Copyright Warning & Restrictions

The copyright law of the United States (Title 17, United States Code) governs the making of photocopies or other reproductions of copyrighted material.

Under certain conditions specified in the law, libraries and archives are authorized to furnish a photocopy or other reproduction. One of these specified conditions is that the photocopy or reproduction is not to be "used for any purpose other than private study, scholarship, or research." If a, user makes a request for, or later uses, a photocopy or reproduction for purposes in excess of "fair use" that user may be liable for copyright infringement,

This institution reserves the right to refuse to accept a copying order if, in its judgment, fulfillment of the order would involve violation of copyright law.

Please Note: The author retains the copyright while the New Jersey Institute of Technology reserves the right to distribute this thesis or dissertation

Printing note: If you do not wish to print this page, then select "Pages from: first page # to: last page #" on the print dialog screen

ABSTRACT

A Petri Net Toolkit for Parallel Program Debugging

by

Potla Kishore Reddy

An effective debugger must support the language and operating system resource abstractions that are available to the programmer. Earlier debuggers worked at the machine architecture level: they dealt with machine instructions and registers. Current debuggers, designed for single process debugging, permit access to program variables and breakpoints and single-stepping at the level of high-level language statements. Eventhough the current debuggers are already implemented to be a powerful tool, they still cannot do a job of parallel debugger.

In this thesis, a computer simulation system has been established by Petri Nets execution providing a convenient and friendly interface as it allows the user to do parallel program debugging.

The Parallel Debugger is simulated by providing a time parameter for each transition and thus simulating the net performance. Hitherto, this time parameter can either be constant or exponentially distributed.

A PETRI NET TOOLKIT FOR PARALLEL PROGRAM DEBUGGING

by

Potla Kishore Reddy

A Thesis
Submitted to the Faculty of
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Master of Science

Department of Computer and Information Science

January, 1993

APPROVAL PAGE

A Petri Net Toolkit
for Parallel Program Debugging

Potla Kishore Reddy

Dr. Daniel Y. Chao, Thesis Adviser
Assistant Professor of Computer and Information Science
NJIT

Dr. Mary M. Eshaghian, Committee Member
Assistant Professor of Computer and Information Science

Dr. David Wang, Committee Member
Assistant Professor of Computer and Information Science
NJIT

BIOGRAPHICAL SKETCH

Author: Potla Kishore Reddy

Degree: Master of Science in Computer and Information Science

Date: January, 1993

Undergraduate and Graduate Education:

- Master of Science in Computer and Information Science,
  New Jersey Institute of Technology, Newark, NJ, 1993
- Bachelor of Science in Mechanical Engineering,
  College of Engineering, Jawaharlal Nehru Technological University,
  Hyderabad, AP, India, 1990

Major: Computer and Information Science

This thesis is dedicated to
my beloved mother

without whom I would not have been
at this stage of success

ACKNOWLEDGEMENT

The author wishes to express his gratitude to his thesis adviser, Dr. Daniel Y. Chao, for his guidance, friendship, and moral support througout this research.

Special thanks to Professors Dr. Mary Eshaghian and Dr. David Wang for serving as members of the committee.

The author is very greatful to the moral support, and suggestions from

Professor in Electrical Engineering, JNTU, Louisiana State University, and IBM, Austin, Texas.

And Finally, a respectful gratitude for my mother Late who passed away on January 17, 1992, who is the inspiration and reason for my success in academic and daily life.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

In the process of developing large-scale parallel programs, one of the most serious problems, we come across is the lack of proper debugging and performance analysis tools. Generally, Debugging Parallel Programs is more challenging than debugging sequential programs. As parallel architectures are more complex, management and controlling of data will be tedious.

Parallel debugging involves the following activities:

i) Executing the program on a parallel architecture or a simulator.

ii) Gathering data from the program.

iii) Processing the data.

iv) Controlling the program's execution.

v) Modifying the programs state.

Each one of the above tasks are complex and they interact significantly. For an efficient Parallel Debugger, each of these activities must be designed in the context of the others.

Parallel programs are difficult to debug because they run for a long time and two executions may yield different results for the same input. These problems can be solved by "Reverse Execution", which is a simple and powerful concept (Pan and Linton 1990). The concept of reverse execution is a simple way to address the difficulties of debugging parallel programs and matches what programmers currently do with repeated execution.

Debugging can be defined as an iterative process of searching for a program error that can cause false behavior of the program execution. Traditionally this iteration is accomplished by setting a breakpoint and repeating execution up to the breakpoint. The break point can be represented as a condition on the execution state, e.g., when a certain statement is reached or when a variable has certain value in the memory in a specific

scope. Each new breakpoint and corresponding execution bring the programmer closer to the bug (error) location in the program. This approach is suitable for short-running sequential programs.

For parallel programs, however, repeating execution is unacceptable for two reasons: First, parallel programs, run for a long time. As parallel programs development take more time and effort that sequential programs, the motivation to write or extract parallelism is to reduce execution time by a significant amount. Otherwise, it's wise to implement the program sequentially.

Second reason is that parallel programs may be non-deterministic; meaning, two runs of a program with identical input may not execute the same code. Generally, non-determinism is often the result of a bug where the programmer allows relative speeds of processes to affect the computation. This kind of bug, called a "race condition", is one of the most difficult kinds of errors to track down.

Parallel programs differ from sequential programs primarily in that the temporal relationships between events are only partially defined. However, for a given distributed computation, debugging utilities typically linearize the observed set of events into a total ordering, thus losing information and allowing potentially capturable temporal errors to escape detection (Fidge 1990).

An effective debugger must support the language and operating system resource abstractions that are available to the programmer. Earlier debuggers worked at the machine architecture level: they dealt with machine instructions and registers.

Current debuggers, designed for single process debugging, permit access to program variables and breakpoints and single-stepping at the level of high-level language statements. But even the current debuggers are already implemented to be a powerful tool, they still can not do a job of parallel debugger. The existing parallel debuggers are complex and the debugging process is tedious as discussed before. To simplify the task of parallel debugging, in this thesis I try to establish a computer simulation system by Petri

Nets execution providing a convenient interface to allow the user to do a so called parallel debugger.

Ideally, a debugger should allow the programmer to "scroll" forward and backward through an execution the same way one can scroll through an execution the same way one can scroll through a text file (Pan and Linton 1990). The programmer should be able to visualize the program execution graphically and should be able to examine the values of local and global data at certain point in the execution state. To attain this state of visualizing the parallel program execution and debugging, we have implemented a graphical toolkit using Petri Net execution and simulating the parallel execution and debugging.

CHAPTER 2

INTRODUCTION TO PETRI NETS

Petri Nets are abstract virtual machines, and are formal graph models used for representing the flow of information and control in system, especially those which exhibit non-deterministic, asynchronous and concurrent properties. Their properties are quite natural and easy to understand and their capability of modeling and analyzing such systems are very powerful. There are many programs which have been developed in order to implement the system, which the Petri Net System has modeled.

The Petri Net System has modeled many applications in many fields. The mainstream of applications includes the embedded system, logic programming, hardware design, parallel processing, knowledge representation, simulation language, space station activities, manufacturing, communications and so on.

But, until recently, Petri Nets could only be represented by data and not by graphics. Therefore, this computer simulation system provides the user with revolutionary X-Window Graphic Image System. So now the user can implement the Petri Net theory by running in the X-Window System using our interface. So the Petri Nets are no longer be abstract in nature.

Petri Net firing rules:

1. A Transition (represented by a line) is said to be enabled iff each of its input places (represented by circles) has a token (represented by a dot) in it.
2. A Transition can fire only if its enabled and there is no collision.

3. When a Transition fires, the new marking is obtained by moving tokens from each of its input places and adding tokens to each of its output places. The number of tokens removed (added) are equal to the relevant number of arcs between the input or output place and the transition.

