Next" button in the dialog resource* (see Figure 2.10) *and declares the dialog place p5 to be the owner of the transition by entering the place ID of the dialog place.*

Figure 2.9 Dialog requesting transition attributes.

Figure 2.10 Resource format for the dialog box shown in Figure 2.6.

The third step is to determine message connections between the various concrete object classes: That is, which objects receive specific action messages sent by which objects. The required action is to draw an output arc connecting the transitions owned by places representing message-sending objects to those places representing message-receiving objects, and associating specific messages with specific output arcs. An object can be both the sender and receiver of an action message at the same time. This can be represented as a transition in which both the input arc and the output arc send a message from the same place (self-loop).

> *After the "Next" button is pressed, the contents of the next record are displayed in the editText box of the query dialog. To do this, the dialog place sends a message to the window place asking for the next record, which is then returned in the form of a string. The string is separated into tokens, which are then displayed on the editText boxes of the query dialog. While the "Next" button is selected, to specify the actions mentioned above, two output arcs are drawn from the transition, representing the "Next" button. The window place and the dialog place are the output places for these two output arcs* (see Figure 2.7). *To*

specify the message sent to the window place, the user double clicks the output arc connecting the transition t6 to the place p4, then a dialog box asking for the annotation of the output arc pops up (see Figure 2.11). *The message "fRec=GetNextRecord()" is typed in. For assuring that the next record is obtained before undertaking further action, the sequence number of the output arc is 1.*

OutputArc Information

Sequence # : 1

Messages :

fRec=GetNextRecord()

Cancel

OK

Predicate :

Figure 2.11 A dialog for asking annotations on an output arc.

Since the query dialog is still on the screen after the "Next" button is clicked, an output arc is drawn back to the dialog place (see Figure 2.7). *To analyze and display the record string obtained from the file, the following messages are inscribed on the output arc and the sequence number of the output arc is set at "2"* (see Figure 2.12)*:*

fName=func::GetToken(fRec,1)

fAddr=func::GetToken(fRec,2)

fPhone=func::GetToken(fRec,3)

SetItemText(5, fName)

SetItemText(6, fAddr)

SetItemText(7, fPhone)

OutputArc Information

Sequence # : 2

Messages :

fName=func::GetPattern(fRec,1)
fAddr=func::GetPattern(fRec,2)
fPhone=func::GetPattern(fRec,3)
SetItemText(5,fName)
SetItemText(6,fAddr)

Cancel

OK

Predicate :

Figure 2.12 Dialog requesting annotation for an output arc.

It follows then that

> *the contents of a record are derived from fRec and are stored in three variables. "func" is a keyword notifying the Code Generator that this is a defined function instead of a member function (method); "GetToken" requires two parameters, a string and an integer* n,. *parsing the string and returning the* n^{th} *token of the string; and "SetItemText" is a method defined in the CLDialog class, display a string for an item in a dialog box. Those methods defined in OSU Framework can be viewed and copied by using the OSU Browser. For details on invoking the Browser, the reader may refer Section 3.2 (page 37-40).*

When a transition is not connected to its input place through an output arc, a message, *DoClose()*, from a corresponding object to itself is implied:

> *The transition t5, indicating the "Ok" button, is not connected to its input places through output arcs, the dialog place, or the window place. This means that the query dialog and the opened file is closed after the "Ok" button is pressed.*

After completing the three steps, the PN editor generates an executable specification (i.e., an annotated Petri net model). This specification can be saved and used by the Code Generator to generate C++ programs based upon the Application Framework.

> *After the entire application is specified by the above steps, PN editor builds a Petri net model for the application. The model can be saved* (see Figure 2.13) *for later use. To generate source programs, the user selects "Generate Code" from the "Tools" menu* (see Figure 2.14). *The details of this example are reconsidered in Section 5.4.*

Figure 2.13 File menu, Petri net editor.

Figure 2.14 Tools menu, Petri net editor.

2.4. Translation of Annotated Petri Nets into C++ Programs

The translation of Petri nets model into the OSU Framework based C++ programs is described below [Keh 91]. The basic translation rules include the following:

Places

- Each user interface place *p* in the Petri net is mapped into a derived class *d* of its corresponding user interface abstract class in the OSU Application Framework;
- The place type, resource ID, and place ID are used to generate the head of the derived class; and
- The variables given in the places are used to generate instance variables of the derived class.

Transitions

- Each transition *t* of *p* is mapped into a public member function *f* of *d*.

Arcs

- Each action message inscribed on the output arcs of t is mapped into a statement in f ; and
- The translation of input arcs generates code that ensures that the transition can be fired if and only if each of its input places contains at least one token.

Messages

Message formats and their mapped C++ statements are listed as follows.

Message format	C++ statement
• MMM	MMM; (Sender and receiver of MMM are the same object.) *rrr*->MMM; (Sender of MMM is different from the receiver; *rrr* is the variable name of the receiver.)
• RRR::MMM	RRR->MMM;
• VVV=MMM	VVV=MMM; (Sender and receiver of MMM are the same object.) VVV= *rrr*->MMM; (Sender of MMM is different from the receiver; *rrr* is the variable name of the receiver.)
• VVV=RRR::MMM	VVV=RRR->MMM;
• **func**::FFF	FFF;
• RRR=**func**::FFF	RRR=FFF;

3. Use of the Annotated Petri Net Editor

To provide the reader with an overview of the PN editor system, a dataflow diagram of the system is provided in Figure 3.0. Input to the editor includes:

- Specifications from the programmer: The Petri net objects and message flows are specified by the programmer. The programmer enters required information for an object (e.g., the resource ID of a Macintosh user interface object).

- Reusable messages selected from the Browser: The Browser parses C++ source programs, builds the class hierarchy tree chart, displays the methods (i.e., messages) for the classes, copies the methods selected by the user, and sends the list of messages to the PN editor.

- Petri net external representation: Petri net external representation is saved by the editor as a textual form file according to internal Petri net representation. The programmer can open existing files as editor input.

Thus, the editor interacts with the programmer, processing the input and converting it to an internal Petri net model. The editor output includes:

- Internal Petri net model (i.e., internal Petri net representation): This model becomes the input for the Code Generator, which in turn generates C++ source code as output.

- External Petri net representation file: As noted above, the editor converts the internal representation file into an external representation file for subsequent use.

Output of the Code Generator (i.e., C++ source code) then becomes the input for the Macintosh Programming Workshop (MPW), which in turn compiles the source code, links the source code to the OSU Framework and to other C++ source program files, loads correspondent resources from the resource file, and generates the desired application.

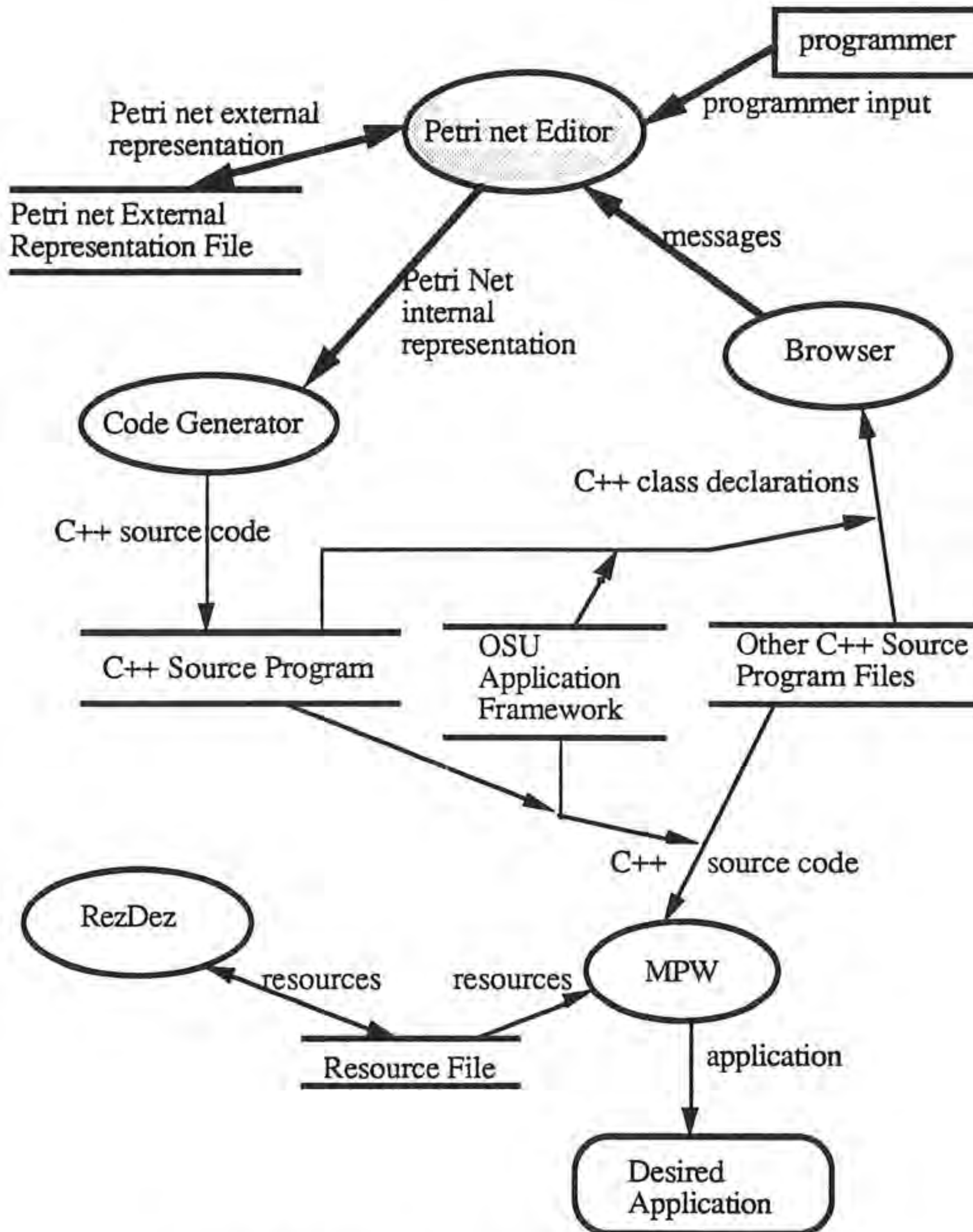


Figure 3.0 Data flow diagram for the Petri Net Editor.

3.1. File Actions

A Petri net graph edited within a window is internally represented as a Petri net model. This internal representation can be saved in the form of a text file for

subsequent reediting. Thus, creating a Petri net graph also creates an internal representation and an external file.

To create a new Petri net graph, the user may select "New" from the "File" menu (Figure 3.1). A new PN editor window, "Untitled," with an initial transition is created on the screen (Figure 3.2). Note that the initial transition stands for the starting point of an application, is unique, and cannot be deleted. A palette tool is also shown beside the window. This tool is the users' medium for informing the system which places or arcs should be created. As shown in Figure 3.3, there are seven different icons for drawing Macintosh user interface objects and two icons for drawing arcs. The user can draw the Petri nets or modify existing Petri nets with the palette tool and menus. These processes are described in Sections 3.2 and 3.3.

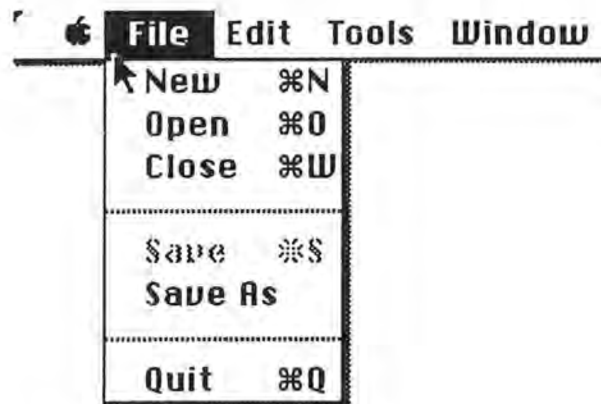


Figure 3.1 File menu items.

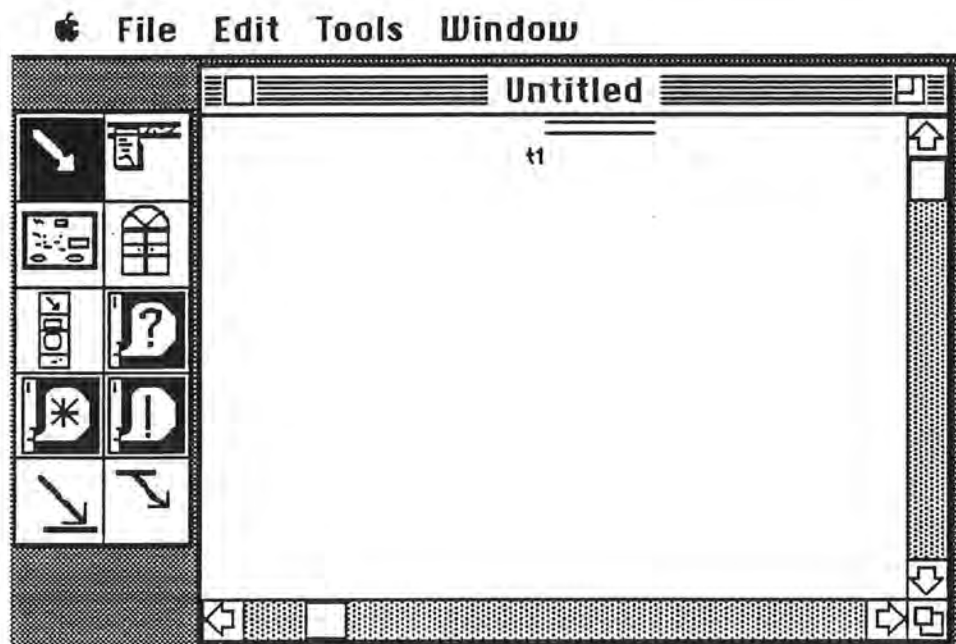


Figure 3.2 New Petri net window.

Internal representations of the Petri net model can be saved in textual form, creating files which can be read by opening them in the MPW environment. To see and edit the graphical representation of a Petri net, the user selects "Open" from the "File" menu (Figure 3.1). A Petri net file can be opened by choosing available files from an SFGetFile dialog box. The editor reads in the text file, builds the Petri net internal representation, loads a window, and display the Petri net graph on the screen.

Just as for any other Macintosh application, to save a file the user may select "Save" or "Save As" from the "File" menu, and to close a file the user may select "Close" from the "File" menu.

3.2. Creating a Petri Net

Initiating a net begins with the creation of places, transitions, and arcs, first selecting the appropriate icon with the mouse (Figure 3.3).

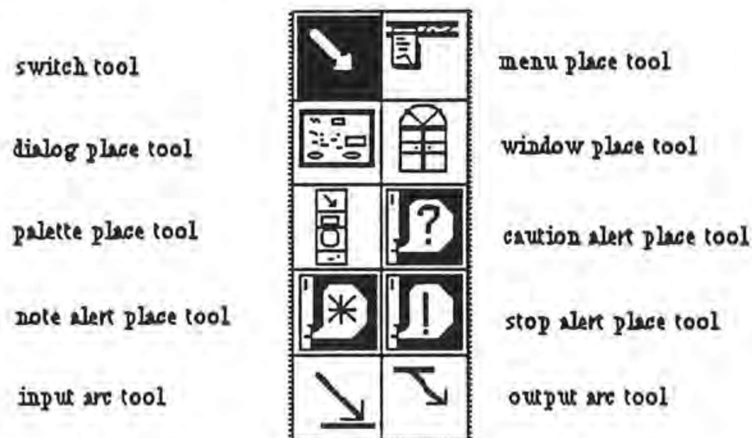


Figure 3.3 Palette tool.

To create a place, from the palette the user selects from among the seven place icons with the mouse, clicking it with the cursor to the desired location. A dialog box requesting further information on the created place is displayed (Figure 3.4(a)). If the created place is a dialog place, two radio buttons are activated to specify the appropriate mode; for other places, the radio buttons are inactive since the modes are set at default values (Figure 3.4(b)).

Place Information

Place ID : p2

Resource Type : DIALOG

Resource ID : 128

◉ Modal

○ Modeless

Instance Variables :

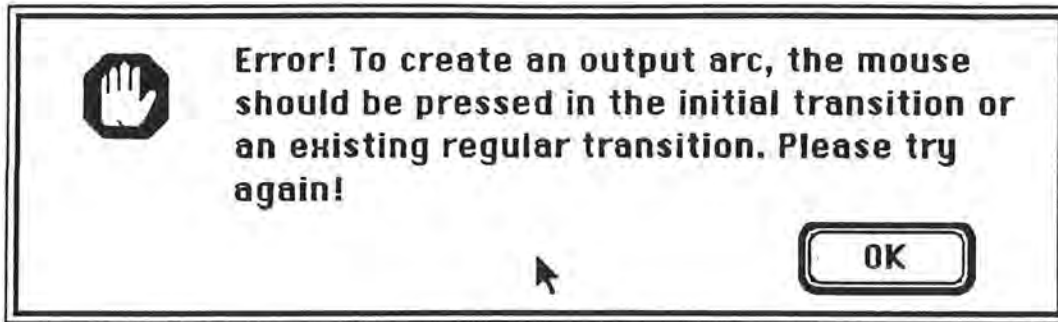
Cancel

OK

Figure 3.4(a) Dialog box for a dialog place (radio buttons active).

Figure 3.4(b) Dialog box for a menu place (radio buttons are inactive).

To create an output arc (i.e., connect a transition to a place), the user selects an output arc icon from the palette, clicks inside the transition and then drags the arc to a place. Note that if the user clicks the wrong end nodes or clicks outside of a node, then an output arc cannot be drawn. In this case, an alert will pop up (see Figure 3.5) to demonstrate the means to create an output arc.



(a)

(b)

Figure 3.5 Alert box for syntax errors and process for creating output arcs.

As noted in Section 2.3, messages, predicates, and sequence numbers can be inscribed on transition output arcs. Thus information can be entered by double-clicking on an output arc to obtain a dialog box, which provides an editText box for the acceptance of a sequence number (Figure 3.6). The user may type in the number or accept a default value of "0". The message for an output arc can be typed in directly; otherwise, the user may call the Browser (Figure 3.7), copy methods from the class hierarchy (Figure 3.8) shown in a Browser window, highlight the output arc (Figure 3.9), and then paste the message to the output arc by selecting "Paste" from the "Edit" menu (Figure 3.10). For details on the operation of the Browser, refer to [Li 91].

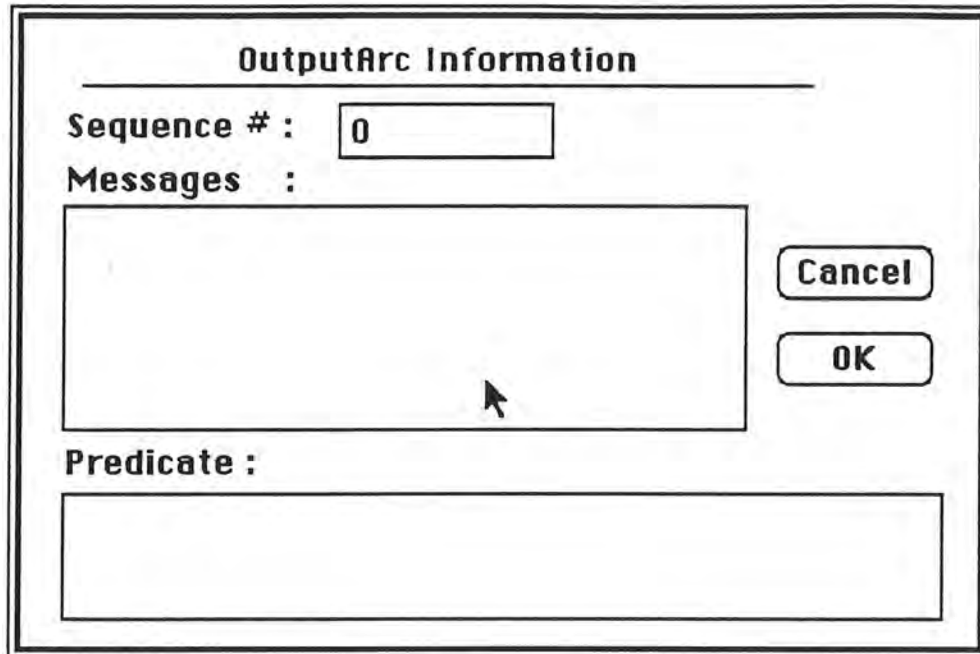


Figure 3.6 Dialog box for output arcs.

Figure 3.7. Selection of "Browse Hierarchy" to invoke the Browser.

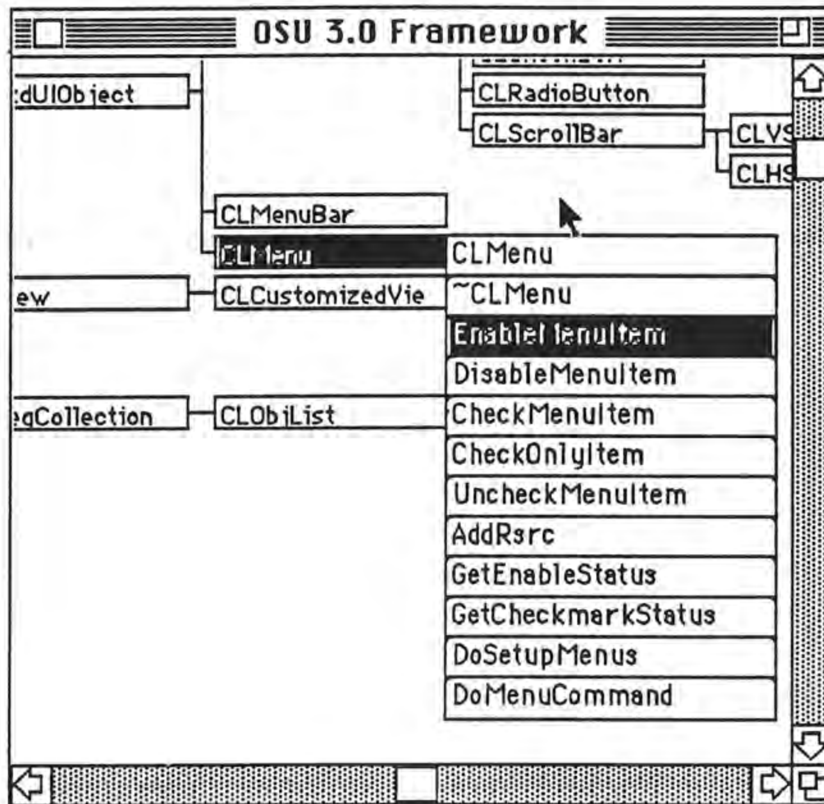


Figure 3.8 Copying methods from the Browser.

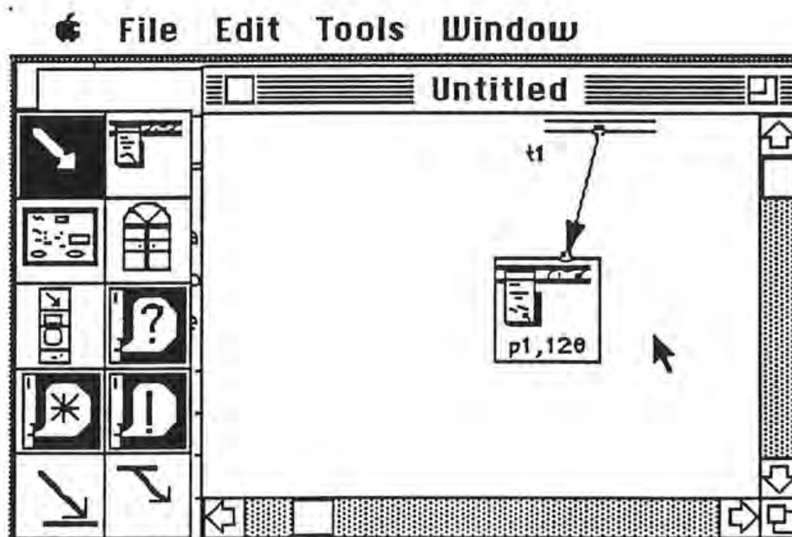


Figure 3.9 Output arc screen for message pasting.

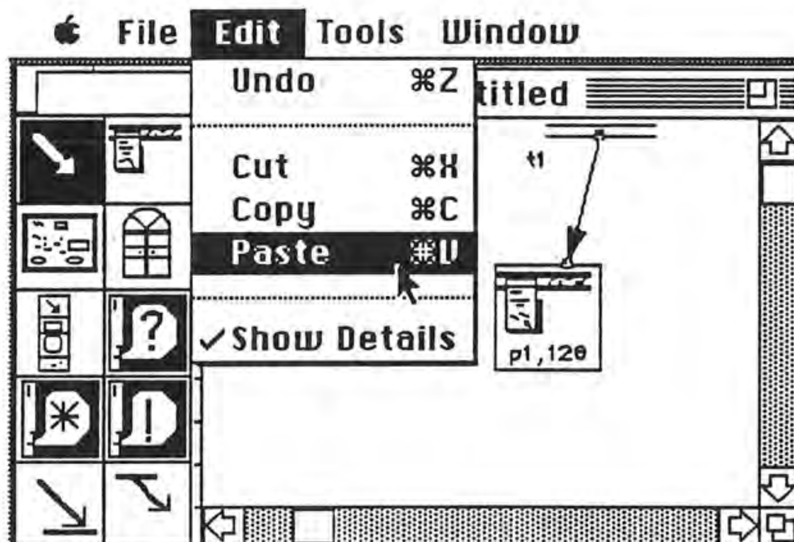


Figure 3.10 Pasting messages on highlighted output arcs.

To assure that messages are copied, the user can double-click the output arc. The current status of that output arc will be shown in a dialog box (Figure 3.11). Thus, the user may edit messages or type in additional messages.

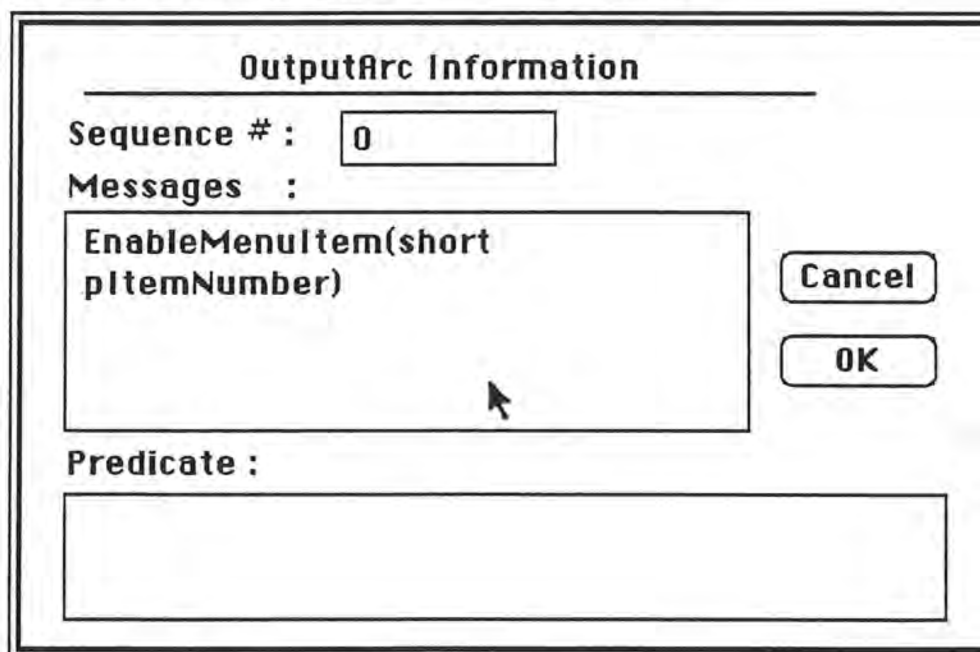


Figure 3.11 Output arc dialog box with messages.

To create an input arc (i.e., to connect a place to a transition), the user clicks the input arc icon in the palette, then clicks inside a place and drags the input arc. When the mouse button is released within an existing transition, the transition is connected to the place. If the mouse is not released in a transition, a new transition may be created (Figure 3.12). A dialog box appears, requesting appropriate transition information (Figure 3.13). In this box, the type of this transition (i.e., a regular or a quit transition) should be specified by clicking the appropriate radio button, then entering the ID of the place owning the transition. If the user does not specify this information, default values are provided by the system. The default value for a transition is a regular transition and the default value for a place ID is the number for the place connected to the created input arc.

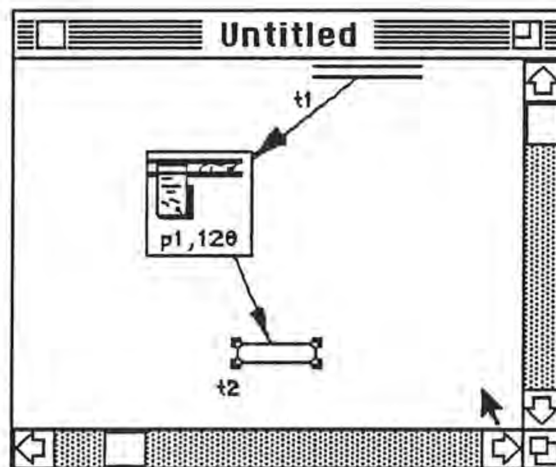


Figure 3.12 Creation of transition by drawing an input arc.

Transition Information

Transition ID : t2

◉ Regular Transition

○ Quit Transition

Belong to Place : 1

item # :

Cancel

OK

Figure 3.13 Dialog box with default transition information.

An input arc connects a place to a transition. The PN editor displays an alert if incorrect procedures are used to draws an input arc (Figure 3.14(a)). For example, if the mouse is not pressed within an icon, an alert box with a directions message appears. If the mouse is not released in a blank area or within an existing transition, a second alert (see Figure 3.14(b)) appears to indicate appropriate directions.

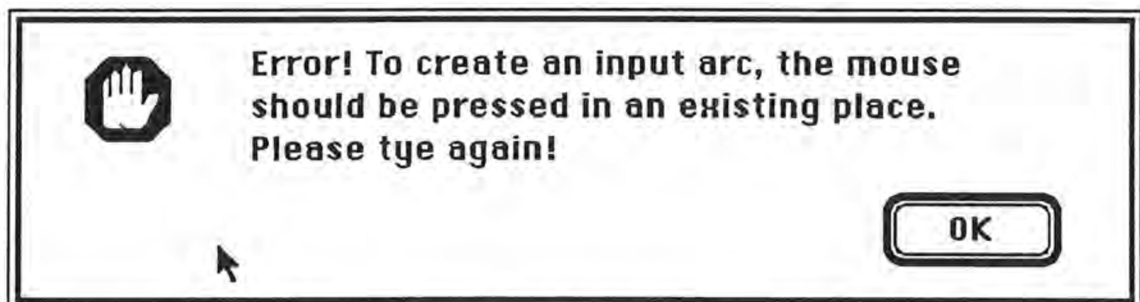


Figure 3.14(a) Alert for syntax error messages and directions for creating input arcs.

Figure 3.14(b) Alert for syntax error messages and directions for creating input arcs.

If a user does not want dialog boxes to appear each time a node or an arc is created, this feature can be disabled by removing the selection from "Show Details" in the "Edit" menu (Figure 3.15). Thus, nodes or arcs will be drawn directly without the appearance of informative dialog boxes. To pop up a box to enter details, the user must double-click either a place, a transition, or an output arc.

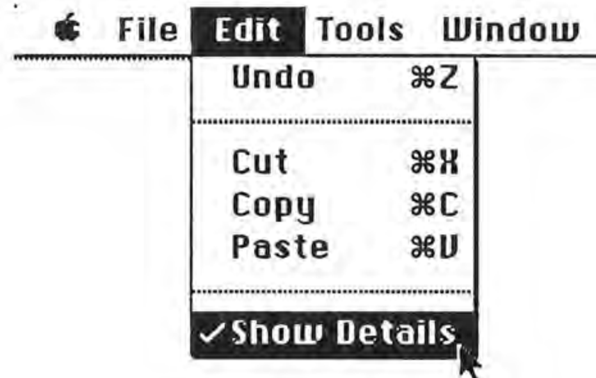


Figure 3.15 Edit menu.

3.3. Editing the Petri Net

For editing a Petri Net graph, places and transitions are treated as nodes, while input and output arcs are referred to as arcs. The following editing procedures may be used.

Drag

If a node is repositioned, then all of its arcs are automatically adjusted (Figure 3.16).

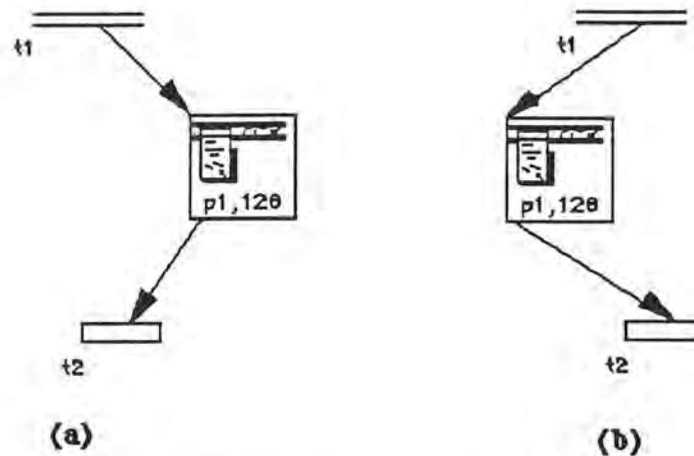


Figure 3.16 Repositioning nodes: (a) prior to drag and (b) following drag.

Cut

To cut an element, the user highlights or groups the shapes, then selects "Cut" from the "Edit" menu (Figure 3.15).

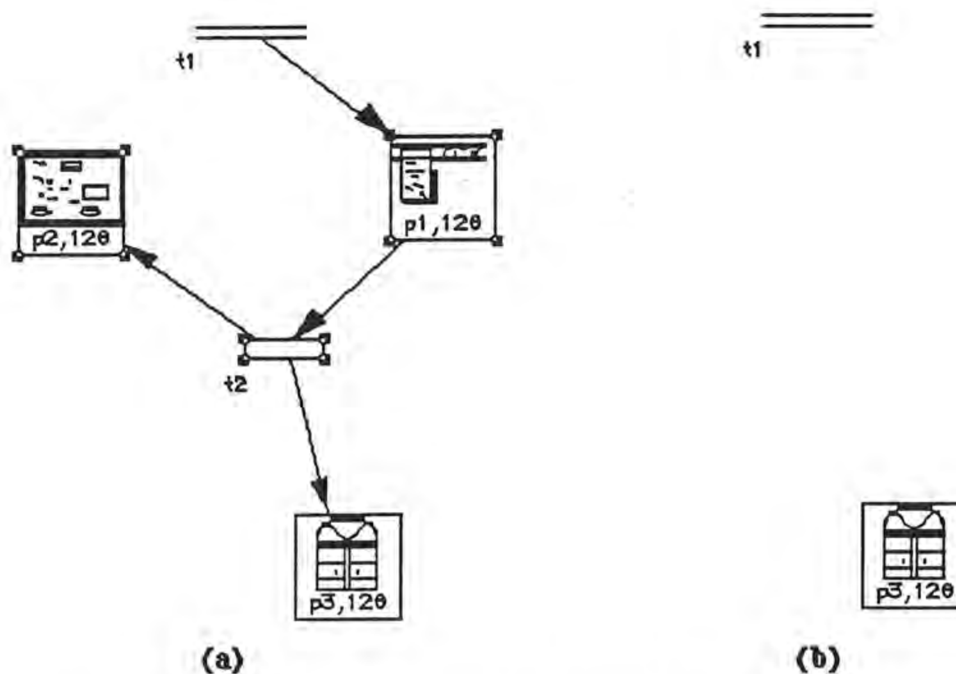


Figure 3.17 Cutting nodes: (a) prior to cut and (b) following cut.

Cutting a node

If a place or a transition is deleted, all of its arcs will be deleted. Only the place or the transition is placed on the clipboard. If a place is cut, its owning transitions are also cut. The process is identical to cutting a group of nodes (see below).

Cutting an arc

Cutting an arc deletes the arc without placing it on the clipboard.

Cutting a group of nodes

Everything contained within the group, including all internal arcs (i.e., arcs with both end nodes, representing a place and a transition, compose a group), is removed (Figure 3.17) and placed in the clipboard (Figure 3.18). Those arcs

with source or destination nodes are removed without being saved to the clipboard (Figure 3.18).

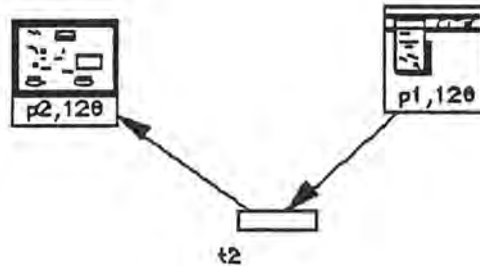


Figure 3.18 Removing features to the clipboard.

Copy

To copy a net and save them in the clipboard, the user highlights groups of shapes, then selects "Copy" from the "Edit" menu (Figure 3.15).

Copying a node

Only the node is duplicated and placed in the clipboard without connecting arcs.

Copying an arc

An arc cannot be placed in a clipboard, which is one of the limitations of the editor. If an arc is pasted, the system should support a means for the user to specify pasting actions for the two end nodes.

Copying a group of nodes

Everything contained within the group, including all of the internal arcs, is copied to the clipboard; arcs with only source or destination nodes are not copied.

Paste

To complete a paste procedure, the user selects "Paste" from the "Edit" menu. The system then copies the shapes in the clipboard and pastes them to the screen.

Pasting a node or a group of nodes

The node(s) is pasted beside the original nodes, as shown in Figure 3.19).

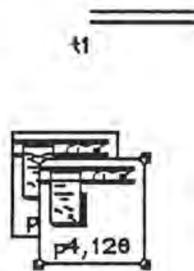


Figure 3.19 Pasting a node.

Pasting an arc

As noted above, the PN editor does not support the means to paste an arc.

3.4. Syntax Checks

Syntax checks are performed to assure that the Petri Net has a legal structure and observes syntax rules. Syntax restrictions may be either built-in restrictions or compulsory restrictions. Build-in restrictions are properties that are automatically warranted by the system since the user interface for the editor is unable to violate these restrictions. Built-in syntax restrictions include the following:

- A transition must be connected by at least one input arc.

- An arc can only be drawn from a place to a transition (or vice versa, see Figures 3.5, 3.14).
- A quit transition cannot have an output arc (see Figure 3.5(a)).
- An initial transition cannot have an input arc (see Figure 3.14(b)).
- Every net must have exactly one initial transition.

The first restriction cannot be violated since a transition cannot be created in the absence of an input arc. The editor does not allow an initial transition to be copied, cut, or pasted. Thus, the final restriction given above cannot be violated. For the remainder of the restrictions, the system provides error messages.

Compulsory syntax restrictions are properties that are not automatically guaranteed by the system, but which must be fulfilled in order to run the Code Generator. The system can also perform a syntax check for compulsory restrictions when "Check Syntax" in the "Tools" menu is invoked (Figure 3.20).

Figure 3.20 Invoke "Check Syntax".

The compulsory syntax restrictions include:

- Each place should be connected to at least one output arc (Figure 3.21).
- The initial transition should have at least one output arc (Figure 3.21).

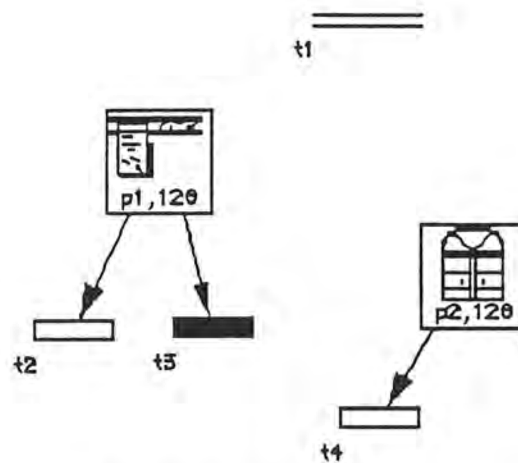


Figure 3.21 A net with syntax errors.

Violations of the compulsory syntax restrictions are reported in alert boxes (Figures 3.22-3.23).

Figure 3.22 Alert box for a syntax error.

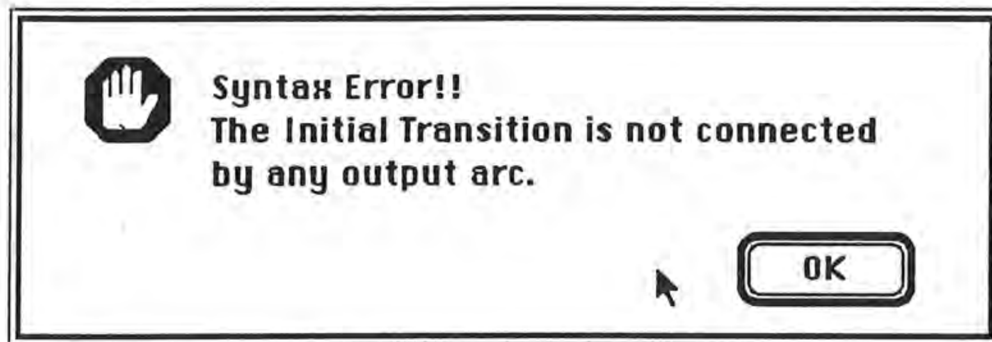


Figure 3.23 Alert box demonstrating syntax error.

3.5. Code Generation

To generate C++ source code, the user selects "Generate Code" from the "Tools" menu (Figure 3.20). The syntax is checked prior to the generation of code. If errors are found, an error messages is first provided (Figures 3.22-3.23), then no further source code is generated (Figure 3.24).

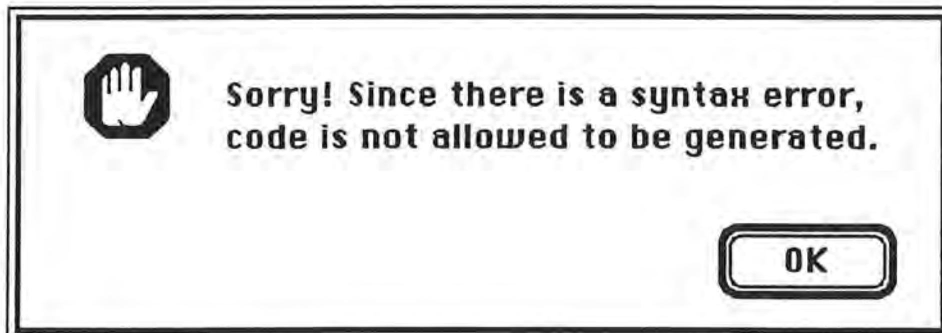


Figure 3.24 Stop alert notice for code generation.

4. Design and Implementation of the Petri Net Editor

The Petri Net Editor was constructed by subclassing and instantiating classes of the OSU Framework. From Figure 4.1, derived classes, indicated by numerical designations, are classified in four sets:

- GUI objects (items 1-13);
- Petri net Storage structures derived from the CLModel class and the data structure library, and Petri net storage objects derived from the CLObject class (item 15, 16, and 17);
- Graphics Objects derived from the shape library (item 18); and
- Petri net domain specific view (item 14).

These classes are discussed in the following four sections. Appending * to the end of a method name indicates that that method overrides an OSU Framework method.