CHAPTER 3

PARALLEL PROGRAM ANALYSIS

Parallel program analysis is tedious, and more difficult than the sequential program analysis, as we have to concentrate on the interactions between processes, instead of focusing on the internal state of individual processes. Symbolic debuggers, which are used to analyze single individual processes are not sufficient to analyze interprocess relationships, including communication, deadlock, and resource contention. So we have to augment the facilities to symbolic debuggers, to perform the analysis of above activities.

Animation techniques that are used for visualizing the effects of procedures or statements of sequential programs can be extended to illustrate interactions between processes over time. These tools and techniques must be integrated into an environment which supports extensive parallelism during computation, but provides a single user-interface during the analysis.

The analysis methodology employed for sequential programs can be used for parallel programs also even though parallel program analysis is different and more complex than that of sequential programs. The essential characteristics of the traditional methodology for program debugging and performance analysis that can be extended to parallel program analysis are as follows (Fowler, Leblanc and Crummey 1990).

Program analysis should be interactive. Besides automation of the collection and presentation of execution data, the data interpretation must include some feedback to the programmer.

Analyses should be repeatable. There exists a classical technique of program debugging to re-execute the program, with more output statements at different points or instances, so that they give more details on program execution, which is vital information, during program debugging.

But this technique depends on the fact that most sequential programs are deterministic and ensure that successive executions are essentially same. Even though parallel programs do not possess this quality, any methodology for parallel program analysis requires that information gleaned during program analysis can be easily derived again

Program analysis is a top-down process. Analysis is too complicated to show the detailed viewpoint at all times. To manage this complexity, we need a tool with abstract view. As this may hide the relevant information, we should take care, that its possible to move from abstract views to concrete details, as the focus of interest is narrowed. It must be feasible to focus from an entire program to a single process within the program and then to a particular procedure within the process, and finally to a statement within a procedure.

As parallel program analysis in general, and performance analysis in particular, requires extensive analysis of temporal relationships, its wise to make those relationships explicit using a spatial dimension in the presentation of an execution. If we can provide an abstract view of an entire execution, as opposed to the abstract view of a single state of an execution provided by animation, which makes it possible to make a survey at a glance of communication patterns of a computation at each stage. In animation systems, a program is represented by a static structure, usually a regular communication structure, like Petri Nets, on which interesting abstract dynamic events can be superimposed over time. Temporal relations are difficult to analyze using animation as they are presented temporarily. The granularity of temporal relationships shown in a single frame of animation is limited because two events at the same location cannot be shown simultaneously.

Program Analysis tries to detect patterns in the program. Static (compile-time) program analysis can detect patterns in the program's description. Dynamic (run-time) program analysis detects patterns in the program's behavior over time. Monitoring and

Visualization display information regarding the execution of the program. Programmer can use this information to detect errors, understand the program layer by layer or its problem area at a specific point, or document a program or problem area. The difference between program visualization and program monitoring is abstract, but monitoring is usually restricted to read-only views of the program or system while visualization extends to interactive messages.

CHAPTER 4

HIGH-LEVEL DEBUGGING

Source level debugging, which is also called low-level debugging focuses on programming errors that can be generated by program statements of the source code. In contrast, high-level debugging where only more abstract terms, like complex abstractions applicable to large parallel programs are focussed.

Large scale parallel programs for which large-computing time is a must are becoming trivial in today's world of high-speed super computing. The development of these programs frequently predates the existence of adequate debugging and performance monitoring tools. Although the state of the art of parallel programming is still in its infancy, many useful parallel programs have already been developed.

Sequential programs can be "parallelized" by instantiating several threads of control that essentially execute the same main program (Ziya and Gertner 1990). Even though most of the code from the resultant parallel version still deals with the sequential case, there will be minor portion of the code that deals with the parallel behavior. Nevertheless, these minor alterations caused by parallel behavior are sufficient to cause profound changes in the behavior of the program.

With the aid of conventional tools like, symbolic debuggers, many functional errors can be detected. Parallel programs, however, introduce a new class of bugs that are related to their dynamic properties, which includes time dependencies such as starvation, race conditions, deadlocks, and more the subtle side-effects of synchronization such as "fairness". They also include errors which are not only bugs, but also which contribute to poor performance.

Debug statements can be viewed as an abstraction mechanism. Instead of viewing program execution as a trace of instructions or source language statements, it can be viewed as a trace of "key events" detected by complex logical assertions.

Parallel program development has to rely on debug statements, even though the method of inserting debug statements, remains very inflexible.

CHAPTER 5

RELATED WORK

The present debuggers in an UNIX environment. such as dbx and sdb aides the programmers with an ability to "single-step" the execution of a program at the assembler or source level language, to "break" or interrupt program execution at selected points, and to examine the values of program variables.

More relatively complex debuggers like cdb on UNIX and dbg on VAX/VMS support complex breakpoints that are conditioned on logical assertions about a given set of variables. In addition to that, now-a-days there are several debuggers are existing to support parallel programs, which needs complex debugging.

Even though the new features are emerging, unfortunately, the architectures of conventional debuggers have not changed to better support those new features. So, logical assertions and multi-process tracing are built on top of existing primitives such as breakpoints. In turn, these features depend on the system services and kernel invention to interact with the target. The overhead is turns out to be considerably high, which runs to the equivalent of thousands or tens of thousands of instructions for each conditional assertion. Despite, of these new features, conventional debuggers remain unsuitable for high-level debugging because of their intrusiveness.

In contrast, many debugging research projects focus on the development of debug tools within completely new programming environments which include sophisticated graphics-oriented user interfaces, new parallel languages, compilers, and program analysis tools (Ziya and Gertner 1990).

Often we find debugging tools as tightly coupled, with access only to binary objects and symbolic information but also to all of the abstractions employed at each stage of the development cycle.

Albeit, these debugging mechanisms are sometimes very powerful, they are closely intertwined with their specific programming environments making them less applicable to the programs developed outside that environment.

The main motivation to develop this Petri Net toolkit for Parallel Debuggers emerged from the idea to make the complex parallel program debugging easier by providing visual form of the parallel program execution.

CHAPTER 6

FEATURES OF THE SIMULATION PROJECT

At present very few, existing parallel debuggers can include all the excellent features of our debugger simulation toolkit. The user interfaces for the existing parallel debugging tools are usually complex and hard to comprehend, even when they are presented graphically. For others, the lack of practical graphical view function support render the manipulation and modifications of system clumsy. The difficulty in familiarizing with the tool is also a potential disadvantage. In this case, Petri net modeling, which we used in our simulation toolkit is superior to other debugging models due to the following reasons :

- the ability to show a precise and graphical representation;
- the availability of machine readable descriptions;
- the existence of analysis techniques for control aspects;
- the capability of employing top-down design methodology;
- the possibility of design and analysis automation;
- the ability to provide high level of user-friendly interface.

The synthesis of our Petri Net (PN) graphic tool is done through implementation of the two-party synthesis procedure proposed by Ramamoorthy and Dong in an X-Window environment together with the knitting technique introduced by Yaw. Our toolkit provides numerous drawing and picture-manipulating functions within multi-layered radio-buttons, dialog-boxes, pull-down menus and pop-up windows, all of which can easily be activated by a click of mouse. Utilities such as ZOOM IN/ZOOM OUT have also been implemented. The graphical user interface (GUI) is designed to be self-explanatory. There is also a HELP feature provided for all the functionalities of the toolkit. A brief description of drawing procedures and functions are given below .

The interactive tool allows high design flexibility The user can draw or create places, transitions, lines and arcs with arrows in both directions as in Figure 1. The user can also use the "Select" function under the "Draw" button to either select all or draw a box to select a particular set of objects. By clicking some menu buttons, the user can move, copy, or erase the selected objects.

The "Zoom" button allow the whole graph to be zoom in/out. The "Undo" button allows the user to undo the current action and return to selected objects. The "Modify" button, which is the most essential for debugging purposes, allows the user to change properties such as token numbers, variable values of a place as in Figure 2 and properties of a transition such as transition firing time and transition statement and the text fonts. Upon clicking the "Modify" at a point inside a transition, the tool pops up a window, asking the user to input the transition name, the statement, and the firing time as in Figure 3.

Note that for all numerical inputs, we use scale widgets to allow the user to select the desired and acceptable value rather than type it in manually as in Figure 4. The place is associated with the following data structure:

```
{
  int symbol_level;
  char symbol_name[10];
  char symbol_type[10];
  char symbol_token[10];
  char symbol_value[10];
}
```

This structure is used to save the data of a place which the user inputs by clicking on the "Modify" option and a point inside the place to pop up a window, asking the user to input the place name and the symbol token. Once the data has been input, the user clicks the "OK" button to continue or the "CANCEL" button to cancel the operation. The program then pops up a second window and the user will be asked to input the data type.

If the data type is an "int" or "char", the program will pop up another window for data input. Once the data has been entered, another window pops up requesting the user to input the place value.

Each transition is associated with the following data structure:

```
{
  char trans_name[10];
  char trans_statement[20];
  char trans_time[26];
}
```

The corresponding C program to simulate the data flow computation can be stored in a C source code file by clicking the "Saveprogram" button of the "Simulate" submenu of the "Analysis" menu. This file will be compiled into an object file for simulation, which is performed by clicking first the "Program" button of the "Simulate" submenu followed by clicking the "Auto" or "Step" or "Break" button.

After the simulation, the user clicks a point inside a place in a random access fashion to check its symbol value for debugging. A window will pop up as in Figure 5 to display the place symbol name, simulation time and the symbol value. The "Text" button allows text to be written at any location on the screen. The "Box", "Circle", and "Line" buttons allows the user to draw rectangle boxes, circles, and ellipci, and lines (can either be dashed or solid, controlled by the "Brush" menu). By clicking the "Print" button, the user can select a particular window to take a snapshot, send it to the printer, and obtain a hardcopy. The "File" button allows files (not just a single file) to be displayed, imported, copied, deleted and edited (using the Xedit). The design can be saved using the "Save" button. The environment is made even more user-friendly by providing various levels of help messages. As the knitting technique for Petri net synthesis can handle both interactions and internal operations, the tool can synthesize, modify, and debug protocols with both data and control flows. Therefore, one can design protocols with both message transitions and data operations. The tool can also simulate both compound statements and condition statements. We can also modify values of variables without recompilation

for new simulation to check the effects of these variables. This feature is extremely valuable when the compilation process is time-consuming. Furthermore, instead of simulating the whole net, one may just simulate a single transition with a statement. This is useful in the initial stage of the program development.

By providing a time parameter for each transition, we are able to simulate the net performance. Currently, this time parameter can either be constant or exponentially distributed. A time Petri net can be simulated in three modes: auto, step and break.

In auto mode, one iteration of transition firings will be performed to return to the initial marking. In step mode, a number of transitions will be fired according to the step number chosen by the user. In break mode, the user can select a number of break points (transitions). A user can click a mouse at a certain place to display the last instant that a token entered the current place, regardless of the mode selected. Note that one does not have to simulate the entire net; the user can just fire certain transitions by assigning tokens to their input places. This allows debugging and simulation at all design phases. A "Trace" button is also provided to record the transition firing events to aid debugging. It is a user-friendly interactive debugger since the user need not have any knowledge about existing debuggers.

As mentioned before, traditional debugging tools such as "dbx" or "sdb" for the UNIX systems check execution results at specified breakpoints against desired behavior. It is hard to visually monitor the control and data flows. Parallel programs involve interactions among physical or logical entities. Just as programming productivity is enhanced by programming each object in parallel, so is debugging effectiveness. By displaying several objects or entities simultaneously on a screen while hiding the details of each object, one can obtain a global picture of system behavior and establish breakpoints at every graphical object using a mouse. Multiple breakpoints, unlike serial programs, are natural for parallel programs. As soon as all breakpoints are specified, the execution of the debugger can be triggered interactively. Executions stop at breakpoints

and designers can check variable values by simply clicking specific graphical objects at the breakpoints. For many interactive debugging tool, this may be a problem. If the graphical representation of the program is too large to be fully displayed on the screen, it is not easy to correctly identify breakpoints. Hierarchical abstraction of Petri Nets becomes extremely useful here. At the highest abstraction, the program consists of objects interacting with each other.

The details of interactions and internal activities inside each object are hidden and it may require several levels of expansion for them to be explicit. To identify specific breakpoints, one has to go down to the lowest level. While looking at the highest level, one should be able to click the mouse at an object to get program statements at the lowest level.

In addition to checking variable values, another important aspect of debugging is to ensure that the flow of control or data follows the specification. Both global and local flows help debugging the parallel program. Verification of parallel programs consists of two phases: interaction and internal operation. To check interaction (or protocol) correctness, one can check the global flow. To check the correctness of internal operations, one can check the local flows of each object.

CHAPTER 7

DISCUSSION AND FUTURE WORK

Future work in the improvement of our Petri Net parallel debugger will focus on the hierarchical representation of modular design. This feature will allow a larger net system to be presented without the problem of being too complex for user to monitor. The idea is to let each transition to represent a module consisting of a set of transitions and places. The internal network structure can be displayed by clicking the mouse on the transition. Overlapping windows can greatly simplify the monitoring process. All in all, the improvement of the GUI is always a goal of our research.

CHAPTER 8

CONCLUSION

In this thesis an advanced approach has been presented on developing an interactive parallel debugging tool based on the theories of Petri net and its graphical interface. By taking advantage of the easy-to-follow graphic tool environment, we have succeeded in overcoming some of the problems present in traditional debuggers. We have also avoided some of the deficiency in other parallel debugging tools.

There are several unique features in our debugging tool:

• Precise and User-friendly Interface : The debugging process becomes transparent to the user when our tool is employed. By choosing any one of the simulation modes (auto, step, and break), the user can actually witness the progress of the program on the screen. This is especially useful when the user is designing a parallel, concurrent program such as communication protocol. An elusive error in protocol design becomes easily detectable when debugged interactively.

• Flexible Debugging Process : Since the debugging tool is Petri net based, it allows simulation and debugging during all design phases. Debugging can be done during the development stage of the parallel system. The user need not wait until the whole network is completed to detect an error in an early stage.

• Higher Productivity : As mentioned before, the debugging tool avoided the large overhead of recompiling programs during debugging. Simple manipulation of objects together with specific modifications of values of variables can accomplish the same effect as recompiling. Executions become spontaneous and precise. The simplification of debugging process will eventually greatly enhance the overall productivity of the design project.

• Versatile Applicability : Petri net system is known to be used as an abstract and formal graphic model to represent the flow of data as well as control information. Our

debugger tool, therefore, not only can be used in communication protocol design but is also applicable to any non-deterministic, asynchronous, and concurrent programs.

• Easy to set up, simulate, and modify : Since the entire network is graphically displayed, it is very easy for the user to set up initial states simply by drawing objects and inserting tokens in them. Modification is equally easy and requires no special procedures.