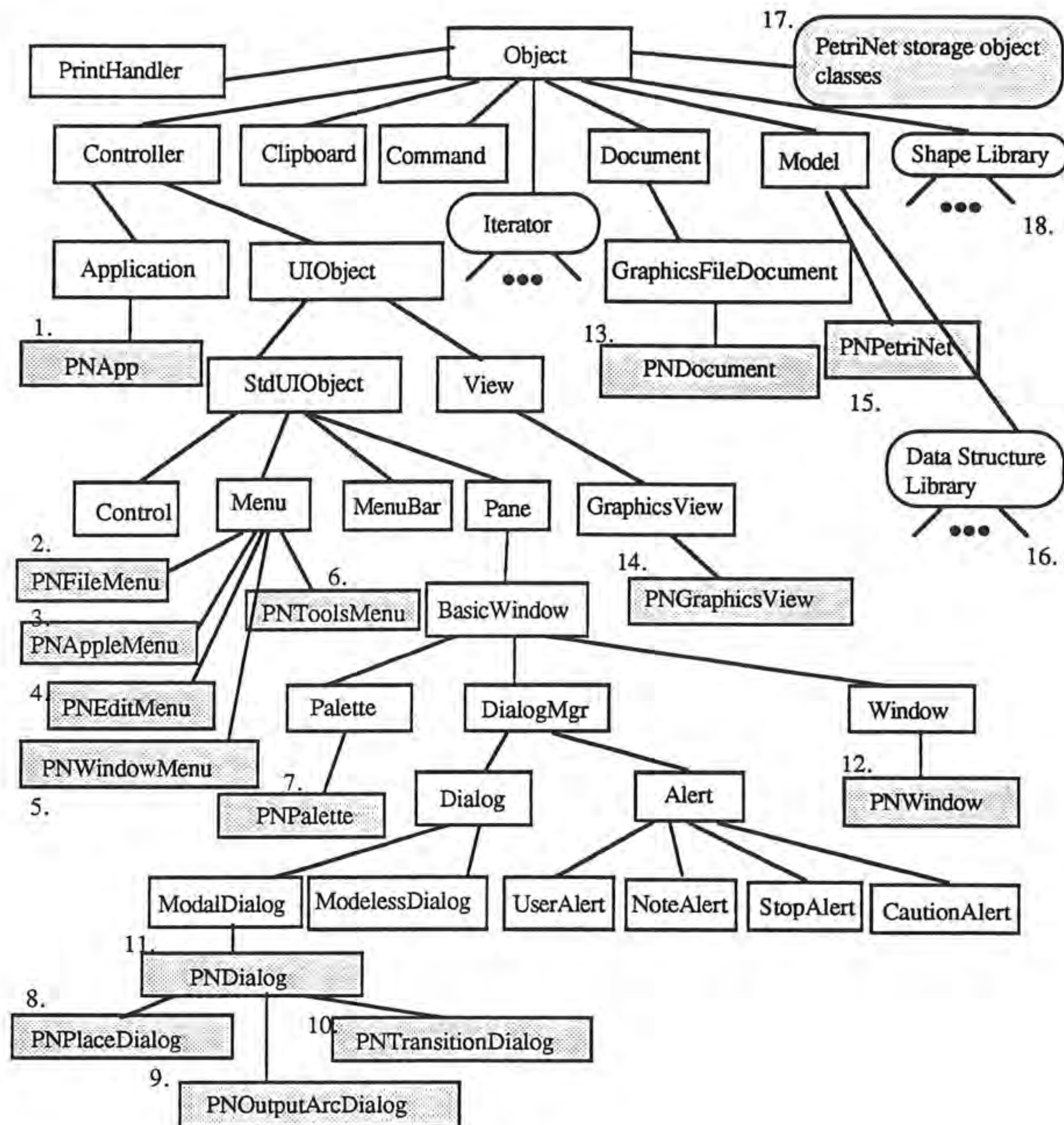


Figure 4.1 PN editor classes derived from the OSU Application Framework.

4.1. User Interface Classes

The user interface classes are standard and can be implemented through the OSU Framework. For the attribution of specific purposes, objects derived from these classes must be modified to override certain of the standard methods provided.

PNApp

The PNApp class implements the main event loop, directing events at the objects to which they are responsible for operations. The PNApp class has a member, fBrowser, that serves as a pointer toward object from the Browser class, which is responsible for initialization of the Browser, obtaining lists of selected methods, closing the Browser window, and removing selected methods from the lists [Li 91]. "CreateMenus*" and "Initialize*" are overridden. The first is used to create menu bar objects and objects which are specific to the PN editor; the second, which is an abstract method from the CLApplication class, is overridden to perform additional application initializations. These tasks include the creation and initialization of palette objects and the fBrowser member.

PNDialog

The PNDialog class is responsible for the displaying of all dialog requiring a user response before proceeding with further tasks. This class exists as a subclass to provide two additional methods, "HandleRadioGroup" and "DrawDLine," that are not

provided in the Framework. The first is used to manipulate the radio group and the second is responsible for line drawing within dialog boxes.

Other Dialog Classes

PNPlaceDialog is responsible for dialog which requests place information; PNTransitionDialog is responsible for dialog which requests transition information; and PNOutputArcDialog is responsible for dialog which requests output arc information. For each of these dialog classes, "DoMouseDown*" is the only method which is overridden.

PNDocument

This class is used to override "CreateModel*," "DoRead*," "DoWrite*," and "DoSetUpMenu*". The first is used to create an instance of the PNPetriNet class as an application model, the second and third are used to read/write the attributes of Petri net data elements and their corresponding shapes from/to files, and the fourth is overridden to set up application menu status.

Menu Classes

The classes PNAppleMenu, PNFileMenu, PNEditMenu, PNToolsMenu, and PNWindowMenu are responsible for menu commands. Each of them overrides the method "DoMenuCommand*" to perform the tasks in accordance with a menu item

selected by the user. To explain these tasks, the command behaviors, in the form of code or pseudo code, is presented below :

```
CLCommand *PNAppleMenu::DoMenuCommand(short pItemNumber){//source code

    Str255 name;
    short  temp;
    CLUserAlert    *aboutPetriNet;

    if (pItemNumber == 1) {
        aboutPetriNet = new CLUserAlert(145);
        aboutPetriNet->Draw();
        delete aboutPetriNet; }
    else {
        GetItem(fMenuHandle,pItemNumber,name);
        temp = OpenDeskAcc(name);}// perform desk accessory
    return 0;
};


CLCommand *PNFileMenu::DoMenuCommand(short pItemNumber){//source code
    PNWindow *aWindow;
    if ((pItemNumber==NEW)||(pItemNumber==OPEN))
        aWindow=new PNWindow();
    else
        aWindow=(PNWindow *) gApplication->GetWindowByName("PNWindow");
    switch(pItemNumber) {
        case NEW:
            aWindow->Initialize();
            aWindow->Draw();
            aWindow->DoNew();
            break;

        case OPEN:
            aWindow->Initialize();
            if (aWindow->DoOpen()) // if a window is opened successfully
                aWindow->Draw(); //draw the window
            else
                delete aWindow;
            break;
        case CLOSE:
            if (aWindow) {
                Boolean success=aWindow->DoClose();
            }//if (aWindow)
            break;
        case SAVE:
            if (aWindow) aWindow->DoSave();
            break;
        case SAVE_AS:
            if (aWindow)
                if(aWindow->DoSaveAs()) {}
            break;
        case QUIT: //quit the application
            Boolean jump=false;
            Boolean success;
            while ((!jump) && (aWindow=(PNWindow *) gApplication-
>GetWindowByName("PNWindow"))) { // close the windows on the screen
                Boolean success=aWindow->DoClose();
```

```
                if (!success) //if any window is not closed
                    jump=true; //the quit action is cancelled
            }
            if (!jump) gApplication->Terminate(); //quit the application
            break;
    } //switch
    return 0;
};


CLCommand *PNEditMenu::DoMenuCommand(short pItemNumber){ //pseudo code
    theView=the view in the front most window;
    switch(pItemNumber) {
        case UNDO:
            check to see it's an undo or redo mode;
            if (undo) call theView->Undo else theView->Redo;
            break;
        case CUT:
            call theView->PNCut();
            break;
        case COPY:
            check to see which window is the front most window;
            if (thefrontmostWindow==PNWindow) // it's Petri Net window
                call theView->PNCopy();
            else //it's a browser window and the user is copying the methods
                call gApplication->fBrowser->addMethodsToList();
            break;
        case PASTE:
            call theView->PNDoPaste();
            break;
        case SHOW_DETAIL:
            checkmarks or uncheckmards "Show Detail" in the "Edit" menu;
            break;
    }
};


CLCommand *PNToolsMenu::DoMenuCommand(short pItemNumber){//pseudo code

    switch(pItemNumber) {
        case CODE_GEN:
            if (CheckSyntax()) {//check syntax
                // no syntax error
                thePetriNet=get the pointer of the Petri Net model;
                    thePetriNet->Translation();
                set the cursor to watch cursor;
            } else { //there is a syntax error, send error messages
                bring an alert for error messages
            }
            break;
        case CHECK_SYNTAX:
            if(CheckSyntax()) ; //check syntax
            break;
        case BROWSE_HIERACHY:
            ((PNApp *)gApplication)->fBrowser->newBrowser();
            break;
    }
    return 0;
};


CLCommand *PNWindowMenu::DoMenuCommand(short pItemNumber){//source code
    PNWindow *aWindow;
```

```
    BRWindow *brWindow;
    switch(pItemNumber) {
        case PNWIND://Petri net window
            aWindow=(PNWindow*) gApplication->GetWindowByName("PNWindow");
            if (aWindow) aWindow->Draw();//bring the window to front
            break;
        case BRWIND://Browser window
            brWindow=(BRWindow*) gApplication->GetWindowByName("BRWindow");
            if (brWindow) brWindow->Draw();
            break;
    }//switch
    return 0;
};
```

PNPalette

The method "DoMouseCommand*" is overridden to highlight icons from the palette, as selected by the user, notifying the view, which is within the front-most window, of the current shape tool.

PNWindow Class

The PNWindow class implements standard window manipulation tasks, including resizing and zooming. "DoNew*" is overridden for performing those tasks required each time a window is created, including calling the initialization method of the view or setting up menu status (e.g., enabling "Close" or "Save As" in the "File" menu). "DoOpen*", "DoClose*", and "DoSaveAs*" are overridden for purposes of refinement. "DoOpen*" and "DoSaveAs*" call parental class versions of "DoOpen" and "DoSaveAs" and set up their own menu status. "DoClose*" calls a parent class version of "DoClose" and checks to determine if the current window is the final Petri Net window on the screen. If the response is positive, the palette is hidden since there can be no window on the screen after "DoClose" is performed. Code for "DoClose*" is provided as follows:

```
Boolean PNWindow::DoClose(void) {
```

```
    Boolean success=CLWindow::DoClose(); // call parental
    if(success) {//if the window is closed successfully
        if(gApplication->GetWindowCountByName("PNWindow")) {}
        else {// window list is empty now
             // set up menu status
            gApplication->fMenuBar->DisableMenuItem(FILE_ID,SAVE);
            gApplication->fMenuBar->DisableMenuItem(FILE_ID,SAVE_AS);
            gApplication->fMenuBar->DisableMenuItem(FILE_ID,CLOSE);
            gApplication->fMenuBar->DisableMenuItem(TOOL_ID,1);
            gApplication->fMenuBar->DisableMenuItem(TOOL_ID,2);
            gApplication->fMenuBar->DisableMenuItem(EDIT_ID,0);
            gApplication->fMenuBar->DisableMenuItem(WIND_ID,PNWIND);
            CLBasicWindow *aPalt;
            aPalt=gApplication->GetWindowByName("PNPALETTE");
            if (aPalt) ShowHide(aPalt->fWindPtr,false);//hide the palette
        }
    }
    return success;
}
```

4.2. Petri Net Storage Structure and Object Classes

PNPetriNet

PNPetriNet is a subclass of CLModel used for the storage of the Petri Net structure, derived from CLModel since this structure is not a standard data structure as defined in the data structure library. PNPetriNet uses two lists to store pointers for places and transitions. As shown in Figure 4.2, there are lists of pointers to places and transitions: **placeList** and **transitionList** are instances, respectively, of PNPlaceList and PNTransitionList, each of which are subclasses of CLObjList. A number of CLObjList methods can be reused to add or remove objects from lists or to find objects; moreover, iterators can be declared for navigation or manipulation of elements within lists.

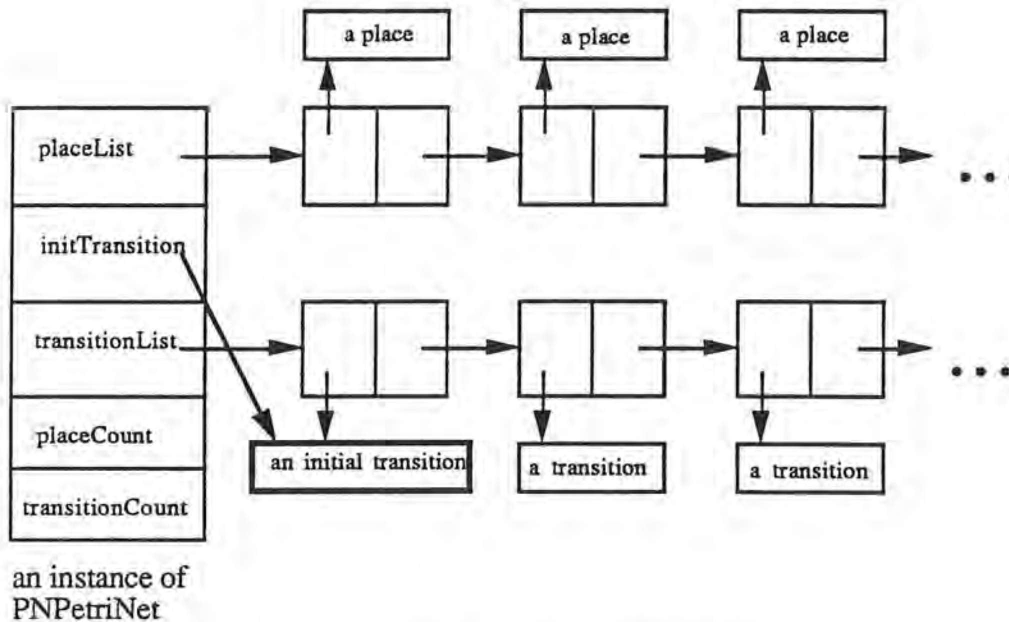


Figure 4.2 Instance of PNPetriNet.

To allow ease of simulation, PNPetriNet maintains a pointer to the initial transition. The simulator can obtain that pointer directly to initiate simulations.

For each place/transition there is a unique ID, placeID/ transitionID. Instance variables, placeCount and transitionCount (Figure 4.2), are used to record the IDs for the most recently created places and transitions. When a new place and/or transition is created, the value of, respectively, placeCount or transitionCount is increased, as is the ID of the new place/transition. Variables and functions included the following:

1) Instance variables:

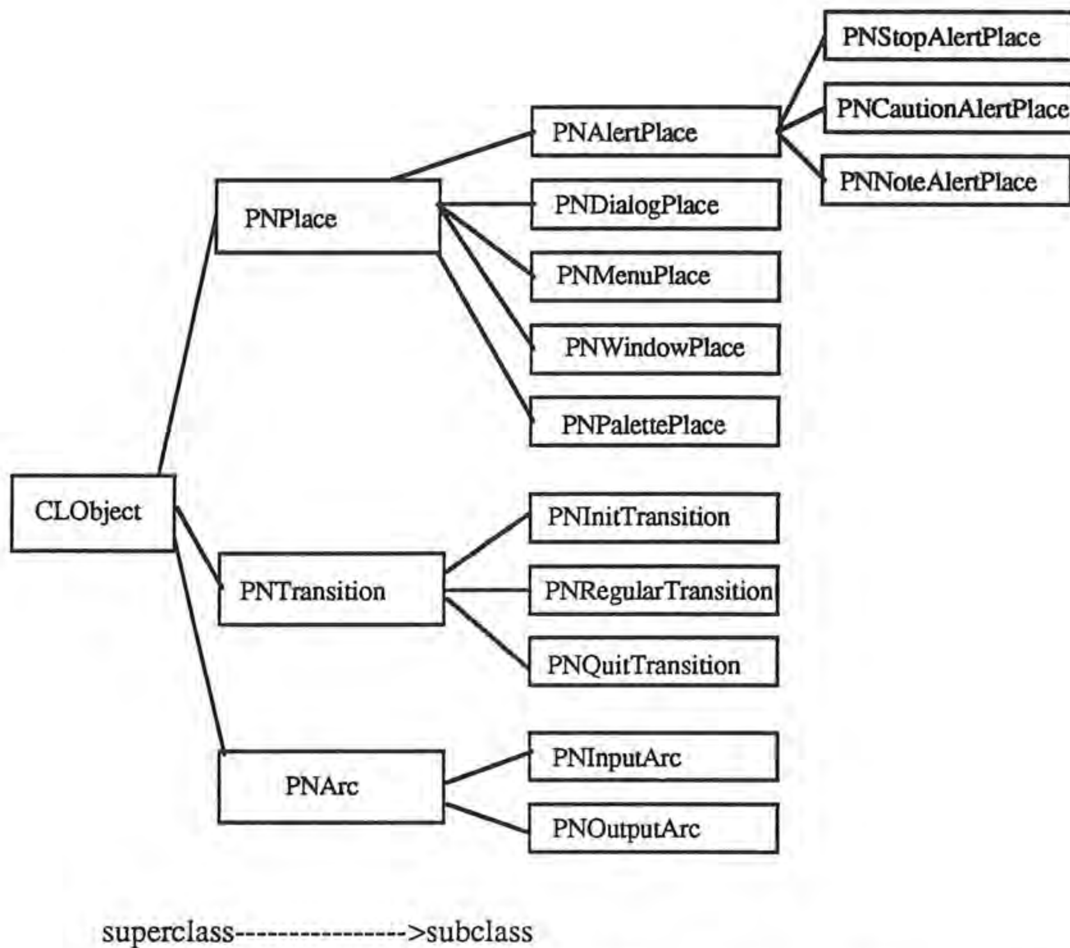
• placeList	list of pointers to places
• transitionList	list of pointers to transitions
• initTransition	pointer to the initial transition
• placeCount	ID of latest created place
• transitionCount	ID of latest created transition

2) Member functions:

• CreateATransition	create a new transition and add the new transition to **transitionList**
• CreateAPlace	create a new place and add the new place to **placeList**
• CheckSyntax	check syntax
• AddAPlace	add a place to **placeList**
• AddATransition	add a transition to **transitionList**
• RemoveAPlace	remove a place from **placeList**
• RemoveATransition	remove a transition from **transitionList**
• DoWrite	write the attributes of elements in the Petri net to a file
• DoRead	read the attributes of elements from a file
• GetPlaceCount	return **placeCount**
• GetTransitionCount	return **transitionCount**
• GetPlaceByID	get the pointer of a place by ID
• GetTransitionByID	get the pointer of a transition by ID

Hierarchy of Petri Net Storage Object Classes

The class hierarchy of the Petri Net storage object classes is constructed as shown in Figure 4.3.



superclass--------------->subclass

Figure 4.3 Class hierarchy of Petri Net storage object classes.

In this hierarchy, there are three major branches, PNPlace, PNTransition, and PNArc, each of which is derived from the CLObject class. The design of these classes is based upon the principle of code sharing, to the greatest degree possible without compromising logical relationships among classes. This class structure than might otherwise have been required for the PN editor for reason of considerations encompassed in the Simulator and the Code Generator. These classes are summarized below.

PNPlace

PNPlace is an abstract class responsible for maintaining the attributes and common behaviors of place objects. Variables and functions for this class are as follows:

1) Instance Variables:

• placeID	unique ID of a place
• instanceVariables	pointer to a list of instance variables
• isModal	boolean value indicating if the place is modal
• mark4CheckSyntax	boolean value indicating if the place is connected by an output arc
• resourceID	resource ID for a place
• inputArcs	pointer to a list of input arcs

2) Member Functions:

• CLIs_a*	return a defined integer to indicate the type of object
• GetPlaceID	return **placeID**
• SetPlaceID	set the value of **placeID**
• DoWriteSelf/DoReadSelf	write/read attributes to/from a file
• GetResourceID	return **resourceID**
• SetResourceID	set the value of **resourceID**
• RemoveInputArc	remove an input arc from **inputArcs**
• GetInputArcList	return **inputArcs**
• SetInputArcList	set **inputArcs**
• GetInstanceVariableList	return the value of **instanceVariables**
• DupDetail*	duplicate the details of an object

Other Place Classes

The following place classes are responsible for object behaviors as follows: 1) PNMenuPlace for the behavior of a menu place; 2) PNWindowPlace for the behavior of a window place; 3) PNPalette for the behavior of a palette place; 4) PNDialogPlace for the behavior of a dialog place; 5) PNAlertPlace for the behavior of an alert place; 6) PNStopAlertPlace for the behavior of a stop alert place; 7) PNCautionAlertPlace for the behavior of a caution alert place; and PNNoteAlertPlace for the behavior of a note alert place. Each has the following member functions (i.e., there are no instance variables for these classes).