APPENDIX

```
#include "petri-net.h"
#include <Xm/ScrollBar.h>
#include <Xm/PushB.h>
#include <Xm/Label.h>
#include <Xm/DialogS.h>
#include <sys/stat.h>
#include "place.bitmap"
#include "trans.bitmap"
#include "h_trans.bitmap"
#include "arrow.bitmap"
#include "extplace.bitmap"
extern int pos_num;
extern float zoomflag;
extern char rec_string[];
extern int t[];
extern int rec;
extern int xx[], yy[];
extern int my_delay;
extern struct timeq{
        int trans;
        float time;
        struct timeq *next;
        };
extern struct timeq *deq();
extern void Auto();
```

```
extern void Step(), Brk();
extern int dis_fg;
extern int traceflag, expflag;

/****************************************
        Simulate
 ****************************************/

void
Simulate(w,data,client_data)
Widgetw;
graphics_data *data;
caddr_t     client_data;
{

    FILE *ft,*ffp;
    /*  initial state pointer */
    struct  place   *place_p;
    struct  trans   *trans_p;
    struct  trans_e *trans_t;
    struct  state   *state_p;
    /*struct  state_e *state_i;*/
    Window win=XtWindow(canvas);
    Window        root,child;
    int           root_x,root_y,win_x,win_y;
    int    flag = 1;
    char   *dstrb="DETERMINISTIC";
```

```
unsigned int mask;
Display *dpy=XtDisplay(canvas);
int test[30];
int b,m,n,d,k,g,j,q,f=0;
int i;/*transition which fires*/
float timemax,tt,timemin,timemax1;
/*current state pointer */
struct  state_e  *state_c;
struct  arc     *arc_p;
/*struct  trans_e *cps2net();*/
int     amark[PMAX];
int     mark_p[PMAX];
struct placelist *list_p;
struct placelist *list_p1;
struct placelist *list_p2;
struct translist *list_t;
struct timeq *head,*first;
struct tokenlist *list_to, *list_tok;
int seed,dummy;
/*****************variables for weight**********************/
int found=0,        /*flag for weight found between lo-hi values*/
   poli=0,/*more than one transition able to fire*/
   *st_list,        /*array of firable transitions at the same time*/
   nf=0,   /*number of firable transition put in queue*/
   li,              /*counter for firable tr. at the same time*/
   lt,lk,    /*counter for fir. trans. from the same place*/
   tm,              /*firable time to be compared*/
```

```
   ti;              /*chosen random transition from array st_list*/
float num,          /*random float between 0-1*/
    lo,hi;   /*low and high boundary to search weight*/
/*structure of a particular place */
struct outplace{
   int xtion;/*output transition*/
   int bufno;/*output buffer nubmer*/
}*sp_list;
char *SPID;
int PID;
struct stat statbuf1,statbuf2;

/***********digital***************/
if(digital_flg == 1)
{
  if(data->current_func == Auto)
    digital_sim(w, data, client_data) ;
  return;
}

/*********** animation ************/
if(dis_fg == 1){
   XClearArea(dpy, win, xx[0], yy[0]-20, 25, 20, 0);
   rec=0;
   sprintf(rec_string,"%d",rec);
   /* reset record number to zero */
   XDrawString(dpy, win, data->gc, xx[0],yy[0],rec_string,strlen(rec_string));
```

```
   /* if in step mode input time unit */

}
/*Display_Scale();*/
/********** animation ***********/
trans_p = &(local.trans_t);
place_p = &(local.place_t);
state_p = &(stg1.statename);
arc_p   = &(stg1.arcname);
state_p->number = 0;
arc_p->number   = 0;
printf("*****************************************Into Simulate...\n");

/* program begin */

/*XtAddEventHandler(canvas, ButtonPressMask, FALSE, findobject, data); */
/*SET THE NUMBER of simulate steps  */

/*srand(24);*/
/*tt=random()/(1024*1024*1024*2);*/

/*bp*/
if(expflag){
  /*seed = PromptBox("Seed    Number    ?",win_x-60,win_y-40);*/
data->wi=2;
  dummy=create_delay_dialog(w,data);
  seed = data->p;
```

```
  printf("seed = %d\n",seed);
  srand(seed);
  strcpy(dstrb,"EXPONENTIAL");
}
/*bpend*/

if(traceflag){
 if((ft = fopen(str_t,"w"))==NULL){
  printf("can't open trace file !\n");
  return;
 }
 printf("Open trace file successfully\n");
 fprintf(ft,"DISTRIBUTION IS %s\n",dstrb);
 if(expflag)fprintf(ft,"Seed Number = %d\n",seed);
}
if(         data->current_func == Step)
{
  /*XQueryPointer(dpy, RootWindow(dpy, DefaultScreen(dpy)), &root,
     &child, &root_x, &root_y, &win_x, &win_y, &mask);
  d = PromptBox("Number of steps ?", win_x-60, win_y-40);*/
     data->wi=1;
     dummy=create_delay_dialog(w,data);
     d=data->kk;
}
else
  d=10000;
/*for (i=0;i<TRANS.number;i++)
```

```
   TRANS.entry[i].brk=0;*/
if(          data->current_func == Brk)
   data->current_func = Auto;
printf("d %d \n", d);
for (i=0;i < PMAX ;i++) {
   mark_p[i] = place_p->entry[i].token_e.number;
   if(i<2) printf("marking.p = %d\n",mark_p[i]);
   TRANS.entry[i].avgtime = TRANS.entry[i].time;
}
for (i = 0;i <PMAX;i++)
   amark[i] = mark_p[i];
head = (struct timeq *)malloc(sizeof(struct timeq));
head->next = NULL;

/*  LU add  */
if (dis_fg ==1 ) {
  refresh(canvas,data);
  /* refresh1(canvas,client_data, client_data); */
}

st_list=(int *)malloc(sizeof(int));
sp_list=(struct outplace *)malloc(sizeof(struct outplace));
for (i=0;i<TRANS.number;i++){
   TRANS.entry[i].exec=0;
   TRANS.entry[i].flag=0;
}
nf=0; /* counter for firable transitions at each step */
```

```
/* get the initial firable transition */
while((trans_t = cps2net(trans_p,place_p,amark,data)) !=NULL) {
  if (data->buffer[TRANS.entry[i].bufferno].erase==0 && TRANS.entry[data->X-
1].brk==0)
     TRANS.entry[data->X-1].exec=1;
  printf("****TRANS.entry[%d].exec=1\n",data->X-1);
  if(TRANS.entry[data->X-1].flag == 0){
    if(expflag)
   TRANS.entry[data->X-1].time=gen_arr(TRANS.entry[data->X-1].avgtime);
    else TRANS.entry[data->X-1].time=TRANS.entry[data->X-1].avgtime;
   TRANS.entry[data->X-1].flag=1;
  }
  printf("time[%d]= %f\n",data->X-1,TRANS.entry[data->X-1].time);
  list_p = TRANS.entry[data->X-1].i_place;
  timemax =0;
  timemax = place_p->entry[list_p->p_name].token_e.time;
  /*  differ  */
  if (fuzzyflag)
    timemax1 = place_p->entry[list_p->p_name].symbol_value;

  while (list_p != NULL ) {
   if ( place_p->entry[list_p->p_name].token_e.time > timemax)
   timemax = place_p->entry[list_p->p_name].token_e.time;
   /*  differ  */
   if (fuzzyflag) {
     if ( place_p->entry[list_p->p_name].symbol_value < timemax1)
     timemax1 = place_p->entry[list_p->p_name].symbol_value;
```

```
    }

    list_p=list_p->next;
  }
  TRANS.entry[data->X-1].enabletime=timemax;
  insert(head,data->X-1,data,timemax);
  nf++; /*increment counter for number of firable tr.*/
}
/*for (j=0;j<d;j++) {*/
j=0;
/*STEP LOOP: loop while step<d and queue not empty*/
/*while ((j<d)&&((first=deq(head))!=NULL)){*/
while ((j<d)){

  /* Catch mouse button press event to quit simulation */
  g = 0;
  while(XtPending() && g == 0){
        XEvent event;
        XtNextEvent(&event);
        /*printf("event type %d\n",event.type);*/
        if(event.type == ButtonPress){
              j = 50000;
              g = 1;
        }
        j++;
  }
```

```
        /* set marking to new marking area */

        printf("j %d \n", j);
        printf("number of firable trans %d\n",nf);

        /*
        for (i=0;i<TRANS.number;i++){
                TRANS.entry[i].exec=0;
        }
        while((first=deq(head))!=NULL);
      head = (struct timeq *)malloc(sizeof(struct timeq));
      head->next = NULL;
        while((trans_t = cps2net(trans_p,place_p,amark,data))
          !=NULL) {
                if (data->buffer[TRANS.entry[i].bufferno].erase==0 &&
TRANS.entry[data->X-1].brk==0)
                        TRANS.entry[data->X-1].exec=1;
                printf("****TRANS.entry[%d].exec=1\n",data->X-1);
                if(TRANS.entry[data->X-1].flag == 0){
                        if(expflag)TRANS.entry[data->X-
1].time=gen_arr(TRANS.entry[data->X-1].avgtime);
                        else TRANS.entry[data->X-1].time=TRANS.entry[data->X-
1].avgtime;
                        TRANS.entry[data->X-1].flag=1;
                }
                printf("time[%d]= %f\n",data->X-1,TRANS.entry[data->X-1].time);
                        list_p = TRANS.entry[data->X-1].i_place;
```

```
                    timemax =0;

                    timemax = place_p->entry[list_p->p_name].token_e.time;

                    while (list_p != NULL ) {
                        if ( place_p->entry[list_p->p_name].token_e.time >
timemax)
                            timemax = place_p->entry[list_p-
>p_name].token_e.time;
                        list_p=list_p->next;
                    }
                TRANS.entry[data->X-1].enabletime=timemax;
                insert(head,data->X-1,data,timemax);