•CLIs_a*	return a defined integer to indicate the type of object
• Duplicate*	duplicate the object

PNTransition

PNTransition is an abstract class which is responsible for maintaining common information among all transition objects. This class has the following variables and functions:

1) Instance Variables:

• transitionID	unique ID for a transition

2) Member Functions:

• CLIs_a*	return a defined integer to indicate the type of object
• GetTransitionID	return **transitionID**
• SetTransitionID	set the value of **transitionID**

• DoWriteSelf/DoReadSelf	write/read attributes to/from a file
• AddOutputArc	virtual method
• RemoveOutputArc	virtual method
• AddInputPlaceArc	virtual method
• RemoveInputPlaceArc	virtual method
• GetBelongToPlace	virtual method
• SetBelongToPlace	virtual method
• GetOutputArcList	virtual method
• DupDetail*	duplicate the details of a transition

PNInitTransition

PNInitTransition is responsible for storing and maintaining information regarding initial transitions. This class has the following variables and functions:

1) Instance Variables:

• outputArcs	list of pointers to output arcs

2) Member Functions:

•CLIs_a*	return a defined integer to indicate the type of object
• AddOutputArc	add an output arc to **outputArcs**
• RemoveOutputArc	remove an output arc from **outputArcs**
• DoWriteSelf/DoReadSelf	write/read its attributes to/from a file
• CheckSyntax	check to see if the initial transition has an output arc
• MarkOutputPlaces	ask **outputArcs** to mark their output places

- Duplicate* duplicate the object
- DupDetail* duplicate the details of the object (a method is called by "Duplicate")

PNRegularTransition

PNRegularTransition is responsible for storing and maintaining information for regular transitions. As shown in Figure 4.4, a regular transition has a list of pointers to output arcs and maintains a pointer to the place which owns this transition. **inputPlaceArcs** is a list of pointers to input place arcs, which in turn store pointers to the input places for this transition; **inputPlaceArcs** is used by the Code Generator to generate boolean statements to check for the existence of certain objects; and **itemNumber** is the number of the item represented by this transition within a GUI object. Variables and functions for this class are described below:

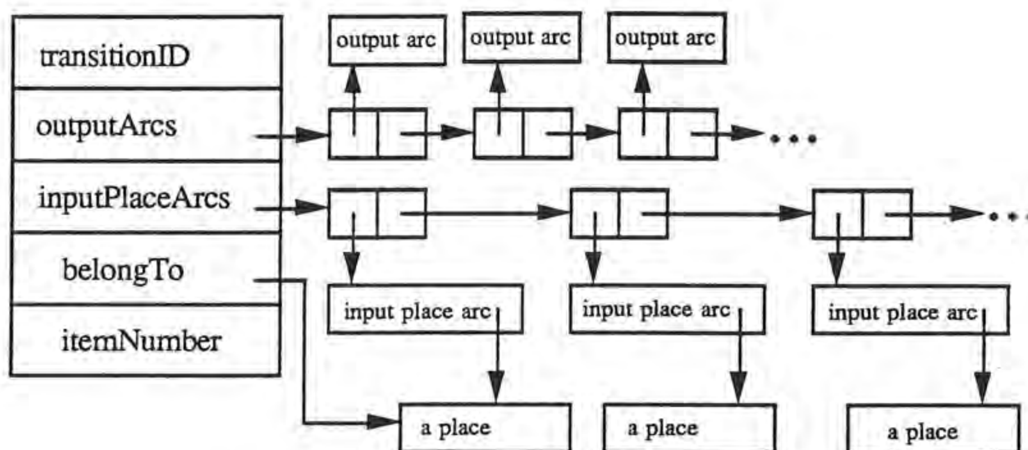


Figure 4.4 Instance of PNRegularTransition.

1) Instance Variables:

• outputArcs	list of pointers to output arcs
• inputPlaceArcs	list of pointers to input place arcs
• belongTo	pointer to a place which owns this transition
• itemNumber	number of the item represented by this transition

2) Member Functions:

• CLIs_a*	return a defined integer to indicate the type of object
• AddOutputArc	add an output arc to **outputArcs**
• RemoveOutputArc	remove an output arc from **outputArcs**
• AddInputPlaceArc	add an input place arc to **inputPlaceArcs**
• RemoveInputPlaceArc	remove an input place arc from **inputPlaceArcs**
• DoWriteSelf/DoReadSelf	write/read its attributes to/from a file
• CheckSyntax	check to determine if the initial transition has an output arc
• MarkOutputPlaces	ask **outputArcs** to mark output places (used during the process of checking syntax)
• GetBelongTo	return **belongTo**
• SetBelongTo	set the value of **belongTo**
• GetItemNumber	return **itemNumber**
• GetOutputArcList	return **outputArcs**
• Duplicate*	duplicate the object
• DupDetail*	duplicate the details of the object (called by "Duplicate")

PNQuitTransition

PNQuitTransition is responsible for storing and maintaining information regarding quit transitions. This class has the following variables and functions:

1) Instance Variables:

• inputPlaceArcs	list of pointers to input place arcs
• belongTo	pointer to a place which owns this transition
• itemNumber	number of the item represented by this transition

2) **Member Functions:**

•CLIs_a*	return a defined integer to indicate the type of object
• AddInputPlaceArc	add an input place arc to **inputPlaceArcs**
• RemoveInputPlaceArc	remove an input place arc from **inputPlaceArcs**
• DoWriteSelf/DoReadSelf	write/read attributes to/from a file
• GetBelongTo	return **belongTo**
• SetBelongTo	set the value of **belongTo**
• GetItemNumber	return **itemNumber**
• Duplicate*	duplicate an object
• DupDetail*	duplicate the details of an object (called by "Duplicate")

PNArc

PNArc is a subclass of CLObject and is an abstract class which maintains common information of arc objects. Variables and functions are described as follows:

1) Instance Variables:

• transition	pointer to a transition connected by this arc
• place	pointer to a place connected by this arc

2) Member Functions:

• CLIs_a*	return a defined integer to indicate the type of object
• GetPlace	return **place**
• GetTransition	return **transition**
• SetPlace	set the value of **place**
• SetTransition	set the value of **transition**
• DoWriteSelf/DoReadSelf	write/read attributes to/from a file
• BuildConnection	virtual method
• ReleaseConnection	virtual method
• DupDetail*	duplicate the details of an object (called by "Duplicate")

PNInputArc

PNInputArc is responsible for maintaining information for an input arc. There are no variables for this class and member functions are described as follows:

• CLIs_a*	return a defined integer to indicate the type of object
• DoWriteSelf/DoReadSelf	write/read attributes to/from a file

• BuildConnection	ask **place** to add an input arc to its input arc list and ask **transition** to create an input place arc
• ReleaseConnection	ask **place** to remove an input arc from its input arc list and ask **transition** to remove an input place arc which maintains a pointer to **place**
• Duplicate*	duplicate this object
• DupDetail*	duplicate the details of an object (called by "Duplicate")

PNOutputArc

PNOutputArc is responsible for maintaining information and behaviors for output arcs. This class has the following variables and functions:

1) Instance Variables:

• predicate	the predicate inscribed on this output arc
• messages	a list of messages inscribed on this output arc
• sequenceNumber	the sequence number inscribed on this output arc

2) Member Functions:

• CLIs_a*	return a defined integer to indicate the type of object
• DoWriteSelf/DoReadSelf	write/read attributes to/from a file
• BuildConnection	ask **transition** to add this output arc to its output arc list
• ReleaseConnection	ask **transition** to remove this output arc from its output arc list

• MarkOutputPlace	mark the place which is connected by this output arc (called when syntax is checked)
• GetMessageList	return **messages**
• SetMessageFromBR	accept messages passed from the Browser for storage
• Duplicate*	duplicate an object
• DupDetail*	duplicate the details of an object (called by "Duplicate")

4.3. Graphics Object Classes

The graphics object (i.e., shape) classes are implemented to maintain graphic representations of Petri net storage objects and are derived from the OSU Framework shape library (Figure 4.5). Behaviors are inherited from parent classes. Each object maintains a pointer to its corresponding Petri Net storage object. The variables and functions of each class are described in each of the following subsections.

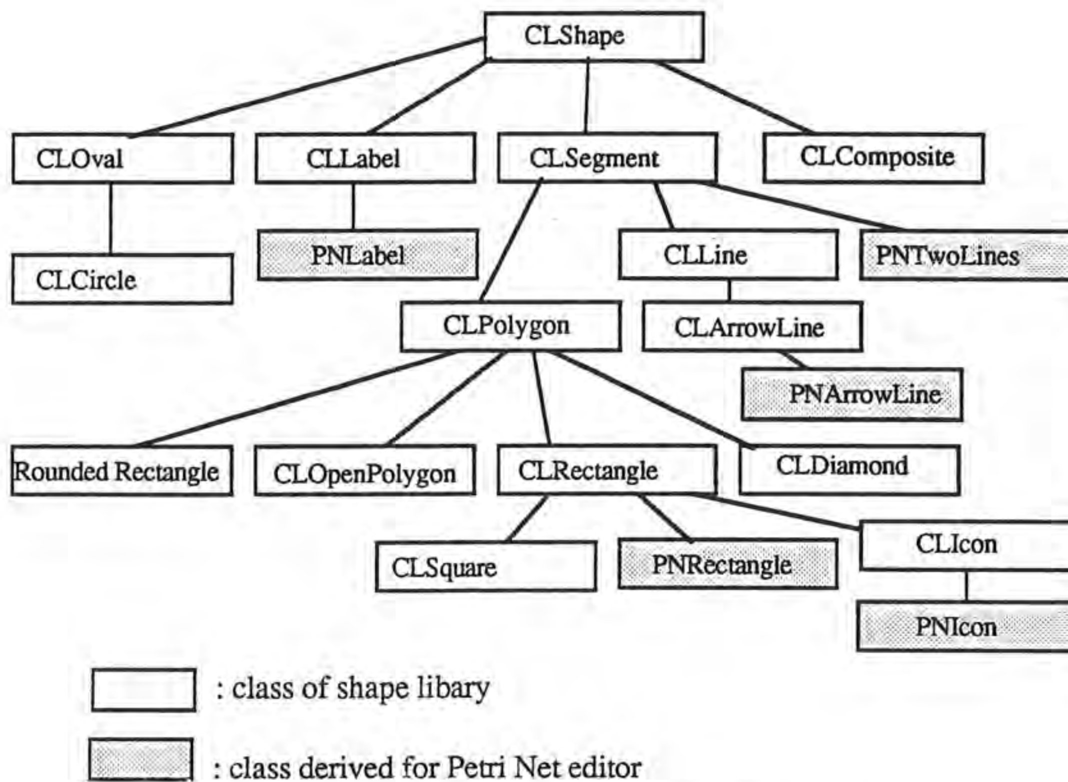


Figure 4.5 Class hierarchy of graphics object classes.

PNIcon

PNIcon is a subclass of CLIcon, used for the representation of a graphics view of a place.

1) Instance Variable:

• dataPointer	pointer to a corresponding place

2) Member Functions:

• CLDrawFrame*	draw an icon with place and a resource ID inscribed

• CLIs_a*	return a defined integer to indicate the type of object
• CLReadSelf/CLWriteSelf*	read/write all attributes from/to a file
• SetDataPointer	set the value of **dataPointer**
• GetDataPointer	return the value of **dataPointer**
• Duplicate*	duplicate an object
• DupDetail*	duplicate the details of an object (called by "Duplicate")

PNArrowLine

PNArrowLine is a subclass of CLArrowLine, used for the graphic representation of input or output arcs.

1) Instance Variables:

• dataPointer	pointer to a corresponding output or input arc
• startNode	pointer to a shape connected by the start point of an arrow line (Figure 4.6)
• endNode	pointer to a shape connected by the end point of an arrow line (Figure 4.6)

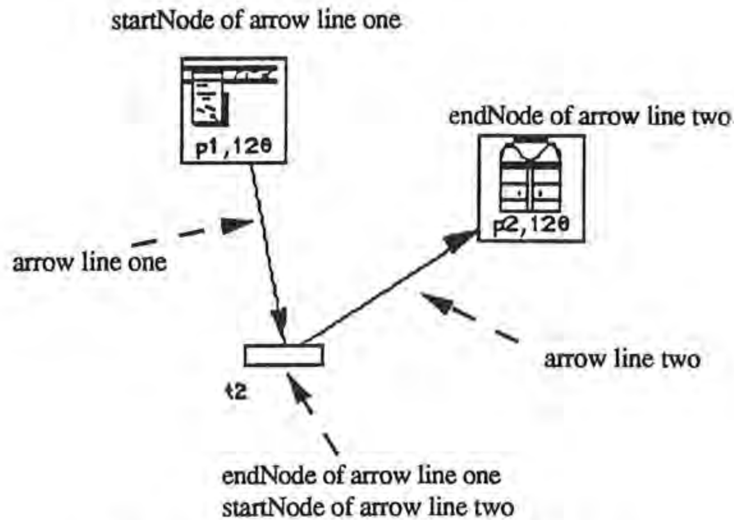


Figure 4.6 Arrow line startNode and endNode.

2) Member Functions:

• CLDrag*	drag an arrow line by changing the values of its locations
• CLIs_a*	return a defined integer to indicate the type of object
• CLReadSelf/CLWriteSelf*	read/write all attributes from/to a file
• SetDataPointer	set the value of **dataPointer**
• GetDataPointer	return the value of **dataPointer**
• SetStartNode	set the value of **startNode**
• GetStartNode	return the value of **startNode**
• SetEndNode	set the value of **endNode**
• GetEndNode	return the value of **endNode**
• NodesHilighted	return a boolean value to indicate if both **startNode** and **endNode** are highlighted
• ShouldBeDuplicated	return a boolean value to indicate if an arrow line should be duplicated
• Adjust	make a drawing of an arrow line clear

• Duplicate*	duplicate an object
• DupDetail*	duplicate the details of an object (called by "Duplicate").

PNTwoLines

PNTwoLines is a subclass of CLSegment and is used to implement a pair of lines to represent the graphics view of an initial transition.

1) Instance Variable:

• dataPointer	pointer to a corresponding initial transition

2) Member Functions:

• CLDrawFrame*	draw a pair of lines
• CLIs_a*	return a defined integer to indicate the type of object
• CLReadSelf/CLWriteSelf*	read/write all attributes from/to a file
• SetDataPointer	set the value of **dataPointer**
• GetDataPointer	return the value of **dataPointer**
• Duplicate*	duplicate an object
• DupDetail*	duplicate the details of an object (called by "Duplicate")

PNRect

PNRect is a subclass of CLRectangle, used to represent the graphics view of a regular transition or a quit transition.

1) Instance Variables:

• dataPointer	pointer to a corresponding transition

2) Member Functions:

• CLIs_a*	return a defined integer to indicate the type of object
• CLReadSelf/CLWriteSelf*	read/write all attributes from/to a file
• SetDataPointer	set the value of **dataPointer**
• GetDataPointer	return the value of **dataPointer**
• Duplicate*	duplicate itself
• OwningIconHilighted	return a boolean value to indicate if an icon, representing the place which owns a transition represented by this rectangle, is hilighted
• Duplicate*	duplicate an object
• DupDetail*	duplicate the details of an object (called by "Duplicate")

PNLabel

PNLabel is a subclass of CLLabel. It shows the id of a transition. It keeps a pointer to the graphics object which is labeled.

1) Instance Variables:

• cpdShape	pointer to a labeled shape

2) Member Functions:

• CLDrawFrame*	draw text
• CLIs_a*	return a defined integer to indicate the type of object

- CLReadSelf/CLWriteSelf* — read/write all attributes from/to a file
- GetCpdShape — return the value of **cpdShape**
- Duplicate* — duplicate an object
- DupDetail* — duplicate the details of an object (called by "Duplicate")

4.4. PNGraphicsView Class

The PNGraphicsView class is responsible for the manipulation of shapes displayed in a window and user interactions. The storage structure (Figure 4.7) fShapeList contains a list of pointers directed toward instances of standard data structures. Standard data structures, such as linked lists, are defined in the OSU Framework data structure library. Pointers to shapes are stored in these instances of data structures and CLGraphicsView navigates the shapes by iterating its storage structures, based upon the following algorithm:

```
CLIter         nextDataStructure(fShapeList);
CLCollection   *aDataStructure ;
CLShape        *aShape;
while (aDataStructure = (CLCollection *) nextDataStructure())
{
    CLIter next(aDataStructure);

    while (aShape = (CLShape *) next())
        ......
};
```

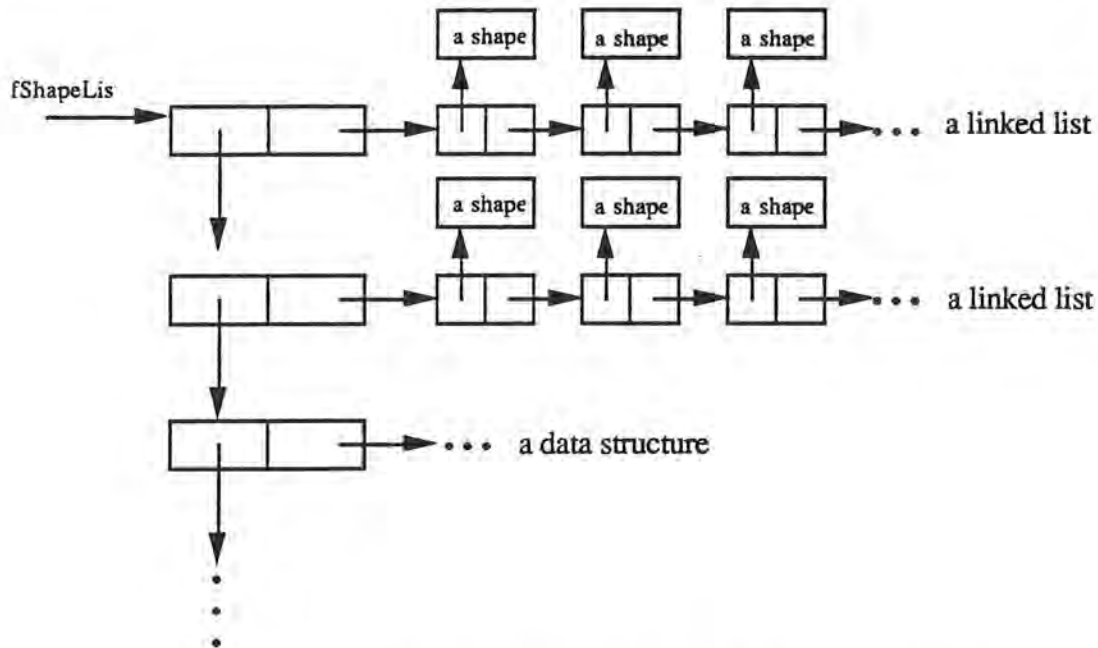


Figure 4.7 Storage structure (fShapeList) of CLGraphicsView.

For undo/redo or paste actions, CLGraphicsView implements fUndoShapeList and fClipShapeList, each of which have a storage structure (Figure 4.7) similar to that for fShapeList. The subclasses of PNGraphicsView, in relation to CLGraphicsView, reflects the following instance variables:

- CLObjList *fPNShapeList;
- CLObjList *fPNUndoShapeList;
- CLObjList *fPNClipShapeList;
- PNHelper *fHelper;
- Boolean fShowDetail;

The variables fPNShapeList, fPNUndoShapeList, and fPNClipShapeList are instances of CLObjList and their functionalities are as follows: 1) fPNShapeList stores pointers toward shapes which are to be displayed on the screen; 2) fPNUndoShapeList stores pointers toward shapes which are to be used when an "undo" action is taken; and 3) fPNClipShapeList stores pointers for the shapes which are duplicated when a paste

action is undertaken. It follows that fPNShapeList is added to fShapeList to enable manipulation by the methods defined for CLGraphicsView of shapes stored in fPNShapeList. In similar fashion, fPNUndoShapeList is added to fUndoShapeList and fPNClipShapeList is added to fClipShapeList. In turn, fShowDetail notes whether or not the view is in the dialog pop-up mode for requesting additional information. The boolean value of fShowDetail is set when the user checks or unchecks "Show Details" from the "Edit" menu. Finally, fHelper is considered in the following section.

PNHelper

The change in shapes will affect corresponding data elements, it is also true that the update of data elements may affect corresponding shapes. Consistency between shapes and data elements must be maintained by some objects. However, it is not good idea to demand that PNGraphicsView or PNPetriNet enforce the principle of consistency since that would serve to reduce the degree of cohesion within the two classes [Budd 90, Lewis 90]. Thus, PNGraphicsView is devoted primarily to the manipulation of shapes, whereas PNPetriNet was developed for the maintenance of data elements within a Petri net; each is required to focus upon its defined and specific purpose, maintaining internal methods directed to that end. A Petri net may be represented as a shape in the PN editor, but may be represented in a different manner in a graphical sequencer; that is, it should be easily understood, easily extracted from specific applications, and easily reused in new situations.

PNGraphicsView maintains an instance variable, fHelper of the class PNHelper. PNHelper is attributed the complicated task of maintaining consistency between shapes and data. After a shape (or several shapes) is created or edited,

PNGraphicsView invokes fHelper to maintain consistency. Thus, the PNHelper class is defined in the following code:

```
class PNHelper : public CLObject {
protected:
    PNPlaceHelper *fPlaceHelper;
    PNTransitionHelper*fTransitionHelper;
    PNInputArcHelper  *fInputArcHelper;
    PNOuputArcHelper  *fOutputArcHelper;
    PNGraphicsView    *fTheView;
    CLObjList      *fPNClipDataList;
public:
    Boolean CreateData(CLShape*aShape, Boolean pShowDetail);
    void    DoubleClick(CLShape *clickedShape);
    void    DuplicateData(CLObjList *aShapeList);
    void    ReleaseData(CLObjList *aShapeList);
    void    AttachData(CLObjList *aShapeList);
    void    Reset();
    PNHelper(PNGraphicsView *pTheView);
    ~PNHelper();
};
```

The class PNHelper has instances of the following classes: PNPlaceHelper, PNTransitionHelper, PNInputArcHelper, and PNOutputArcHelper. The major methods for these four helper classes are described as follows:

1) PNPlaceHelper

• GetPlaceInfor	obtain details regarding a place
• UpdatePlaceInfor	help a place update information
• BringDialog	invoke a dialog box to ask for detailed
• HelpCreation	help create a place
• DoubleClick	obtain information from a place, display the information in a dialog box, obtain user update, then help modify place information

2) PNTransitionHelper

• GetTransitionInfor	obtain transition details
• UpdateTransitionInfor	help a transition update information

- BringDialog — invoke a dialog box to ask for detailed or updated transition information
- HelpCreation — help create a transition
- DoubleClick — obtain transition information, display the information in a dialog box, obtain user update, then help modify transition information

3) PNInputArcHelper

- HelpCreation_Phase1 — help create an input arc
- HelpCreation_Phase2 — help create an input arc

4) PNOutputArcHelper

- GetOutputArcInfor — obtain details regarding an output arc
- UpdateOutputArcInfor — help an output arc update its information
- BringDialog — invoke a dialog box to ask for detailed or updated information about an output arc
- HelpCreation — help to create an output arc
- DoubleClick — obtain information from an output arc, display the information in a dialog box, obtain user update, then help modify output arc information

As may be seen from the above definitions, these classes are used to serve as data element interfaces for a Petri Net model. Since they are responsible for displaying detailed base data and obtaining updated information through the invocation of dialog boxes, they may be treated as observer classes [Budd 90] for the base data elements in a Petri Net. The instance variables of these classes are used by the PNHelper methods "CreateData" and "DoubleClick".