            }
        */
        if((first=deq(head))!=NULL) {
            timemin=first->time;
          if (dis_fg ==1 ) {
            i= first->trans;
            /* if product is finished */
            if(i == 0) {
              /* clear present record number */
              XClearArea(dpy, win, xx[0], yy[0]-20, 25, 20, 0);
              /* increment record number and print it out */
              rec++;
              sprintf(rec_string,"%d",rec);
              XDrawString(dpy,win, data->gc,xx[0],yy[0],rec_string,strlen(rec_string));
```

```
            }
          }
        }
         else
         {
           onlyflag=0;
        if(traceflag){
              fclose(ft);
              traceflag=0;
        }
          return;
         }
      m=0;
    /*tt=((float)random())/(1024*1024*1024);
    printf("tt %f",tt );*/
      /*put first firable to the array*/
      st_list[0]=first->trans;
      nf--;
      tm=st_list[0];/*for comparison with other xtions*/
      li=1;
      /*put all firable transitions that will fire at the same time,
      if any,  in array st_list*/
      while(((first=deq(head))!=NULL)&&
            (TRANS.entry[first->trans].enabletime+TRANS.entry[first-
>trans].time==
          timemin + TRANS.entry[tm].time)){
            st_list[li]=first->trans;
```

```
            printf("tr = %d\n",st_list[li]);
            nf--;
            li++;
        }
        /*for(g=0;g<TRANS.number;g++){
            i=(int)(g+(tt+1.5)*TRANS.number)%TRANS.number;
            printf("i %d \n", i);
            printf("g %d \n",g);
            printf(" erase %d \n", data->buffer[TRANS.entry[i].bufferno].erase);
            if(data->buffer[TRANS.entry[i].bufferno].erase==0 &&
                TRANS.entry[i].exec==1 &&
                TRANS.entry[i].enabletime+TRANS.entry[i].time==timemin +
TRANS.entry[first->trans].time &&
                TRANS.entry[i].flag==1 ){
                m=1;
                TRANS.entry[i].flag=0;
                if((first=deq(head))!=NULL)
                 timemin=first->time;
        */
        /*if last transition's time not equal to transiton tm
          put it back to queue*/
        if(first!=NULL){
           insert(head,first->trans,data,first->time);
        }

        /*if more than one will fire at the same time,
          pick one randomly (=ti)*/
```

```
if(li>1){
        ti=st_list[rn_choice(li)-1];
        /*check if more transitions come from
        the same place transition ti comes from*/
        lt=0;
        poli=FALSE;
        list_p=TRANS.entry[ti].i_place;
        while((list_p!=NULL)&&(!poli)){
         list_t=PLACE.entry[list_p->p_name].o_trans;
         while(list_t!=NULL){
          if(TRANS.entry[list_t->t_name].exec==1){
            sp_list[lt].xtion=list_t->t_name;
            sp_list[lt].bufno=list_t->bufferno;
            lt++;
          }
          list_t=list_t->next;
         }
         /*if more than one exit the loop,
           does not check further*/
         if(lt>1)poli=TRUE;
         list_p=list_p->next;
        }

        if(poli){
         num=(float)rand()/factor();
         found=FALSE;
         lo=0.00;
```

```
 hi=data->buffer[sp_list[0].bufno].time;
 if(hi==0.0)
  /*no weight record,pick one randomly*/
  i=sp_list[rn_choice(lt)-1].xtion;
 else{
  /*now consider weight*/
  lk=1;
  while((!found)&&(lk<lt)){
    printf("lo=%f;hi=%f\n",lo,hi);
    /*if probability range match the random*/
    if((num>=lo)&&(num<hi))
         found=TRUE;
    /*else  continue with next range*/
    else{
         lo=hi;
         hi=hi+data->buffer[sp_list[lk].bufno].time;
         lk++;
    }
  }
  /*if found within range, get the xtion number*/
  if(found)i=sp_list[lk-1].xtion;
  /*if not stick with the one chosen before*/
  else i=ti;
 }
}/*end of if poli*/
else/* no other transition from the output place*/
  i=ti;
```

```
        }/*end of if li>1*/
        /*no other transition with same time as transition tm*/
        else
          i=st_list[0];

        printf("FIRES %d\n",i);/*fires transition i*/
        m=1;
                        /*** program start ***/

         statement_input == 1;
    /*  execute the trans_statement */
       if (programflag == 1 )
        {
         /*exec_trans_statement(data,i,TRANS.entry[i].flag1);*/
/*
exec_stmt1(data,i,9999);
         TRANS.entry[i].flag1 =0;
*/
         if ((ffp=fopen("tran.f","w"))==NULL) {
        printf("Can't open the tran.f file");
        exit(0);
        }
         fprintf(ffp,"%d\n",i);
         fclose(ffp);
         if (stat("processid.f",&statbuf2)==-1) system("exec_stmt1&");
        else {
                kill(PID,9);
```

```
            system("rm processid.f");
            system("exec_stmt1&");
            }
        for(;;)
        if(stat("flag.f",&statbuf1) != -1){
            ffp=fopen("flag.f","r");
            fscanf(ffp,"%s\n",SPID);
            PID=atoi(SPID);
            fclose(ffp);
            system("rm flag.f");
             break;
            }
         update_dbdq(data);
         TRANS.entry[i].flag1 =0;
}


        XBell(dpy,500);
           onlyflag=1;
           refreshA(w,data);
           /*refresh1(canvas,client_data, client_data);          */
        /* count down tokens in input places */
                    list_p = TRANS.entry[i].i_place;
                    timemax =0;

                    timemax = place_p->entry[list_p->p_name].token_e.time;

                    while (list_p != NULL ) {
```

```
                    if ( place_p->entry[list_p->p_name].token_e.time >
timemax)
                        timemax = place_p->entry[list_p-
>p_name].token_e.time;
                if(traceflag)
                    fprintf(ft, "transition %d,  fires at time  = %f
\n",i,TRANS.entry[i].time+timemax);
                if(strstr(place_p->entry[list_p->p_name].p_name,"E")==NULL ||
count_trans(place_p->entry[list_p->p_name].i_trans)!=0)
                {
                    n=PLACE.entry[list_p->p_name].token_e.number;
                    printf("\nIN TO cps2firsim **********\n");

                if(list_p->bufferno!=PMAX+1)
                    k=n-data->buffer[list_p->bufferno].token;
                else
                  k=n-1;
                    printf("n %d",n );
                    if (k < 0)  {
                        f=1;
                        break;
                    }
                    /*sprintf(test,"%d",n);
                    printf("test%s",test);

                    printf(" bufferno%d \n", PLACE.entry[list_p-
>p_name].bufferno);*/
```

```
                        /*XDrawString(dpy,win,data->andgc,data-
>buffer[PLACE.entry[list_p->p_name].bufferno].x2-
                          5,
                          data->buffer[PLACE.entry[list_p-
>p_name].bufferno].y2+5, test,strlen(test));*/
                        /*for (g=0;g<10000;g++)
  ;*/

                        /*************display in xwindow*****************/
                        /*sprintf(test,"%d",k);
                        for (b=0;b<100;b++) {
                              XDrawString(dpy,win,data->gc,data-
>buffer[PLACE.entry[list_p->p_name].bufferno].x2-
                                5,
                                data->buffer[PLACE.entry[list_p-
>p_name].bufferno].y2+5, test,strlen(test));
                        }*/
                        /*for (g=0;g<10000;g++)
  ;*/
                        place_p->entry[list_p->p_name].token_e.number=k;
                  /*if(strstr(place_p->entry[list_p->p_name].p_name,"E")==NULL
)*/
                        data->buffer[PLACE.entry[list_p-
>p_name].bufferno].token=k;
                  if(place_p->entry[list_p->p_name].token_e.tokenL!=NULL)
                    list_tok = place_p->entry[list_p->p_name].token_e.tokenL;
                    if(list_tok->next==NULL)
```

```
                    place_p->entry[list_p->p_name].token_e.tokenL=NULL;
                  else
                  {
                    place_p->entry[list_p->p_name].token_e.tokenL=list_tok-
>next;
                    place_p->entry[list_p->p_name].token_e.time=list_tok-
>next->time;
                    list_tok->next=NULL;
                  }
                  free(list_tok);


                    /*for (g=0;g<10000;g++)
 ;*/
                    amark[list_p->p_name]--;
                }
                    list_p = list_p->next;
              } /*end while loop*/
            /*    LU add  */
              if (dis_fg==1) {
             /* rearrange the globle array t[]  */
             if (i < (pos_num -1)){
                strcpy(t[TRANS.entry[i].pos],"");
            strcpy(t[TRANS.entry[i].pos+1],t[TRANS.entry[i].pos]);
             } else if(i == (pos_num -1)) {
                strcpy(t[0],t[pos_num - 1]);
                strcpy(t[pos_num - 1],"");
```

```
                }
                  /* actually move cart to new place */
                  move_dis(data->gc,data->foreground,data->background,data-
>andgc,TRANS.entry[i].pos);
                }

                list_p = TRANS.entry[i].o_place;
                while (list_p != NULL ) {
                /*place_p->entry[list_p->p_name].token_e.time =
                   TRANS.entry[i].time+ timemax ;*/
                /*printf("transition %d,TRANS.entry[i].time %f,
TRANS.entry[i].time+ timemax =%f \n",
                   i, TRANS.entry[i].time,TRANS.entry[i].time+timemax);
                printf("\ntime = %f at place %d \n", place_p->entry[list_p-
>p_name].token_e.time,
                   list_p->p_name );*/
                /*  differ  */
                if (fuzzyflag) {
                        place_p->entry[list_p->p_name].symbol_value =
TRANS.entry[i].time* timemax1 ;
                        printf("transition %d,TRANS.entry[i].time %f,
TRANS.entry[i].time* timemax1 =%f \n",
                         i, TRANS.entry[i].time,TRANS.entry[i].time*
timemax1);
                      printf("\nsymbol = %f at place %d \n", place_p-
>entry[list_p->p_name].symbol_value,
                         list_p->p_name );
```

```
                }

                if(strstr(place_p->entry[list_p->p_name].p_name,"E")==NULL ||
count_trans(place_p->entry[list_p->p_name].o_trans)!=0)
                {
                        n=PLACE.entry[list_p->p_name].token_e.number;
                        printf("list_p->p_name %d\n",list_p->p_name);
                        printf("^^^^^^^^^^count up  n=>%d\n",n);
                /*if(strstr(place_p->entry[list_p->p_name].p_name,"E")==NULL)*/
                if(list_p->bufferno!=PMAX+1)
                        k=n+data->buffer[list_p->bufferno].token;
                else
                    k=n+1;
                        printf("k %d",k);
                        if (f ==1 ) {
                                f=0;
                                break;
                        }
                        XBell(dpy,500);
                        /*XDrawString(dpy,win,data->andgc,data-
>buffer[PLACE.entry[list_p->p_name].bufferno].x2-
                                5,
                                data->buffer[PLACE.entry[list_p-
>p_name].bufferno].y2+5, test,strlen(test));*/

                        /*   for (g=0;g<10000;g++)
  ;*/
```

```
                    /*******display in Xwindow *******/
                    /*sprintf(test,"%d",k);
                    for (b=0;b<100;b++) {
                        XDrawString(dpy,win,data->gc,data-
>buffer[PLACE.entry[list_p->p_name].bufferno].x2-
                        5,
                        data->buffer[PLACE.entry[list_p-
>p_name].bufferno].y2+5, test,strlen(test));*/
                        /*XDrawString(dpy,win,data->gc,data-
>buffer[list_p->p_name].x2 -5,
        data->buffer[list_p->p_name].y2 +5, test,strlen(test));
                    }*/

                    /*  for (g=0;g<10000;g++)
  ;*/
                    place_p->entry[list_p->p_name].token_e.number=k;
                if(strstr(place_p->entry[list_p->p_name].p_name,"E")==NULL)
                    data->buffer[PLACE.entry[list_p-
>p_name].bufferno].token=k;

                    list_to = PLACE.entry[list_p->p_name].token_e.tokenL;
                    if(list_to==NULL)
                    {
            list_tok = (struct tokenlist *) malloc(sizeof(*list_tok));
            list_tok->time=TRANS.entry[i].time+timemax;
            list_tok->next=NULL;
```

```
                    PLACE.entry[list_p->p_name].token_e.tokenL=list_tok;
                    PLACE.entry[list_p-
>p_name].token_e.time=TRANS.entry[i].time+timemax;
                  }
                  else {
                  while(list_to->next!=NULL)
                    list_to=list_to->next;
                  list_to=list_tok;
                     }
                        /*for (g=0;g<10000;g++)
 ;*/
                        amark[list_p->p_name]++;
                     }
                        list_p=list_p->next;
                  }/*end while loop*/
       /*end of counting up token */
              /*   break;
              }end of if */

       /*put the rest firable back to queue*/
     for(g=0;g<li;g++){
      list_p=trans_p->entry[st_list[g]].i_place;
      if((list_p!=NULL)&&(fire(list_p,place_p,amark,data))){
        trans_p->entry[st_list[g]].exec=1;
        printf("%d back to queue\n",st_list[g]);
        if(expflag)
       TRANS.entry[st_list[g]].time=gen_arr(TRANS.entry[st_list[g]].avgtime);
```

```
    else
   TRANS.entry[st_list[g]].time=TRANS.entry[st_list[g]].avgtime;
  timemax =0;
  timemax = place_p->entry[list_p->p_name].token_e.time;
  while (list_p != NULL ) {
   if ( place_p->entry[list_p->p_name].token_e.time > timemax)
  timemax = place_p->entry[list_p->p_name].token_e.time;
   list_p=list_p->next;
  }
  TRANS.entry[st_list[g]].enabletime=timemax;
  insert(head,st_list[g],data,timemax);
  nf++; /*increment counter for number of firable tr.*/
 }
 else
  trans_p->entry[st_list[g]].exec=0;
}

/*put the next firable into the queue*/
list_p=TRANS.entry[i].o_place;
while(list_p!=NULL){
  list_t=PLACE.entry[list_p->p_name].o_trans;
  printf("p_name %d\n",list_p->p_name);
  while(list_t!=NULL){
   list_p2=TRANS.entry[list_t->t_name].i_place;
   if((list_p2!=NULL)&&(fire(list_p2,place_p,amark,data))&&
     (trans_p->entry[list_t->t_name].exec!=1)){
     trans_p->entry[list_t->t_name].exec=1;
```

```
  if(expflag)
 TRANS.entry[list_t->t_name].time=
 gen_arr(TRANS.entry[list_t->t_name].avgtime);
  else
 TRANS.entry[list_t->t_name].time=
 TRANS.entry[list_t->t_name].avgtime;
timemax =0;
timemax = place_p->entry[list_p2->p_name].token_e.time;
while (list_p2 != NULL ) {
 if ( place_p->entry[list_p2->p_name].token_e.time > timemax)
timemax = place_p->entry[list_p2->p_name].token_e.time;
 list_p2=list_p2->next;
}
TRANS.entry[list_t->t_name].enabletime=timemax;
insert(head,list_t->t_name,data,timemax);
printf("next %d\n",list_t->t_name);
nf++; /*increment counter for number of firable tr.*/
  }
  list_t=list_t->next;
 }
 list_p=list_p->next;
}


        for (q = 0;q <PMAX;q++)
        {
                if(amark[q] != mark_p[q])
                        break;
```

```
                }
                /*bp  if(j!=0 && q==PMAX)
                        if(traceflag){
                         traceflag=0;
                         fclose(ft);
                        }
                        return;*/
                /*for (q=0;q<TRANS.number;q++)
        TRANS.entry[q].exec=0;*/
                /*while((first=deq(head))!=NULL);
                while((trans_t = cps2net(trans_p,place_p,amark,data))
                  !=NULL) {
                        if(TRANS.entry[data->X-1].exec==1)
                                TRANS.entry[data->X-1].exec=1;
                        if(TRANS.entry[data->X-1].flag==0){
                                if(expflag)
                                  TRANS.entry[data->X-
1].time=gen_arr(TRANS.entry[data->X-1].avgtime);
                                else
                                  TRANS.entry[data->X-1].time=TRANS.entry[data->X-
1].avgtime;
                        }
                        printf("****TRANS.entry[%d].exec=1\n",data->X-1);
                        TRANS.entry[data->X-1].flag=1;
                        insert(head,data->X-1,data);
                }*/
    /*}*/
```

```
                /*for (k=0;k<10000;k++) ;*/
        if(m==0){
                if(traceflag){
                  traceflag=0;
                  fclose(ft);
                }
                return;
        }
      call_delay(my_delay,TRANS.entry[i].time);
      refresh(canvas,data);
      j++; /*increment step*/
      } /*end d while loop*/
      onlyflag=0;
      printf("End of Simulate\n");
      if(traceflag)
      {
      traceflag = 0;
      fclose(ft);
      }
return;
}


/*remove from queue*/
struct timeq *deq(head)
struct timeq *head;
```

```
{
  struct timeq *temp1,*temp2;
  temp1 = head ->next;
  if(temp1 != NULL){
    head->next = temp1->next;
    temp1->next = NULL;
    temp2 = head->next;
    while(temp2!=NULL){
      temp2 = temp2->next;
    }
    return(temp1);
  }
  else
    return(NULL);
}
```