1) **CreateData**: According to the type of the newly created shape, PNHelper asks an observer object to assist in the creation of a Petri Net storage object.

2) **DoubleClick**: PNHelper asks an observer object to accommodate double-click actions in accordance with the type of the parameter, "clickedShape."

In addition, "DuplicateData" is called when a cut, copy, or paste action is performed in PNGraphicsView, "ReleaseData" is called when a cut or undopaste action is performed, and "AttachData" is called when an undocut or paste action is performed. Finally, "Reset" is called to reset fPNClipDataList for the next cut or copy action.

In the remaining subsections, the variable actions for PNGraphicsView are considered.

Dragging

Before a shape can be dragged, the previous shape location is saved and combined with the view notify region in a new notify region. After a shape is dragged, the new shape location is set and combined with the view notify region in a new notify region. To undo dragging actions, PNGrapicsView unites shape regions with notify regions, switching the values of previous and current shape locations and uniting them with the new region. To redo a dragging action, PNGraphicsView repeats an "undo the drag" action.

For dragging action, only shapes' information can be updated. This is because the Petri Net storage objects are not concerned with dragging action. The methods of PNGraphicsView related to dragging action may be summarized as follows:

• PNBeforeDrag	record old positions and invalidate the regions for shapes which are to be dragged (called by "UserBeforeTrackMove*")
• PNCountTotalOffset	determine the offset for dragging
• PNAfterDrag	record the new locations and invalidate the new regions for dragged shapes (called by "UserAfterTrackMove*")
• CLUserDuringTrackMove*	draw a rubber band
• CLDragUndo*	undo a drag action
• CLDragRedo*	redo a drag action

Cutting

PNGraphcisView reviews fPNShapeList, removing any shapes which are to be cut and adding them to fPNUndoShapeList. Each removed shape is checked to determine if should be duplicated and added to fPNClipShapeList. If the response is positive, PNGraphcisView asks the shape to duplicate itself and returns the duplicated copy, which is then added to fPNClipShapeList for subsequent paste actions. The regions of the shapes removed from fPNShapeList are united with the notify region of the view.

After the shapes are processed, PNGraphicsView asks fHelper to revised the data (i.e., the storage structures). PNGraphicsView sends the message "DuplicateData" to fHelper, passing fPNClipShapeList to fHelper to assist in

duplicating the data in correspondence with the shapes stored in fPNClipShapeList. PNGraphicsView then sends the message "ReleaseData" to fHelper, providing fPNUndoShapeList as a parameter. fHelper directs the Petri net model to remove data in correspondence with the shapes stored in fPNUndoShapeList.

To undo a cut action, PNGraphicsView asks fHelper to revise the data, passing fUndoShapeList to send the message "AttachData" to fHelper. After recovery of the data, PNGraphicsView removes the shapes from fPNUndoShapeList and adds them back into fPNShapeList, where the processed shape regions are combined in the notify region of the view. In addition, to redo a cut action, PNGraphicsView repeats the steps taken for cut actions.

The PNGraphicsView methods related to cut action may be summarized as follows:

• PNCut	undertake a cut action, a method called by the "Edit" menu and "UserRedo*"
• PNPlusAClipShape	duplicate a shape and add the new copy to fPNClipShapeList, a protected method called by "PNCut" and "PNCopy"
• PNCutUndo	undertake an undo cut action, called by "UserUndo*"

Copying

PNGraphicsView reviews fPNShapeList and duplicates shapes which are to be copied. The duplicated copies are inserted in fPNClipShapeList for subsequent paste actions. To make copies of the data, PNGraphicsView sends the message "DuplicateData" to fHelper, passing the fPNClipShapeList to fHelper. To undo a copy action, fPNClipShapeList is reset for the next copy or cut action and fHelper is asked

to reset its fPNClipDataList value. To redo a copy action, PNGraphicsView repeats the actions undertaken for a copy action.

The PNGraphicsView method for copy action:

- PNCopy — undertake a copy action, called by the "Edit" menu and "UserRedo*"

Pasting

PNGraphicsView reviews fPNClipShapeList and duplicates shapes. The duplicated copies are added to fPNShapeList and fPNUndoShapeList. To display the new shapes, the regions of new shapes are united with the notify region of the view. After processing , PNGraphicsView sends the message "DuplicateData" to fHelper with fPNUndoShapeList given as the parameter. Then, fHelper iterates fPNUndoShapeList and obtains a data pointer for each shape in the list. The data are asked to duplicate themselves and to return copies, whereupon fHelper asks the Petri Net model to add the new elements to its lists.

To undo a paste action, PNGraphicsView reviews fPNUndoShapeList and removes the shapes from fPNShapeList which appear in fPNUndoShapeList. The shape regions are then united with the notify region of the view. After processing, PNGraphicsView sends the message "ReleaseData" to fHelper with fPNUndoShapeList given as the parameter and fHelper reviews the fPNUndoShapeList and obtains data pointers for the shapes. By sending a message to the Petri Net model and passing the data pointer as a parameter, fHelper asks the Petri Net model to remove the data from the model. To redo a paste action, PNGraphicsView repeats the actions taken for a paste action.

The PNGraphicsView methods related to paste actions are summarized as follows:

- PNPaste — undertake a paste action, called by the "Edit" menu and "UserRedo*"
- PNUndoPaste — undo a paste action, called by "UserUndo*"

Creating

To create a shape, by default PNGraphicsView uses the current shape tool set when the palette tool icon is highlighted. The region of the created shape is united with the notify region of the view and the shape pointer is added to fPNShapeList and fPNUndoShapeList. PNGraphicsView sends the message "CreateData" to fHelper with the newly created shape and a boolean value for fShowDetail. PNGraphicsView receives the boolean value returned from "CreateData." If the boolean value is false, PNGraphicsView removes the shape from fPNShapeList.

To undo a create action, PNGraphicsView undertakes an undo paste action, removing the shapes appearing in fPNUndoShapeList from fPNShapeList and asking fHelper to remove the corresponding data from the Petri Net model. The shape regions in fPNUndoShapeList are then united with the notify region of the view. To redo a create action, PNGraphicsView repeats the actions necessary for an undo cut action, adding the shapes appearing in fPNUndoShapeList to fPNShapeList and asking fHelper to add the corresponding data back into the Petri Net model. The shape regions in fPNUndoShapeList are then united with the notify region of the view.

The PNGraphicsView methods related to a create action are summarized as follows:

• PNPtInIcon	return an icon which a point is placed within (called by "PNMouseDownLegally" and "PNMouseReleaseLegally")
• PNPtInRect	return a rectangle which a point is within (called by "PNMouseDownLegally" and "PNMouseReleaseLegally"),
• PNMouseDownLegally	determine if the mouse has been clicked in a legal area (called by "UserBeforeTrackMove*")
• PNMouseReleaseLegally	determine if the mouse has been released in a legal area (called by "UserAfterTrackMove*")
• PNBringStopAlert	invoke a stop alert telling the user that the mouse has not been clicked or released in a legal area (called by "UserAfterTrackMove*" and "UserBeforeTrackMove*")
• CLReleaseMouse*	a method which is overridden to accomplish special purposes for the creation of a new shape, such as asking fHelper to create a new data element corresponding to the newly created shape
• CLUserNewShape*	a method which is overridden to create a new shape in accordance with the current shape tool

Other Methods

In CLGraphicView, to accomplish certain domain-specific tasks, PNGraphicsView overrides all methods whose names begin with "User." These methods are defined as follows:

• UserAnalysisEvent*	for additions to the "AnalysisEvent" of CLGraphicsView, special conditions must be checked
• UserCheckAction*	determine which actions will be taken

- UserBeforeTrackMove* — completes set-up tasks before the mouse can be dragged, such as checking syntax to determine if a creation action is involved, or calling "PNBeforeDrag" prior to a drag action,
- UserAfterTrackMove* — completes certain tasks after the mouse is dragged, such as checking syntax if a creation action, or calling "PNAfterDrag" if a drag action
- UserUndo* — undo an action
- UserRedo* — redo an action

5. Experience with the Petri Net Editor

In this chapter, four example applications are implemented with the PN editor, including *MiniDraw*, a help system, a calculator, and a record query. The intent of the experiment was to determine the degree to which use of the editor would reduce programming time and effort required to implement a GUI application.

5.1. MiniDraw

The *MiniDraw* application supports multiple concurrently displayed windows, cut-and-paste editing operations, reading and writing data to and from document files on disk, undo and redo of multiple commands, and setting patterns for drawing a variety of shapes. Figure 5.1 shows a basic screen dump for this application.

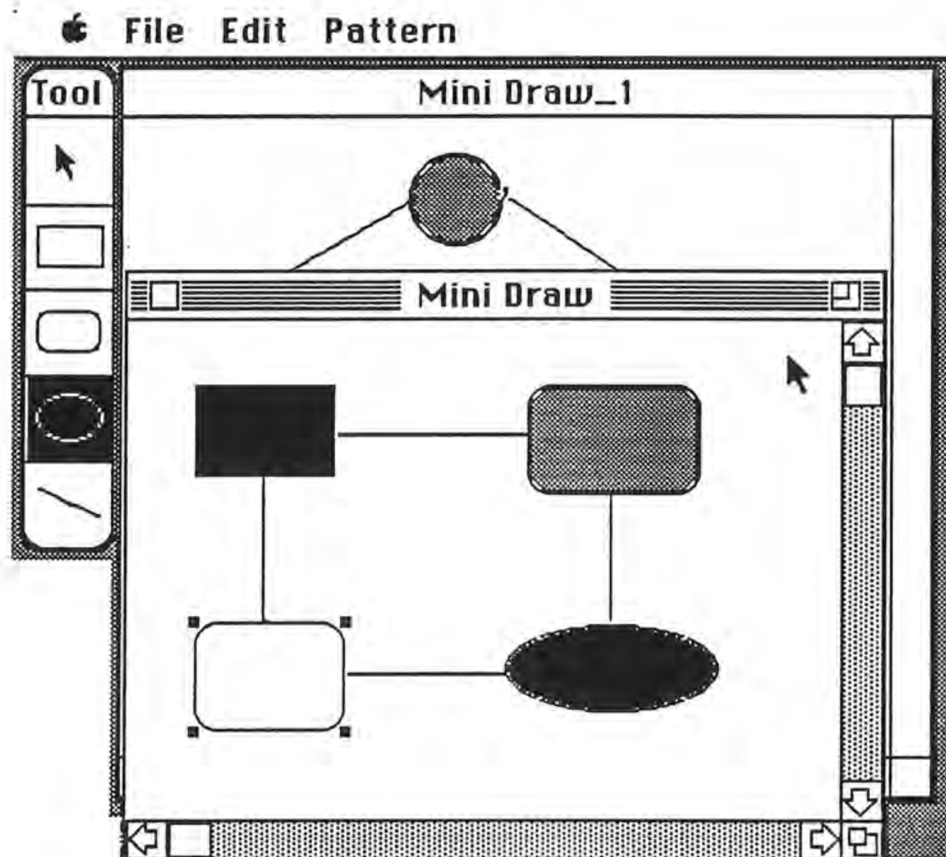


Figure 5.1 Basic screen for Mini Draw .

The design for this example is specified in Figure 5.2. The annotated output arcs for this Petri net are listed as follows (i.e., each row indicating a single output arc, the "from" column indicating the transitions from which the output arcs are drawn, the "to" column indicating places to which output arcs are connected, and the "seq. no." column indicating the sequence numbers inscribed on the output arcs).

from	to	messages	seq. no.	comments
t1	p1		1	
t1	p2		2	
t1	p3		3	
t1	p4	CheckMenuItem(3)	4	check "White" menu item
t1	p7		5	
t1	p5	HilightItem(1)	6	hilight item 1 of palette tool
t3	p2	EnableMenuItem(3)	0	enable "Close" menu item
		EnableMenuItem(5)		enable "SaveAs" menu item

from	to	messages	seq. no.	comments
		DisableMenuItem(4)		disable "Save" menu item
t3	p7		0	
t4	p2	EnableMenuItem(3)	0	enable "Close" menu item
		EnableMenuItem(4)		enable "Save" menu item
		EnableMenuItem(5)		enable "SaveAs" menu item
t4	p7	DoOpen()	0	open a file with a SFGetfile dialog
t5	p2		0	
t6	p2		0	
t6	p7	DoSave()	0	save the file
t23	p2	EnableMenuItem(4)	0	enable "Save" menu item
t23	p7	DoSaveAs()	0	do "SaveAs"
t8	p3		0	
t8	p7	Undo()	0	undo the action
t9	p3		0	
t9	p7	Redo()	0	redo the action
t10	p7	aView=GetViewByName("CLGraphicsView")		
			1	get the graphics view from window
t10	p3	EnableMenuItem(5)	2	enable "Paste" menu item
		aView::DoCut()		ask aView to do a cut action
t11	p7	aView=GetViewByName("CLGraphicsView")		
			1	get the graphics view from window
t11	p3	EnableMenuItem(5)	2	enable "Paste" menu item
		aView::DoCopy()		ask aView to do a copy action
t12	p7	aView=GetViewByName("CLGraphicsView")		
			1	get the graphics view from window
t12	p3	aView::DoPaste()	2	ask aView to do a paste action
t13	p7	aView=GetViewByName("CLGraphicsView")		
			1	get the graphics view from window
t13	p3	EnableMenuItem(3)	2	enable "Cut" menu item
		EnableMenuItem(4)		enable "Copy" menu item
		aView::DoSelectAll()		select all shapes
t14	p7	aView=GetViewByName("CLGraphicsView")		
			1	get the graphics view from window
t14	p4	CheckOnlyItem(1)	2	check "Black" menu item
		aView::SetPattern(1)		set pattern to "Black"
t15	p7	aView=GetViewByName("CLGraphicsView")		
			1	get the graphics view from window
t15	p4	CheckOnlyItem(2)	2	check "Gray" menu item
		aView::SetPattern(2)		set pattern to "Gray"
t22	p7	aView=GetViewByName("CLGraphicsView")		
			1	get the graphics view from window
t22	p4	CheckOnlyItem(3)	2	check "White" menu item
		aView::SetPattern(3)		set pattern to "White"
t16	p7	aView=GetViewByName("CLGraphicsView")		
			1	get the graphics view from window
t16	p5	HilightItem(1)	2	hilight "selectionTool" icon
		aView::CLSetCurrentShapeTool(selectionTool)		
				set the current tool to "selectionTool"

from	to	messages	seq. no.	comments
t17	p7	aView=GetViewByName("CLGraphicsView")	1	get the graphics view from window
t17	p5	HilightItem(2)	2	hilight "selectionTool" icon
		aView::CLSetCurrentShapeTool(Rectangle)		set the current tool to "Rectangle"
t18	p7	aView=GetViewByName("CLGraphicsView")	1	get the graphics view from window
t18	p5	HilightItem(3)	2	hilight "RoundRect" icon
		aView::CLSetCurrentShapeTool(RoundRect)		set the current tool to "RoundRect"
t19	p7	aView=GetViewByName("CLGraphicsView")	1	get the graphics view from window
t19	p5	HilightItem(4)	2	hilight "Oval" icon
		aView::CLSetCurrentShapeTool(Oval)		set the current tool to "Oval"
t21	p7	aView=GetViewByName("CLGraphicsView")	1	get the graphics view from window
t21	p5	HilightItem(5)	2	hilight "Line" icon
		aView::CLSetCurrentShapeTool(Line)		set the current tool to "Line"

For this example, messages inscribed on output arcs are methods inherited from the OSU Framework, and "DoCut", "DoPaste", "DoCopy", "DoSelectAll", "SetPattern", and "CLSetCurrentShapeTool" are CLGraphicsView methods. Due to the design of the OSU Framework, a CLGraphicsView object can not be directly accessed. Thus, to send a message to the view within a window, "GetViewByName" is sent to the window place to obtain a pointer for the CLGraphicsView object. The returned pointer is stored in a variable, **aView**, which can then be declared a local variable for the method generated for that transition, or it can be declared as an instance variable for the place sending the message.

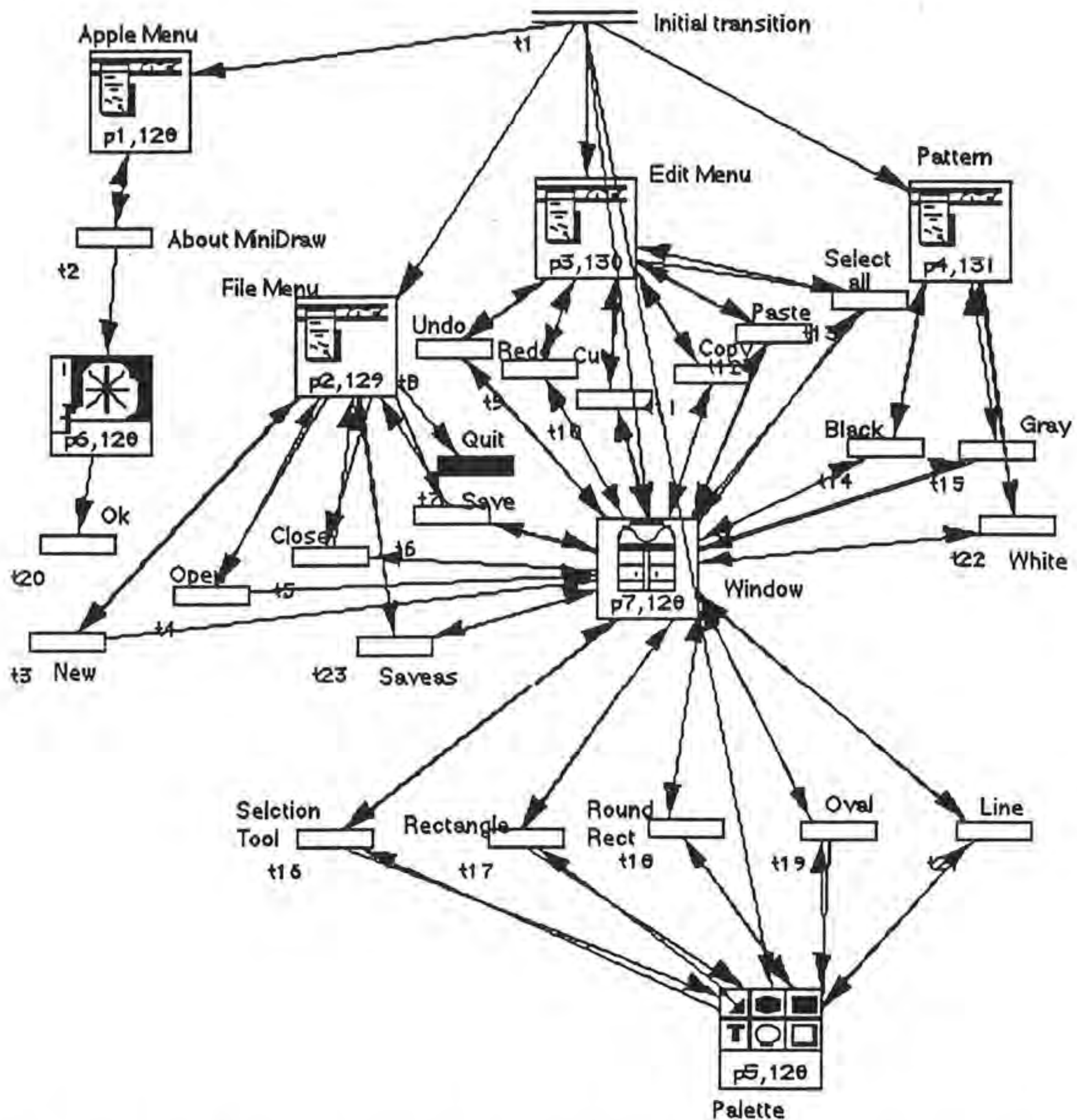


Figure 5.2 Petri net representation for MiniDraw example (screen dump with added textual explanations of entities).

For this example, 529 lines of code were automatically generated and 16 lines of code were added by hand. The code added by hand is composed of "include" statements and the statements declaring "aView" as a local variable. The source programs for this application are listed in Appendix B.

5.2. Help System

It is intended that the help system is integrated with the PN editor to serve as an on-line help system and as a means to illustrate the utility of the Petri net editor. To invoke the help system, the user selects "Help" from the "Others" menu (Figure 5.3). (Note that when this system is integrated into the editor, the "Quit" item will be eliminated and the "Others" menu will become the fifth menu main title.) To introduce this system, a modeless dialog box is popped-up, from which two buttons, "Exit" or "Next Page," can be selected to obtain additional information (Figure 5.4).

Figure 5.3 Help system menu.

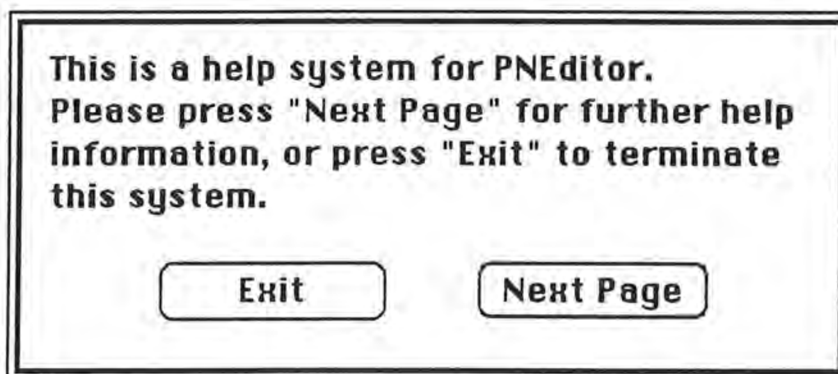


Figure 5.4 Dialog box for the help system.

If the user selects the "Next Page" button, succeeding modeless dialog boxes will appear in correspondence with the user commands. The sequence of these dialog boxes is illustrated in Figure 5.5.

If the user selects the "Next Page" button, succeeding modeless dialog boxes will appear in correspondence with the user commands. The sequence of these dialog boxes is illustrated in Figure 5.5.