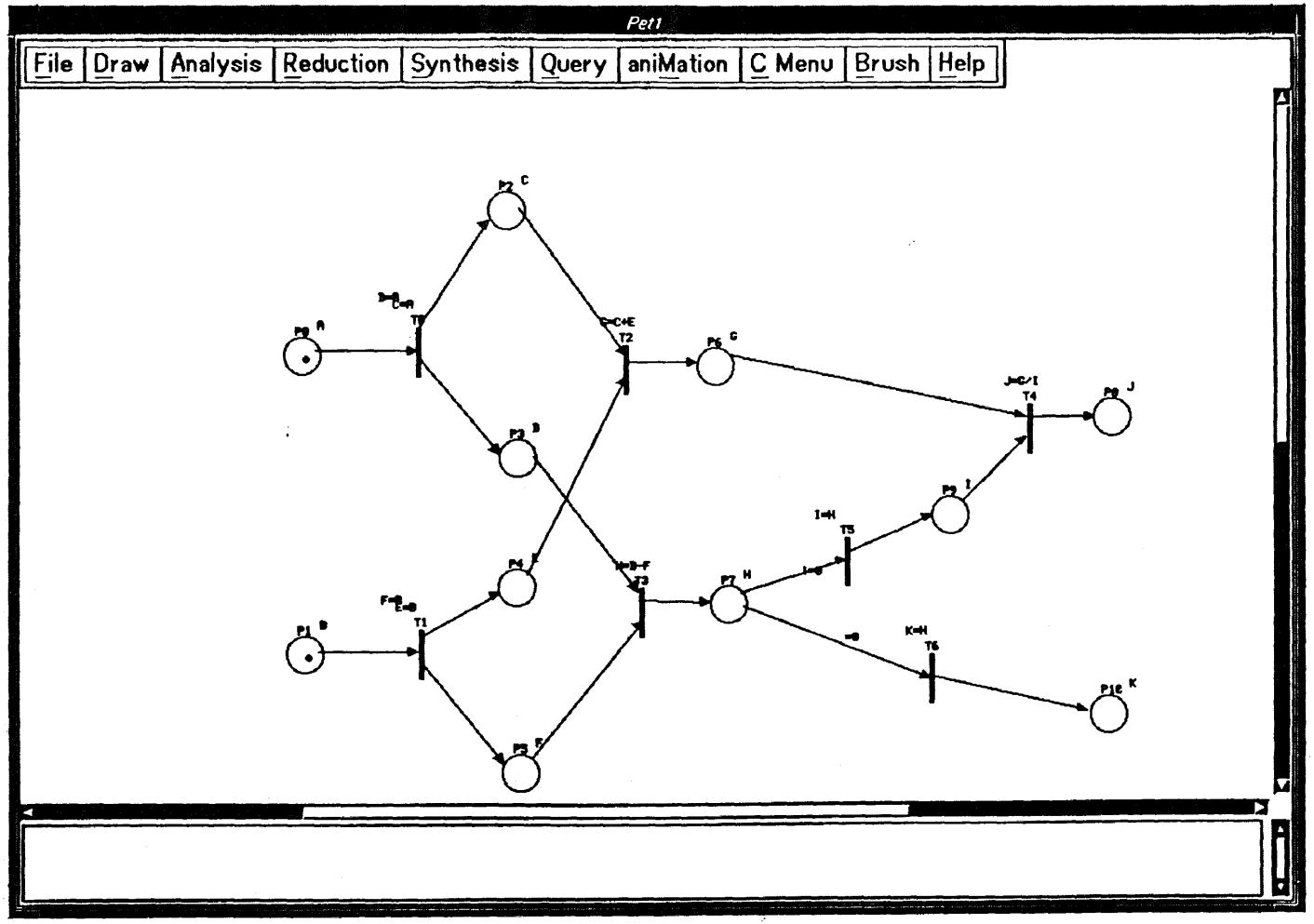

Figure 1 A Typical Petri Net Illustration.