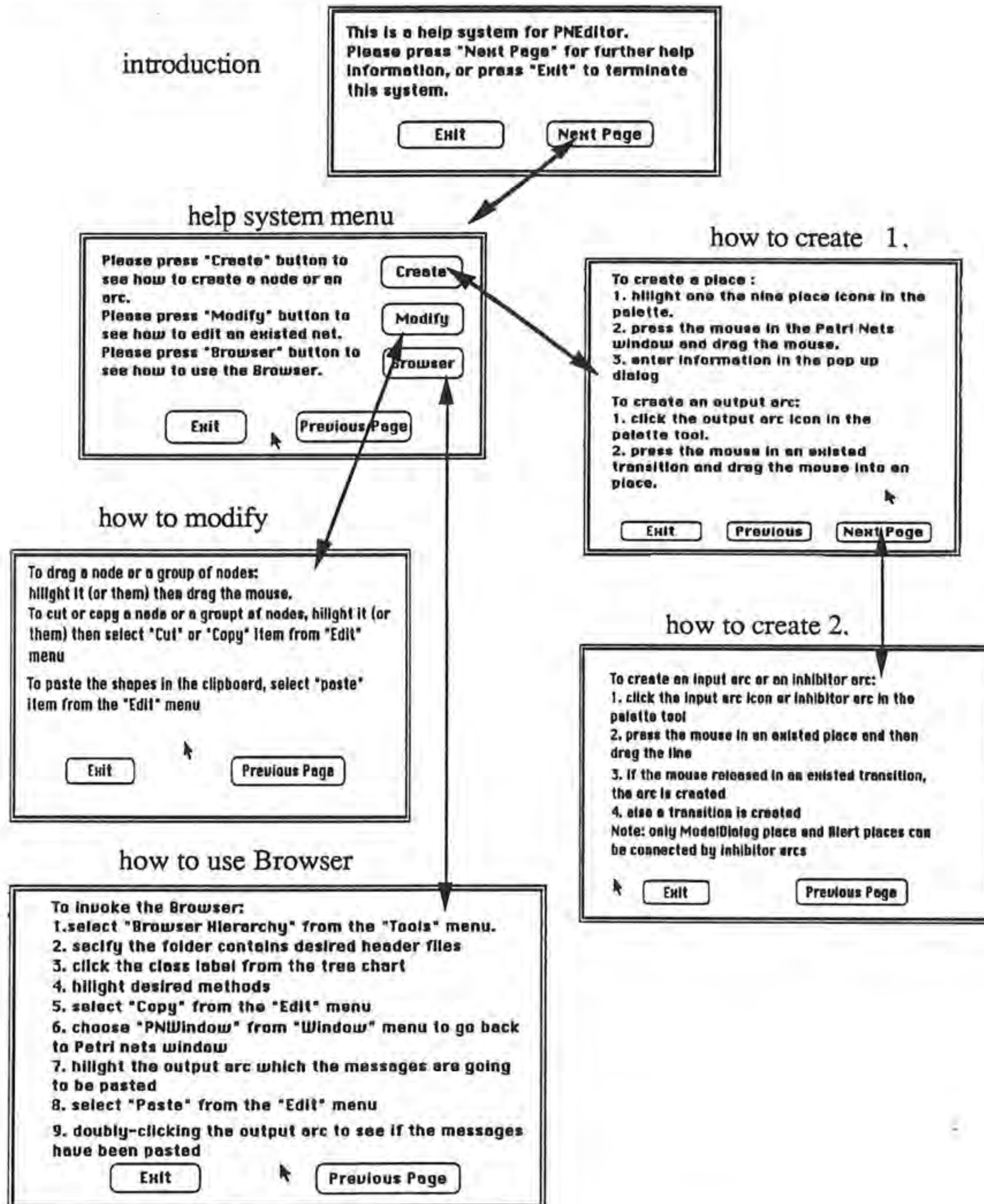


Figure 5.5 Overview of the help system.

The Petri net representation for this example is given in Figure 5.6. One menu place and six modeless dialog places are included in the representation. Relationships among these places are specified by arcs, but no messages are inscribed on any of the output arc. Rather, the connections among arcs, places, and transitions imply the creation and deletion of dialog relationships. The principal advantage for the use of the PN editor for this type of function is that the user obtain a view of an entire system without becoming lost in lists of detailed statements.

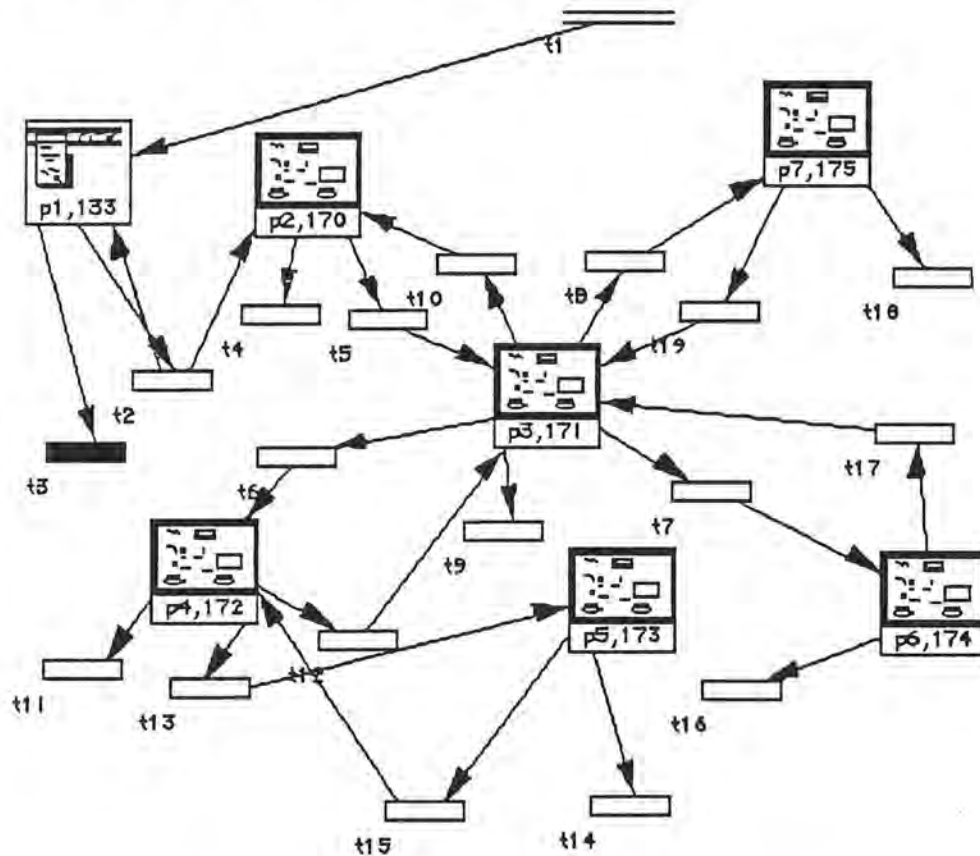


Figure 5.6 Petri net representation for the help system.

The size of the generated programs was 330 lines of code, with 327 lines of code generated by Petri net editor. Three "include" statements were added by hand.

5.3. Calculator

The third example built with the Petri net editor was a calculator. The calculator consisted of a modeless dialog box with buttons for accepting user input and an editText box in which results can be displayed (Figure 5.7). In the example given, a calculator is shown on the screen after the application is launched. The "Off" button (Figure 5.7) is used to turn off the calculator and to quit the application.

Figure 5.7 Calculator and its resource format.

The Petri net representation for this application is shown in Figure 5.8. A dialog place was drawn for the derivation of a subclass from the CLModelessDialog class. Five instance variables, given as follows, were declared for saving application-specific data (Figure 5.9) :

- double fOperand; store the current value of the operand
- double fResult; stores the result of an operation
- char fOperator; stores the arithmetic operator

- char *fStr; stores a number in the form of text
- Boolean hasDecPt; indicates if there is a decimal point within the current operand

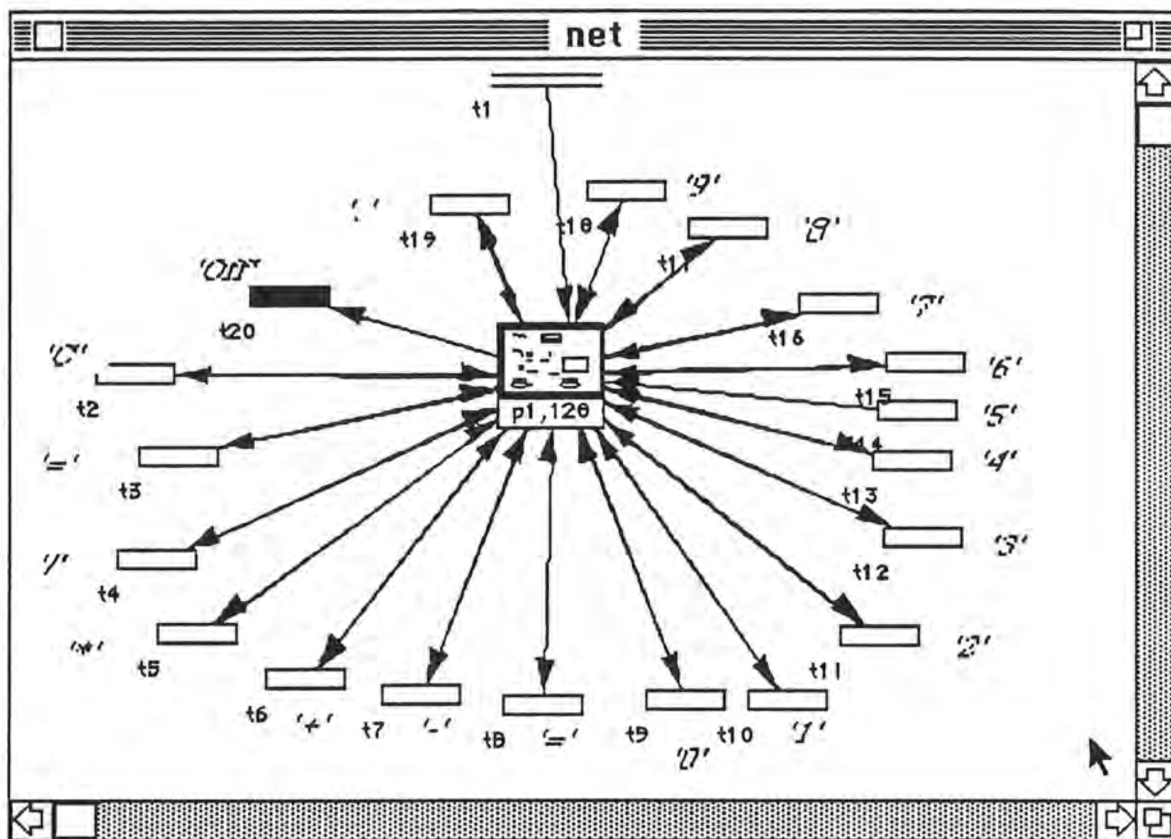


Figure 5.8 Petri net representation for a calculator

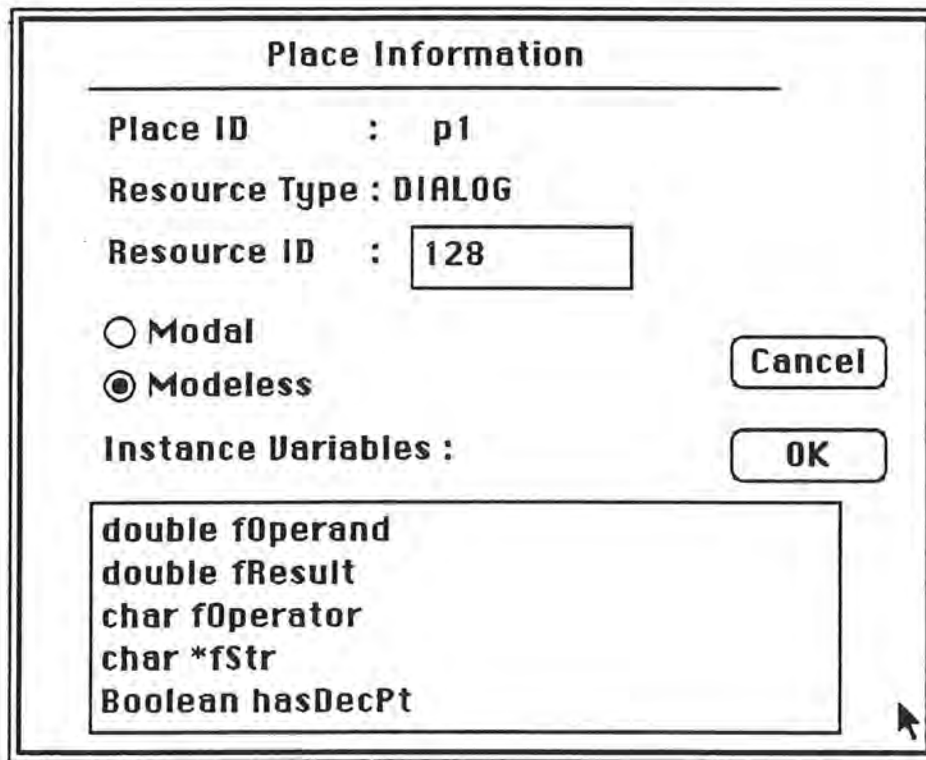


Figure 5.9 Dialog request for information on a dialog place.

As shown in Figure 5.8, 19 transitions were created to represent digital buttons, the decimal point button, the arithmetic operation buttons, and the "Off" button for a calculator. The transition t20, which represents the "Off" button, is a quit transition that terminates the execution of this application. The dialog place is connected by an output arc from the initial transition, indicating that a modeless dialog object is created after the application is started. The message "DoClear()" is sent to the modeless dialog, initializing the instance variables for a new modeless dialog object. This method is also called when the "C" (clear) button is pressed.

When a digital button is selected by the user, the digit is appended to fStr, which is then converted to a number. The value of fStr is then displayed in the editText box, which is the item 1 in the modeless dialog resource representation given in Figure

5.7. Consider this button as an example. The messages inscribed on the output arc (see Figure 5.8) from the transition t10 to the place p1 are

GetDigit("1"),
fOperand=func::atof(fStr),

and

SetItemText(1,fStr),

in which "GetDigit" is a new member function of the dialog place, "atof" is a string convert function defined in the C library, and "SetItemText" is a method from the CLDialog class used to display a string in the editText box (Figure 5.10).

OutputArc Information

Sequence # : 0

Messages :

GetDigit("1")
fOperand=func::atof(fStr)
SetItemText(1,fStr)

Cancel

OK

Predicate :

Figure 5.10 Messages inscribed on the output arc drawn from t10 to p1.

When the first decimal point is entered, '.' is appended to fStr, which is then displayed in the editText box. If a decimal point already exists, then nothing is done when the '.' button is selected. The messages "fStr=func::strcat(fStr,".")", "hasDecPt=true", and "SetItemText(1,fStr)" are inscribed on the output arc from t19 to

p1 (Figure 5.11). The predicate "!hasDecPt" is entered by the user. The statements generated for the transition t20 are as follows :

```
If (!hasDecPt) {
        fStr=strcat(fStr,'.');
        SetItemText(1,fStr);
        hasDecPt=true;
}
```

OutputArc Information

Sequence # : 0

Messages :

fStr=func::strcat(fStr,'.')
SetItemText(1,fStr)
hasDecPt= func::true

Cancel

OK

Predicate :

!hasDecPt

Figure 5.11 Messages and predicate inscribed on the output drawn from t19 to p1.

When an operation button is selected, the saved operation is performed, the result of the operation is displayed in the editText box, and the new operator is saved. This is done by sending messages back to the modeless dialog box. For instance, the following messages are inscribed on the output arc from the transition t6, which represents the '+' button, to p1:

- PerformCalculation() — do the calculation according to the old value in fOperator
- func::num2str(fResult,fStr) — convert the result to a string

- SetItemText(1,fStr) show the result value
- fOperator= func::'+' save the operator for later use
- func::strcpy(fStr,"0") set the string to "0"
- hasDecPt=func::false set the value of hasDecPt to false
- fOperand=func::0 reset the value of fOperand

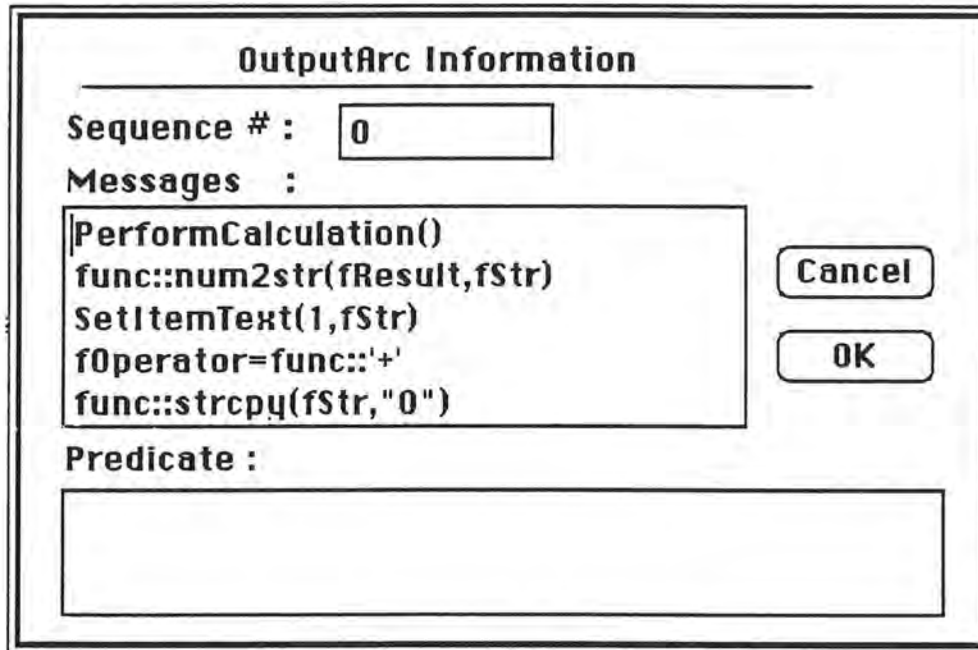


Figure 5.12 Messsages inscribed on the output arc drawn from t6 to p1.

Note that "num2str" is a user-defined function which converts a number type "double" to a string. This example demonstrates how to attach functionalities to user interface objects through the use of messages inscribed on output arcs. The message "PerformCalculation" is an application-specific method for the modeless dialog place and is written by the user. The code for"PerformCalculation" should appear as follows:

```
switch(fOperator) {
      case '+':
         fResult=fResult+fOperand;
         break;
      case '-':
         fResult=fResult-fOperand;
         break;
      case '*':
         fResult=fResult*fOperand;
         break;
      case '/':
         fResult=fResult/fOperand;
         break;
      case '=':
         break;
   }
```

Some redundant segments of generated code can be abstracted into separate procedures to improve the program readability. For example, the following code segment appears in the methods for performing operation buttons, and can be abstracted into a single module for subsequent recall:

```
PerformCalculation();
num2str(fResult,fStr);
SetItemText(1,fStr);
fStr=strcpy(fStr,"0");
hasDecPt=false;
fOperand=0;
```

Most of the code added by hand concerned management of key down events. The method "DoKeyDown" was overridden to convert the keycodes for pressed keys to the number of items in the modeless dialog box. This example demonstrates one of the limitations of the PN editor. That is, an action taken for a key down event cannot be specified with the editor. The size of the programs for this example was 395 lines, of which 285 lines of code were automatically generated and 110 lines were added by hand.

5.4. Record Query

The record query example consists of a system for scrolling personal records within a file. After selecting "Open" from the "File" menu, the user chooses the file to be queried. A query dialog box with four buttons and three editText boxes is popped-up (Figure 5.13) to scroll the file. The "Next" and "Prev" buttons are selected to query the record. To change the contents of a record, new data are typed in and the "Change" button is selected to confirm the update (Figure 5.14).

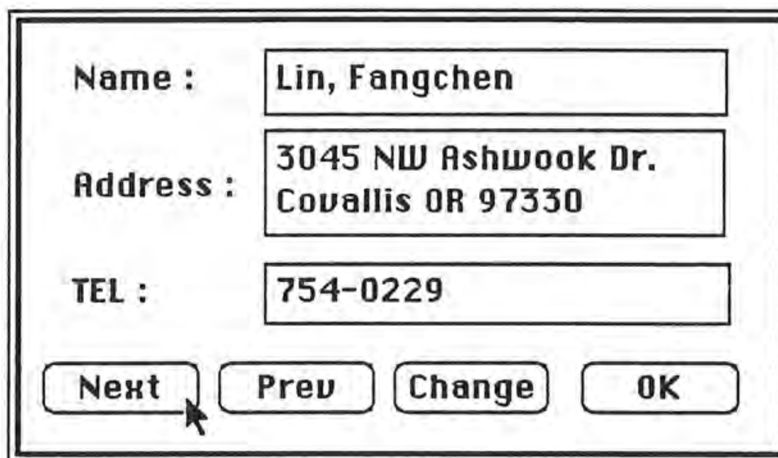


Figure 5.13 Query dialog box.

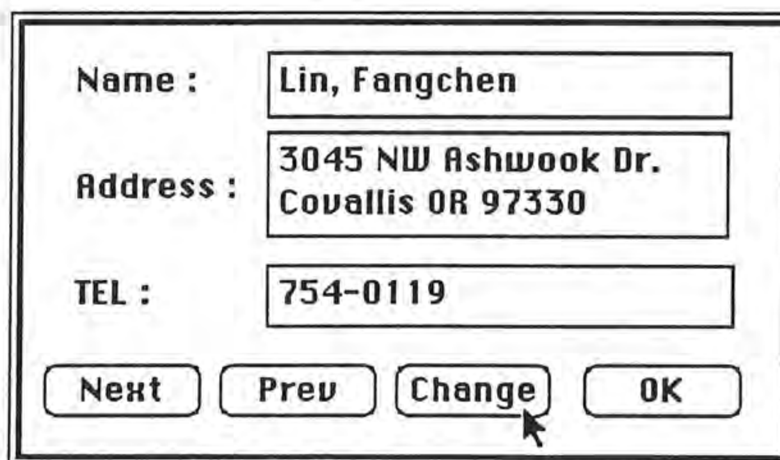


Figure 5.14 Select "Change" button to confirm update.

The Petri net representation of this example is given in Figure 5.15. Two menu places, p1 and p2, are connected by an initial transition. Thus, two menus, the "Apple" menu and the "File" menu, were added to the menu bar after the application was launched. The transition t3 is drawn to represent "Open" and the quit transition t4 is drawn to represent "Quit" in the "File" menu. Note that t3 has three output arcs connecting the menu place, the dialog place, and the window place. Therefore, after selection of "Open" , this view is retained on screen and a window object and a dialog object are created.

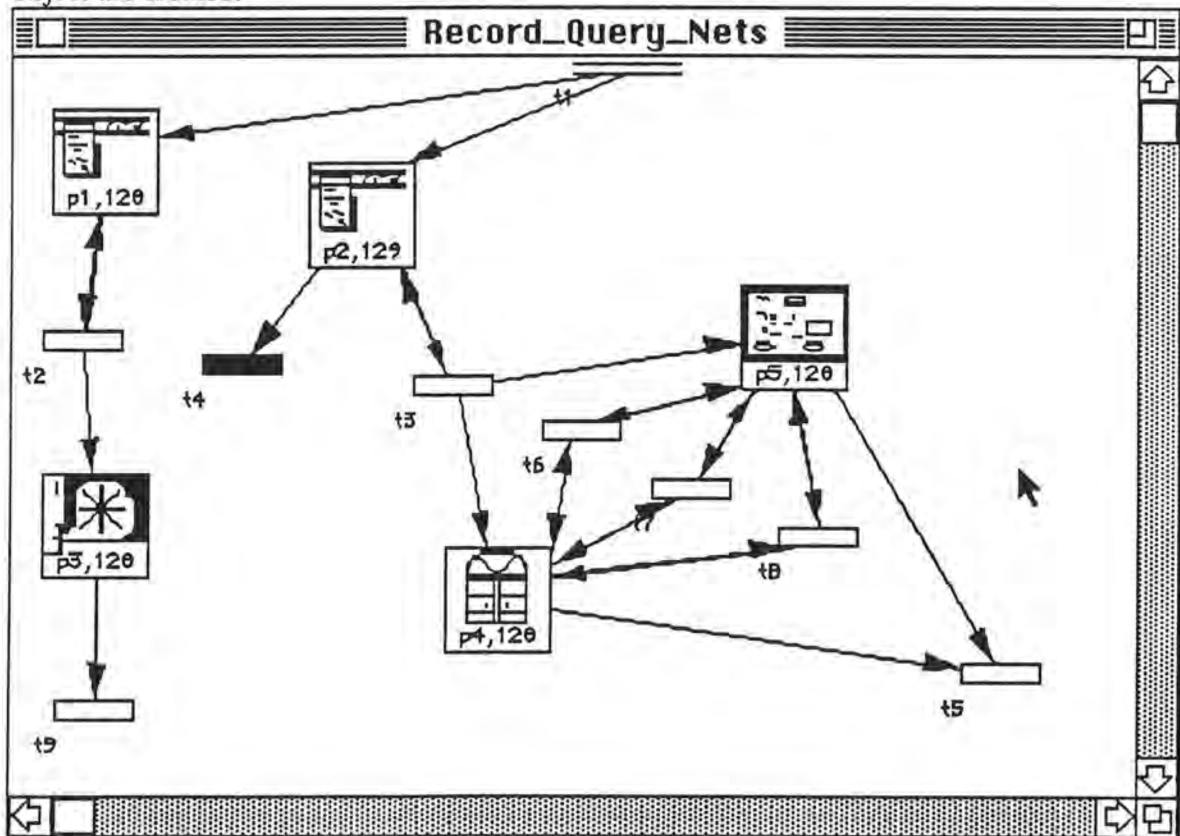


Figure 5.15 Petri net representation for the record query example.

The dialog place has four transitions representing four buttons in the query dialog: t5 for the "Ok" button, t6 for the "Next" button, t7 for the "Prev" button, and t8 for the "Change" button. Note that t5 has two input arcs from the dialog place and the window place, but that it has no output arcs, indicating that after selection of the "Ok" button, the dialog and window objects are deleted; t6, t7 and t8 each have both input and output arcs connecting the dialog and window places, in these cases indicating that the dialog and window objects still continue to exist following selection of the "Next", "Prev", or "Change" buttons.

To get the next record in a file, the dialog place sends the message "fRec=GetNextRecord()" to the window place. The record obtained from the file is returned in the form of a string. To analyze that string and display the contents of the record separately, the dialog place sends the following messages to itself:

```
fName=func::GetToken(fRec,1)
fAddr=func::GetToken(fRec,2)
fPhone=func::GetToken(fRec,3)
SetItemText(5, fName)
SetItemText(6, fAddr)
SetItemText(7, fPhone)
```

"GetToken" is a user-defined function which takes two parameters, a string and an integer n. It parses the string and returns the nth token in the string; "func" is a keyword for indicating that "GetToken" is a function instead of a method; and "SetItemText" is a method inherited from the CLDialog class used to display a string in an editText box. It may be observed that the contents of a personal record are stored in

three strings separately, then displayed in editText boxes. The above messages are inscribed on the output arc drawn from t6 to p5 (Figure 5.16).

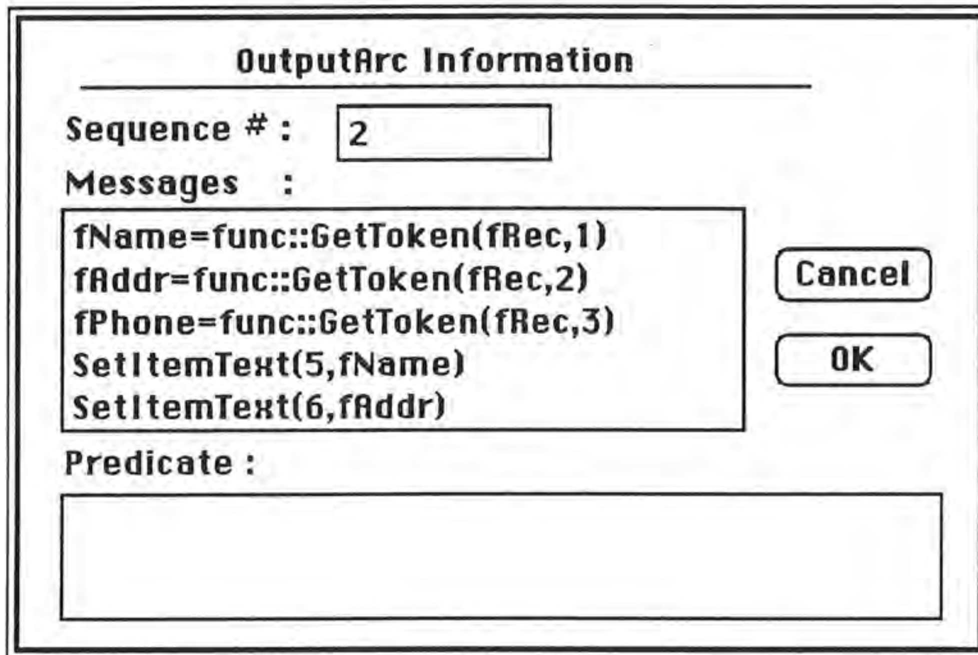


Figure 5.16 Dialog box accepting messages.

The output arcs with messages are summarized as follows.

from	to	messages	no.	comments
t7	p4	fRec=GetPrevRecord()	1	obtain previous record
t7	p5	fName=func::GetToken(fRec,1)	2	get first token
		fAddr=func::GetToken(fRec,2)		get second token
		fPhone=func::GetToken(fRec,3)		get third token
		SetItemText(5,fName)		display the name
		SetItemText(6,fAddr)		display the address
		SetItemText(7,fPhone)		display the phone no.
t6	p4	fRec=GetNextRecord()	1	obtain next record
t6	p5	fName=func::GetToken(fRec,1)	2	get first token
		fAddr=func::GetToken(fRec,2)		get second token
		fPhone=func::GetToken(fRec,3)		get third token
		SetItemText(5,fName)		display the name
		SetItemText(6,fAddr)		display the address
		SetItemText(7,fPhone)		display the phone no.

from	to	messages	no.	comments
t8	p5	fName=GetItemText(5)	1	method of CLDialog
		fAddr=GetItemText(6)		get the text of editText box
		fPhone=GetItemText(7)		
		fRec=func::strcat(fName,"/")		concate fName with "/"
		fRec=func::strcat(fRec,fAddr)		concate fRec with fAddr
		fRec=func::strcat(fRec,"/")		concate fRec with "/"
		fRec=func::strcat(fRec,fPhone)		concate fRec with fPhone
t8	p4	ChangeRecord(fRec)	2	update the record
t3	p4	DoOpen()	1	open a file
		char *lRec=GetNextRecord()		obtain the first record
t3	p5	char*lName=func::GetToken(lRec,1)	2	get first token
		char*lAddr=func::GetToken(lRec,2)		get second token
		char*lPhone=func::GetToken(lRec,3)		get third token
		SetItemText(5,lName)		display the name
		SetItemText(6,lAddr)		display the address
		SetItemText(7,lPhone)		display the phone no.
t3	p2		3	

According to the design principles of the OSU Framework, a document object is responsible for reading and writing data contained in the model from/to disks. For this example, a document class was subclassed from the CLDocument class for: 1) reading the records from a file and building a linked list to store the records, 2) obtaining the next and previous records in the linked list, 3) modifying records, and 4) writing the records in the linked list to a file. However, a document object was created as an instance variable of a window object. Due to encapsulation, the dialog place cannot send messages directly to a document object for record manipulation. The methods implemented in a document object are called by sending messages to the window place, which then send corresponding messages to the document object.

There were 549 lines of code in this example, with 283 lines of code generated with the Code Generator. The statements added by hand concerned creation of the document class, user-defined functions, and "include" statements.

5.5. Statistics

Time and Effort

The time and effort required to implement these four examples is summarized as follows.

	MiniDraw	Help System	Calculator	Query Record
*lines of codes generated	529	327	285	283
*MM (effort)	0.58	0.35	0.30	0.30
*TDEV (time)	2.03	1.67	1.59	1.58
*total lines of codes	545	330	395	549
*MM (effort)	0.6	0.35	0.42	0.6
*TDEV (time)	2.05	1.68	1.81	2.06
*%saving in effort	96%	99%	71%	50%
*%saving in time	99%	99%	87%	76%
*number of places	7	7	1	5
*number of transitions	20	18	20	9
*number of arcs	66	26	38	25
*number of messages	50	0	61	30

Reusability

The following table shows how many objects in the OSU Framework were used, how many subclasses were added, and how many methods in the OSU Framework were reused in each example.

	MiniDraw	Help System	Calculator	Record Query
objects in the framework used	31	15	10	19
number of subclasses added	8	8	2	8
number of methods used	276	71	64	109

Currently, there are 81 classes (objects), 969 methods in the OSU Framework. Figure 6.1, 6.2, 6.3, and 6.4 show the class hierarchies of the Framework with different patterns to indicate which objects were used or subclassed for these four examples.

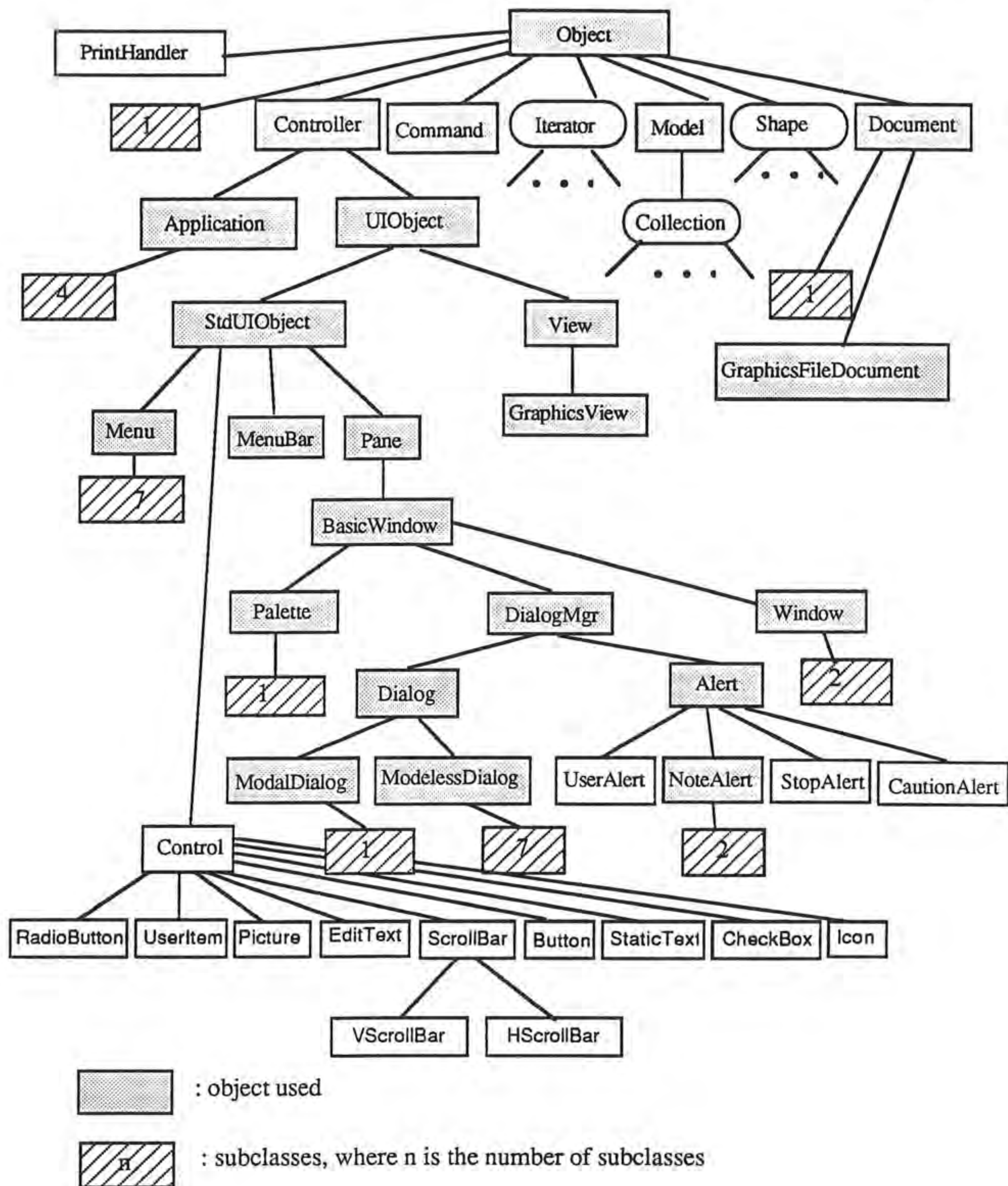


Figure 6.1 Objects used and subclassed , the OSU Framework

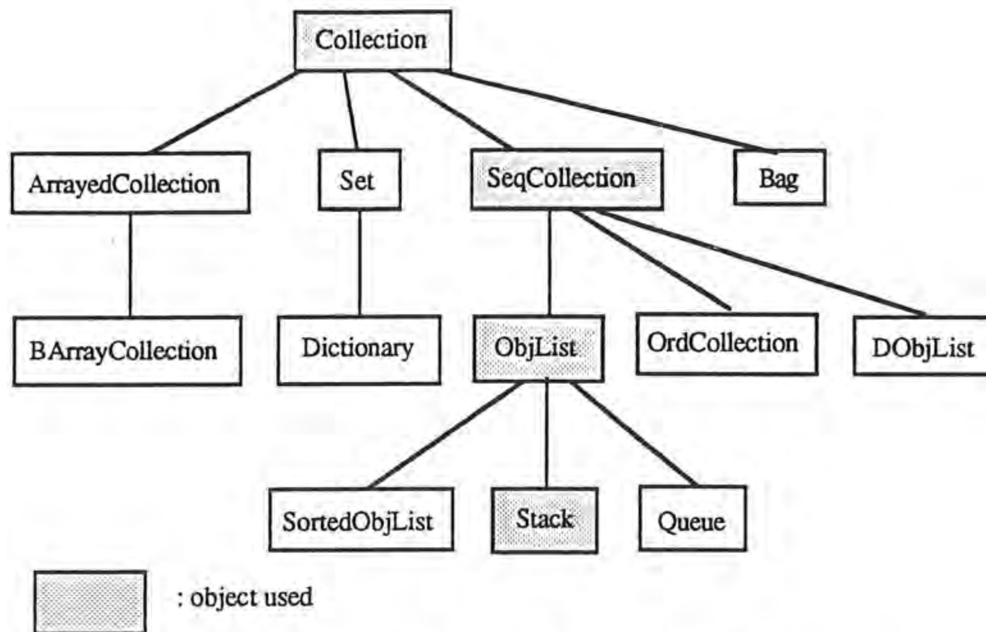


Figure 6.2 Objects used, the Data Structure class hierarchy

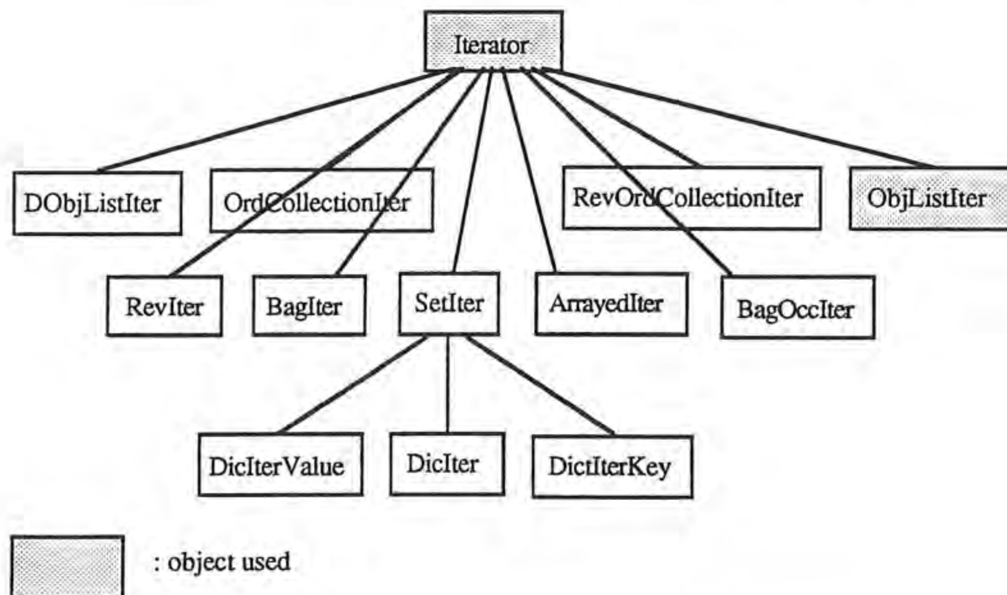


Figure 6.3 Objects used, the Iterator class hierarchy

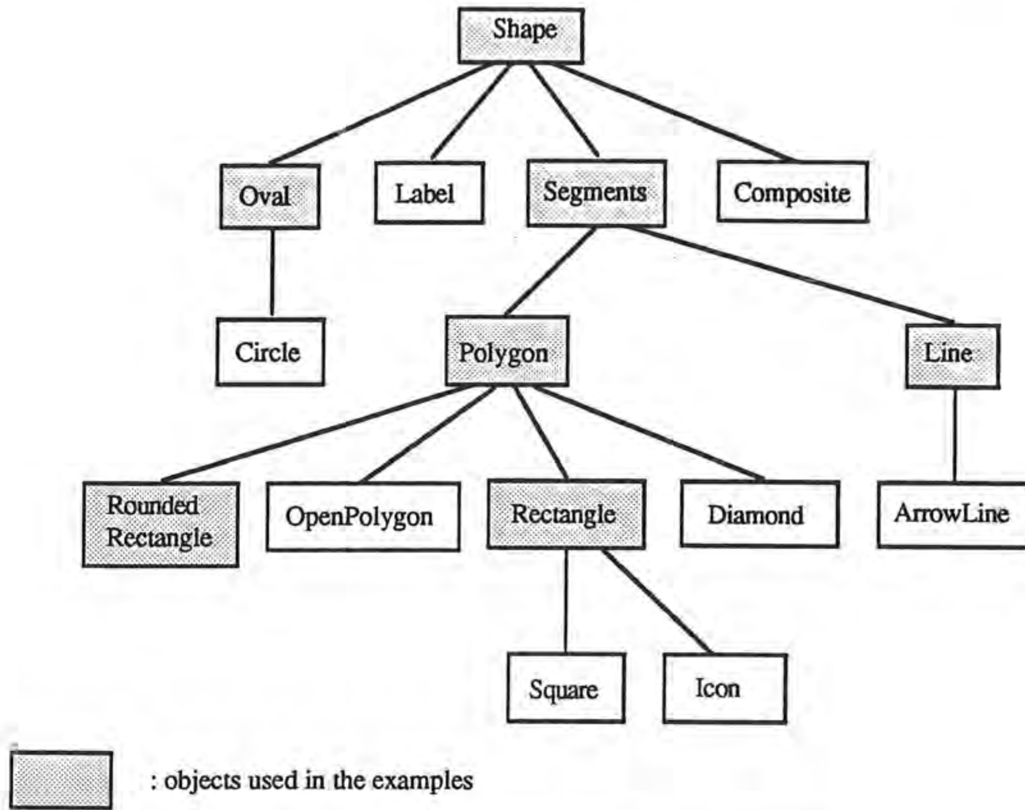


Figure 6.4 Objects used, the Shape Library

6. Conclusion

The implementation of the Petri net editor demonstrates the reusability of the OSU Framework classes, thus reaffirming the basic principle of reusable design. A number of bugs were reported as feedback during the coding phase for the implementation of the Petri net editor.

As demonstrated in Chapter 5, the Petri net editor provides a favorable environment for the construction of GUI application prototypes, saving considerable programming time and effort. Since a graphical editor provides programmers with a useful application overview, this approach avoids the possibility of losing programmers in a confusing mass of statements.

The major limitation of Petri net editor is that interactions among objects within a place cannot be specified. For example, the interactions among a view, a model, and a document, each of which are within a window object, cannot be specified. In the case of deriving subclasses from these classes, it is necessary for the developer to step beyond the use of the Petri net editor and to create domain-specific classes by hand. However, the newly derived classes are relatively easy to create since they can reuse both design and implementation from the OSU Application Framework.

A significant task for the future improvement of the Petri net editor is to revise action specifications for transitions. As noted in Chapter 2, a transition, representing a

mouse-selectable area in a GUI object, initiates action messages which react to user input. They are translated from a derived class method and the annotations inscribed on their output arcs are translated into statements of that method. However, some types of statement structures still cannot be derived from the specifications of output arcs, messages, predicates, and/or the sequence numbers currently provided by the editor. For example, an "if..else" statement cannot be correctly described by using predicates. In addition, a loop structure cannot be represented by the use of any means currently available within the editor.

7. Appendix

Appendix A.

Statistics of Project :

number of classes : 56

lines of source code : 6695 (.h 1525 + .cp 5170)

Statistics of Reusability :

number of classes (objects) in the OSU Framework : 81

number of methods in the OSU Framework : 969

lines of source code in the OSU Framework : 19090 (comments are also counted)

number of objects reused by PN editor : 33

number of methods reused by PN editor : 262

Appendix B.

```
//
// myAlert.h
//
#include "cldialog.h"

#ifndef MYALERT_H
#define MYALERT_H

class NoteAlert_128_6 : public CLNoteAlert {
public:
   NoteAlert_128_6():(128) { SetName("NoteAlert_128_6"); }
   void DoMouseDown(short pItemHit);
   void DoItem1();
}; // end of class NoteAlert_128_6

#endif MYALERT_H

//
// myAlert.cp
//

#include "myAlert.h"

#pragma segment myAlert

void NoteAlert_128_6::DoMouseDown(short pItemHit) {
   switch(pItemHit) {
   case 1:
      DoItem1(); break;
   } // end of switch(pItemHit)
} // end of DoMouseDown(pItemHit)


void NoteAlert_128_6::DoItem1() {// ok button
   DoClose();
} // end of member function

// end of file myAlert.cp

//
// myApplication.h
//
#include "clapplication.h"

#ifndef MYAPPLICATION_H
#define MYAPPLICATION_H

class MyApplication : public CLApplication {
public:
   void Initialize();
}; // end of class MyApplication


#endif MYAPPLICATION_H
```

```
//
// myApplication.cp
//
#include "myApplication.h"
#include "myMenu.h"
#include "myWindow.h"

void MyApplication::Initialize() {
   Menu_128_1  *Menu_128_1Obj = new Menu_128_1;//apple menu
   Menu_128_1Obj->AddRsrc();
   gApplication->fMenuBar->AddMenu(Menu_128_1Obj);
   Menu_129_2  *Menu_129_2Obj = new Menu_129_2;//file menu
   Menu_129_2Obj->DisableMenuItem(4);//disable "Save"
   gApplication->fMenuBar->AddMenu(Menu_129_2Obj);
   Menu_130_3  *Menu_130_3Obj = new Menu_130_3;//edit menu
   gApplication->fMenuBar->AddMenu(Menu_130_3Obj);
   Menu_131_4  *Menu_131_4Obj = new Menu_131_4;//pattern menu
   Menu_131_4Obj->CheckMenuItem(3);//white
   gApplication->fMenuBar->AddMenu(Menu_131_4Obj);
   Window_128_7   *Window_128_7Obj = new Window_128_7;
   Window_128_7Obj->Initialize();
   Window_128_7Obj->DoNew();
   Window_128_7Obj->Draw();
   Palette_128_5  *Palette_128_5Obj = new Palette_128_5;
   Palette_128_5Obj->HilightItem(1);//hilight selection tool
   Palette_128_5Obj->Draw();
} // end of Initialize()


// end of file myApplication.cp

//
// myMenu.h
//
#include "clmenu.h"

#ifndef MYMENU_H
#define MYMENU_H

class Menu_128_1 : public CLMenu {// apple menu
public:
   Menu_128_1():(128) { SetName("Menu_128_1"); }
   class CLCommand * DoMenuCommand(short pMenuItem);
   class CLCommand * DoItem1();
}; // end of class Menu_128_1



class Menu_129_2 : public CLMenu {// file menu
public:
   Menu_129_2():(129) { SetName("Menu_129_2"); }
   class CLCommand * DoMenuCommand(short pMenuItem);
   class CLCommand * DoItem1();
   class CLCommand * DoItem2();
   class CLCommand * DoItem3();
   class CLCommand * DoItem4();
```

```
    class CLCommand * DoItem5();
    class CLCommand * DoItem6();
}; // end of class Menu_129_2


class Menu_130_3 : public CLMenu {//edit menu
public:
    Menu_130_3():(130) { SetName("Menu_130_3"); }
    class CLCommand * DoMenuCommand(short pMenuItem);
    class CLCommand * DoItem1();
    class CLCommand * DoItem2();
    class CLCommand * DoItem3();
    class CLCommand * DoItem4();
    class CLCommand * DoItem5();
    class CLCommand * DoItem6();
}; // end of class Menu_130_3


class Menu_131_4 : public CLMenu {//pattern menu
public:
    Menu_131_4():(131) { SetName("Menu_131_4"); }
    class CLCommand * DoMenuCommand(short pMenuItem);
    class CLCommand * DoItem1();
    class CLCommand * DoItem2();
    class CLCommand * DoItem3();
}; // end of class Menu_131_4

#endif MYMENU_H

//
// myMenu.cp
//
#include "myMenu.h"
#include "myAlert.h"
#include "myWindow.h"
#include "clGraphicsView.h"
#include "myApplication.h"

#pragma segment myMenu

CLCommand * Menu_128_1::DoMenuCommand(short pMenuItem) {
    switch(pMenuItem) {
    case 1:
        return DoItem1();
    } // end of switch(pMenuItem)
} // end of DoMenuCommand(pMenuItem)


CLCommand * Menu_128_1::DoItem1() {// About MiniDraw
    CLCommand    *cmdObj = 0;

    NoteAlert_128_6        *NoteAlert_128_6Obj = new
NoteAlert_128_6;
    NoteAlert_128_6Obj->Draw();
```

```
    return  cmdObj;
} // end of member function


CLCommand * Menu_129_2::DoMenuCommand(short pMenuItem) {
    switch(pMenuItem) {
    case 1:
       return DoItem1();
    case 2:
       return DoItem2();
    case 3:
       return DoItem3();
    case 4:
       return DoItem4();
    case 5:
       return DoItem5();
    case 6:
       return DoItem6();
    } // end of switch(pMenuItem)
} // end of DoMenuCommand(pMenuItem)


CLCommand * Menu_129_2::DoItem1() {//New
    CLCommand    *cmdObj = 0;

    EnableMenuItem(3);
    EnableMenuItem(5);
    DisableMenuItem(4);
    Window_128_7    *Window_128_7Obj = new Window_128_7;
    Window_128_7Obj->Initialize();
    Window_128_7Obj->DoNew();
    Window_128_7Obj->Draw();
    return  cmdObj;
} // end of member function


CLCommand * Menu_129_2::DoItem2() {//Open
    CLCommand    *cmdObj = 0;

    EnableMenuItem(3);
    EnableMenuItem(4);
    EnableMenuItem(5);
    Window_128_7    *Window_128_7Obj = new Window_128_7;
    Window_128_7Obj->Initialize();
    Window_128_7Obj->DoOpen();
    Window_128_7Obj->Draw();
    return  cmdObj;
} // end of member function


CLCommand * Menu_129_2::DoItem3() {//Close
    CLCommand    *cmdObj = 0;

    Window_128_7    *Window_128_7Obj;

    Window_128_7Obj = (Window_128_7 *) gApplication-
>GetWindowByName("Window_128_7");
```

```
   if (!Window_128_7Obj) return 0;
   Window_128_7Obj->DoClose();
   //by hand start
   Window_128_7Obj = (Window_128_7 *) gApplication-
>GetWindowByName("Window_128_7");
   if (!Window_128_7Obj) {
      DisableMenuItem(3);
      DisableMenuItem(4);
      DisableMenuItem(5);
   }
   //by hand end
   return  cmdObj;
} // end of member function


CLCommand * Menu_129_2::DoItem4() {//Save
   CLCommand   *cmdObj = 0;

   Window_128_7   *Window_128_7Obj;

   Window_128_7Obj = (Window_128_7 *) gApplication-
>GetWindowByName("Window_128_7");
   if (!Window_128_7Obj) return 0;
   Window_128_7Obj->DoSave();
   return  cmdObj;
} // end of member function

CLCommand * Menu_129_2::DoItem5() {//SaveAs
   CLCommand   *cmdObj = 0;

   Window_128_7   *Window_128_7Obj;

   Window_128_7Obj = (Window_128_7 *) gApplication-
>GetWindowByName("Window_128_7");
   if (!Window_128_7Obj) return 0;
   Window_128_7Obj->DoSaveAs();
   EnableMenuItem(4);
   return  cmdObj;
} // end of member function

CLCommand * Menu_129_2::DoItem6() {//Quit
   CLCommand   *cmdObj = 0;

   gApplication->Terminate();
   return  cmdObj;
} // end of member function


CLCommand * Menu_130_3::DoMenuCommand(short pMenuItem) {
// Edit
   switch(pMenuItem) {
   case 1:
      return DoItem1();
   case 2:
      return DoItem2();
   case 3:
      return DoItem3();
```

```
      case 4:
         return DoItem4();
      case 5:
         return DoItem5();
      case 6:
         return DoItem6();
      } // end of switch(pMenuItem)
} // end of DoMenuCommand(pMenuItem)


CLCommand * Menu_130_3::DoItem1() {//Undo
   CLCommand    *cmdObj = 0;

   Window_128_7    *Window_128_7Obj;

   Window_128_7Obj = (Window_128_7 *) gApplication-
>GetWindowByName("Window_128_7");
   if (!Window_128_7Obj) return 0;
   CLGraphicsView *aView = (CLGraphicsView *) Window_128_7Obj-
>GetViewByName("CLGraphicsView");
   aView->Undo();
   return  cmdObj;
} // end of member function


CLCommand * Menu_130_3::DoItem2() {//Redo
   CLCommand    *cmdObj = 0;

   Window_128_7    *Window_128_7Obj;

   Window_128_7Obj = (Window_128_7 *) gApplication-
>GetWindowByName("Window_128_7");
   if (!Window_128_7Obj) return 0;
   CLGraphicsView *aView = (CLGraphicsView *) Window_128_7Obj-
>GetViewByName("CLGraphicsView");
   aView->Redo();
   return  cmdObj;
} // end of member function


CLCommand * Menu_130_3::DoItem3() {//Cut
   CLCommand    *cmdObj = 0;

   Window_128_7    *Window_128_7Obj;

   Window_128_7Obj = (Window_128_7 *) gApplication-
>GetWindowByName("Window_128_7");
   if (!Window_128_7Obj) return 0;
   EnableMenuItem(5);
   CLGraphicsView *aView = (CLGraphicsView *) Window_128_7Obj-
>GetViewByName("CLGraphicsView");
   aView->DoCut();
   return  cmdObj;
} // end of member function


CLCommand * Menu_130_3::DoItem4() {//Copy
```

```
    CLCommand    *cmdObj = 0;

    Window_128_7    *Window_128_7Obj;

    Window_128_7Obj = (Window_128_7 *) gApplication-
>GetWindowByName("Window_128_7");
    if (!Window_128_7Obj) return 0;
    EnableMenuItem(5);
    CLGraphicsView *aView = (CLGraphicsView *)Window_128_7Obj-
>GetViewByName("CLGraphicsView");
    aView->DoCopy();
    return  cmdObj;
} // end of member function


CLCommand * Menu_130_3::DoItem5() {//Paste
    CLCommand    *cmdObj = 0;

    Window_128_7    *Window_128_7Obj;

    Window_128_7Obj = (Window_128_7 *) gApplication-
>GetWindowByName("Window_128_7");
    if (!Window_128_7Obj) return 0;
    CLGraphicsView *aView = (CLGraphicsView *)Window_128_7Obj-
>GetViewByName("CLGraphicsView");
    aView->DoPaste();
    return  cmdObj;
} // end of member function


CLCommand * Menu_130_3::DoItem6() {//Select All
    CLCommand    *cmdObj = 0;

    Window_128_7    *Window_128_7Obj;

    Window_128_7Obj = (Window_128_7 *) gApplication-
>GetWindowByName("Window_128_7");
    if (!Window_128_7Obj) return 0;
    EnableMenuItem(3);
    EnableMenuItem(4);
    CLGraphicsView *aView = (CLGraphicsView *)Window_128_7Obj-
>GetViewByName("CLGraphicsView");
    aView->DoSelectAll();
    return  cmdObj;
} // end of member function


CLCommand * Menu_131_4::DoMenuCommand(short pMenuItem) {
    switch(pMenuItem) {
    case 1:
        return DoItem1();
    case 2:
        return DoItem2();
    case 3:
        return DoItem3();
    } // end of switch(pMenuItem)
} // end of DoMenuCommand(pMenuItem)
```

```
CLCommand * Menu_131_4::DoItem1() {//Patern=Black
   CLCommand   *cmdObj = 0;

   Window_128_7   *Window_128_7Obj;

   Window_128_7Obj = (Window_128_7 *) gApplication-
>GetWindowByName("Window_128_7");
   if (!Window_128_7Obj) return 0;
   CheckOnlyItem(1);
   CLGraphicsView *aView = (CLGraphicsView *)Window_128_7Obj-
>GetViewByName("CLGraphicsView");
   aView->SetPattern(qd.black);
   return  cmdObj;
} // end of member function


CLCommand * Menu_131_4::DoItem2() {//Pattern=Gray
   CLCommand   *cmdObj = 0;

   Window_128_7   *Window_128_7Obj;

   Window_128_7Obj = (Window_128_7 *) gApplication-
>GetWindowByName("Window_128_7");
   if (!Window_128_7Obj) return 0;
   CheckOnlyItem(2);
   CLGraphicsView *aView = (CLGraphicsView *)Window_128_7Obj-
>GetViewByName("CLGraphicsView");
   aView->SetPattern(qd.gray);
   return  cmdObj;
} // end of member function


CLCommand * Menu_131_4::DoItem3() {//Pattern=White
   CLCommand   *cmdObj = 0;

   Window_128_7   *Window_128_7Obj;

   Window_128_7Obj = (Window_128_7 *) gApplication-
>GetWindowByName("Window_128_7");
   if (!Window_128_7Obj) return 0;
   CheckOnlyItem(3);
   CLGraphicsView *aView = (CLGraphicsView *)Window_128_7Obj-
>GetViewByName("CLGraphicsView");
   aView->SetPattern(qd.white);
   return  cmdObj;
} // end of member function


// end of file myMenu.cp

//
// myWindow.h
//
#include "clpalette.h"
```

```
#include "clwindow.h"

#ifndef MYWINDOW_H
#define MYWINDOW_H

class Window_128_7 : public CLWindow {
public:
   Window_128_7():(128) { SetName("Window_128_7"); }
   class CLDocument * CreateDocument();
}; // end of class Window_128_7


class Palette_128_5 : public CLPalette {
public:
   Palette_128_5():(128) { SetName("Palette_128_5"); }
   class CLCommand * DoMouseCommand(short pItemHit);
   class CLCommand * DoItem1();
   class CLCommand * DoItem2();
   class CLCommand * DoItem3();
   class CLCommand * DoItem4();
   class CLCommand * DoItem5();
}; // end of class Palette_128_5


#endif MYWINDOW_H

//
// myWindow.cp
//
#include "myWindow.h"
#include "myApplication.h"
#include "clGraphicsView.h"

#pragma segment myWindow

class CLDocument * Window_128_7::CreateDocument() {
   return new CLGraphicsDocument(this,'MNDR','MNDR');
} // end of CreateDocument()


CLCommand * Palette_128_5::DoMouseCommand(short pItemHit) {
   switch(pItemHit) {
   case 1:
      return DoItem1();
   case 2:
      return DoItem2();
   case 3:
      return DoItem3();
   case 4:
      return DoItem4();
   case 5:
      return DoItem5();
   } // end of switch(pItemHit)
} // end of DoMouseCommand(pItemHit)
```

```
CLCommand * Palette_128_5::DoItem1() {//Selection Tool
   CLCommand    *cmdObj = 0;

   Window_128_7    *Window_128_7Obj;

   Window_128_7Obj = (Window_128_7 *) gApplication-
>GetWindowByName("Window_128_7");
   if (!Window_128_7Obj) return 0;
   HilightItem(1);
   CLGraphicsView *aView = (CLGraphicsView *) Window_128_7Obj-
>GetViewByName("CLGraphicsView");
   aView->CLSetCurrentShapeTool(selectionTool);
   return  cmdObj;
} // end of member function


CLCommand * Palette_128_5::DoItem2() {//Rectangle
   CLCommand    *cmdObj = 0;

   Window_128_7    *Window_128_7Obj;

   Window_128_7Obj = (Window_128_7 *) gApplication-
>GetWindowByName("Window_128_7");
   if (!Window_128_7Obj) return 0;
   HilightItem(2);
   CLGraphicsView *aView = (CLGraphicsView *) Window_128_7Obj-
>GetViewByName("CLGraphicsView");
   aView->CLSetCurrentShapeTool(Rectangle);
   return  cmdObj;
} // end of member function


CLCommand * Palette_128_5::DoItem3() {//RoundRect
   CLCommand    *cmdObj = 0;

   Window_128_7    *Window_128_7Obj;

   Window_128_7Obj = (Window_128_7 *) gApplication-
>GetWindowByName("Window_128_7");
   if (!Window_128_7Obj) return 0;
   HilightItem(3);
   CLGraphicsView *aView = (CLGraphicsView *)Window_128_7Obj-
>GetViewByName("CLGraphicsView");
   aView->CLSetCurrentShapeTool(RoundRect);
   return  cmdObj;
} // end of member function


CLCommand * Palette_128_5::DoItem4() {//Oval
   CLCommand    *cmdObj = 0;

   Window_128_7    *Window_128_7Obj;

   Window_128_7Obj = (Window_128_7 *) gApplication-
>GetWindowByName("Window_128_7");
   if (!Window_128_7Obj) return 0;
   HilightItem(4);
```

```
   CLGraphicsView *aView = (CLGraphicsView *)Window_128_7Obj-
>GetViewByName("CLGraphicsView");
   aView->CLSetCurrentShapeTool(Oval);
   return  cmdObj;
} // end of member function


CLCommand * Palette_128_5::DoItem5() {//Line
   CLCommand   *cmdObj = 0;

   Window_128_7   *Window_128_7Obj;

   Window_128_7Obj = (Window_128_7 *) gApplication-
>GetWindowByName("Window_128_7");
   if (!Window_128_7Obj) return 0;
   HilightItem(5);
   CLGraphicsView *aView = (CLGraphicsView *)Window_128_7Obj-
>GetViewByName("CLGraphicsView");
   aView->CLSetCurrentShapeTool(Line);
   return  cmdObj;
} // end of member function


// end of file myWindow.cp

//
// main.cp
//
#include "myApplication.h"

MyApplication *theApplication;

main() {
   theApplication = new MyApplication;
   theApplication->Run();
}
```

8. References

[Apple 85] Apple Computer. *Inside Macintosh,* Volume I, Addison-Wesley, Reading, MA, 1985. Chapter 13, The Dialog Manager, pp. 397-403.

[Budd 90] Budd, Timothy. *An introduction to object-Oriented programming.* Addison-Wesley, Reading, MA, 1990.

[Keh 91] Keh, Huan-Chao. "Comprehensive Support for Developing Graphical, Highly Interactive User Interface Systems," Ph.D Dissertation, Dept. of Computer Science, Oregon State University, Corvallis, OR, 1991.

[Keh 90] Keh, Huan-Chao. Lewis, T.G. "Direct-Manipulation User Interface Modeling With High-Level Petri Nets," Tech. Report 90-60-17, Dept. of Computer Science, Oregon State University, Corvallis, OR, 1990.

[Lai 91] Lai, Chai. "Adding Object-Oriented Structured Graphics and Graphics Building Block to the MVC Paradigm Application Framework," Masters Paper, 91-60-16, Dept. of Computer Science, Oregon State University, Corvallis, OR, 1991.

[Lewis 90] Lewis, T.G. *CASE: Computer-Aided Software Engineering*, Van Nostrand Reinhold, New York, NY, 1990.

[Lewis 89] Lewis, T.G., Handloser, F.T., Bose, S., and Yang, S. "Prototypes from Standard User Interface Management Systems," *IEEE Computer*, May 1989.

[Li 91] Li, Tong. "OSU v3.0 Browser: Window into GUI Applications," Masters Paper, 91-60-6, Dept. of Computer Science, Oregon State University, Corvallis, OR, 1991.

[Luo 91] Luo, Chung Cheng, and Lewis , T.G. " Oregon Speedcode Universe 3.0 Programming Manual," Project Report, 91-60-31, Dept. of Computer Science, Oregon State University, Corvallis, OR, 1991.

[Myer 90] Myers, B.A. et al. Garnet-Comprehensive Support for Graphical, Highly Interactive User Interfaces, *IEEE Computer* 23, 11 (Nov. 1990), 71-85.

[Peter 81] Peterson, J.L. *Petri Net Theory and the Modeling of Systems.* Prentice-Hall, Englewood Cliffs, NJ, 1981.

[Roger 87] Roger S. Pressman *Software Engineering A Practitioner's Approach.* MaGraw-Hill Book Co., 1987

[Wilson 90] Wilson, David A. Rosenstein, Larry S. and Shafer, Dan. *Macintosh Inside Out Programming with MacApp*, Addison-Wesley, Reading, MA, 1990.

[Yang 89] Yang, Sherry. "OSU: A High Speed Software Development Environment," Tech. Report 89-60-21, Dept. of Computer Science, Oregon State University, Corvallis, OR, 1989.