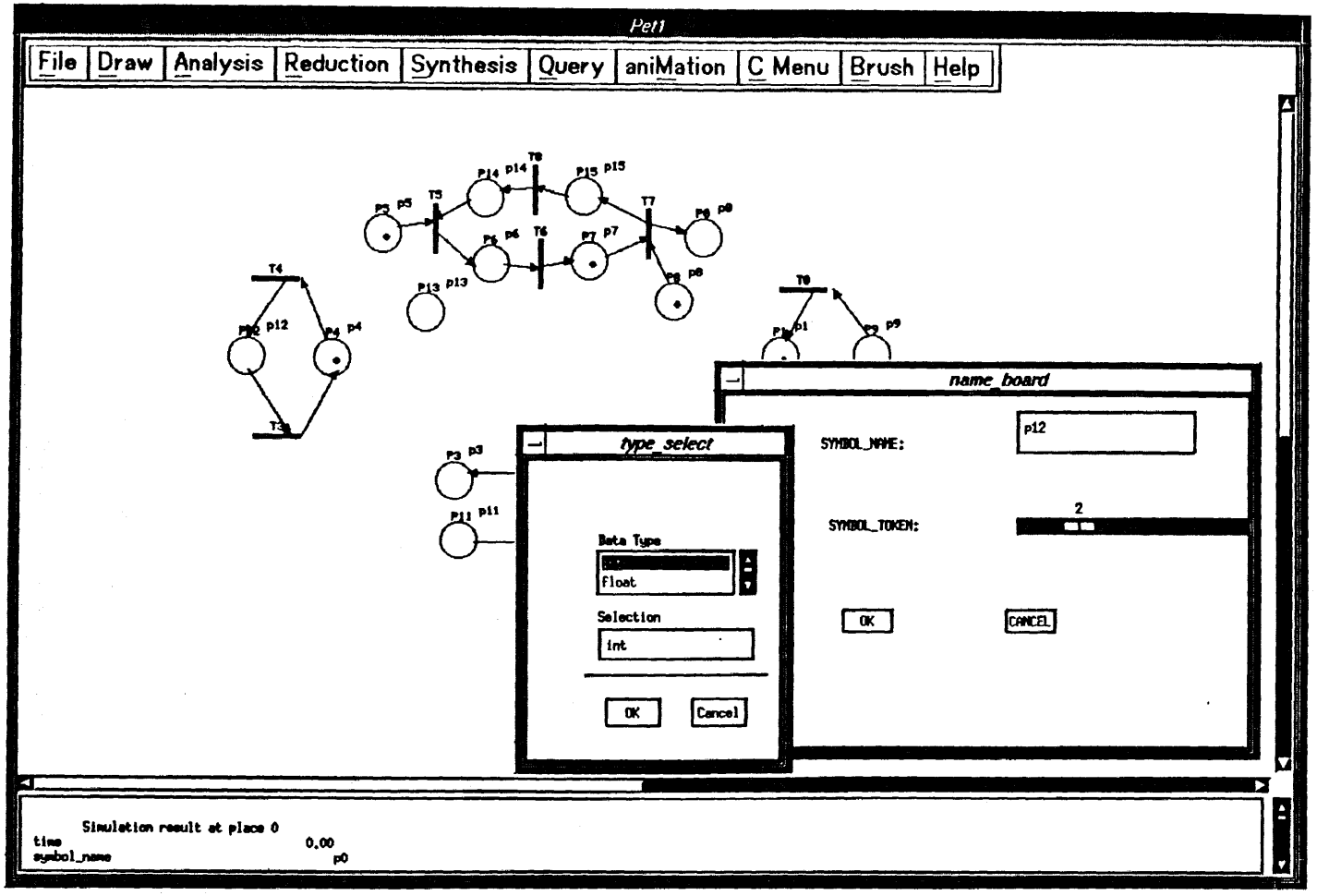

Figure 2 A Selection Window for a Variable Type, Name, and Token Number for a Place.

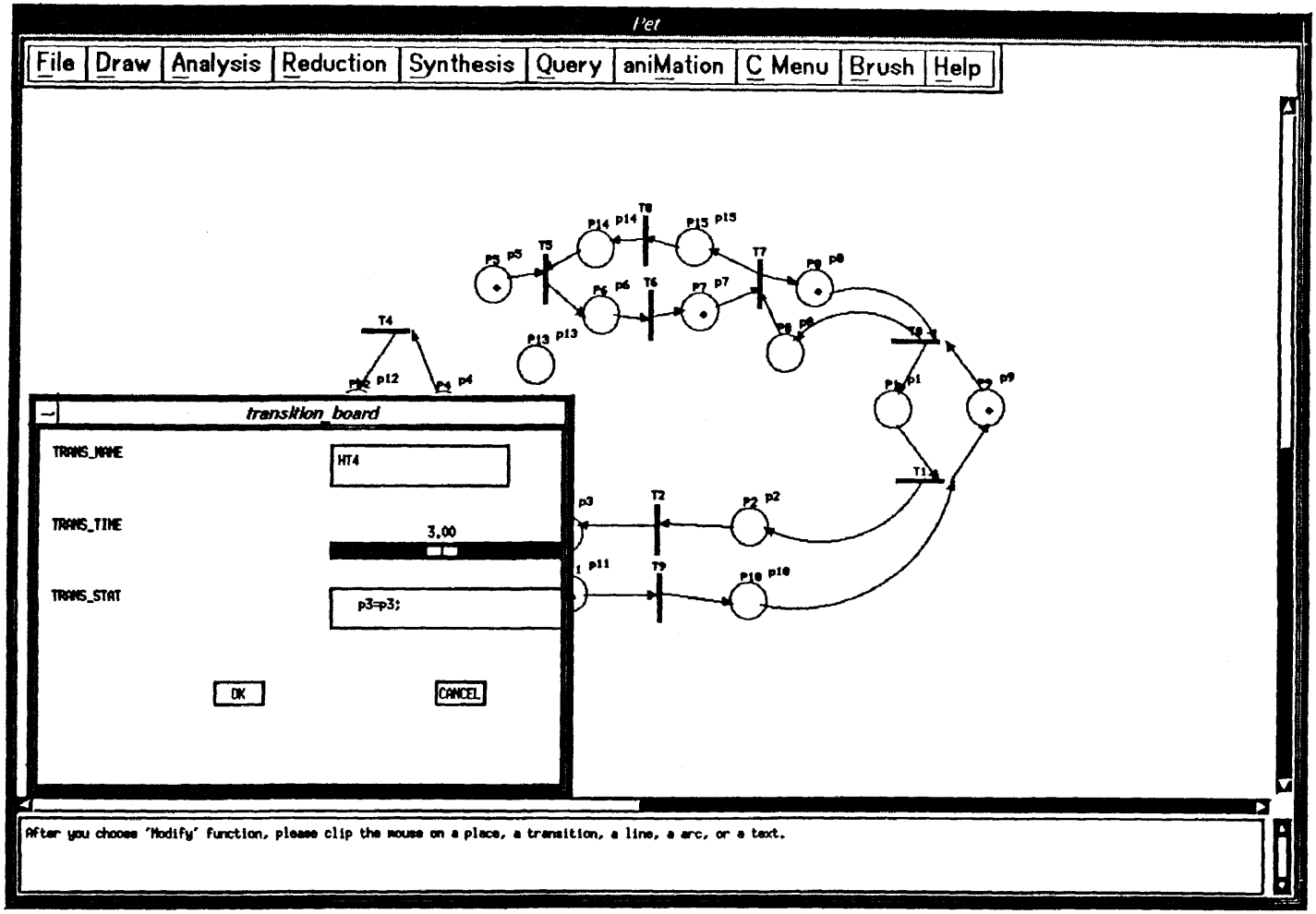

Figure 3 A Selection Window for Name, Firing Time, and Statement for a Transition.

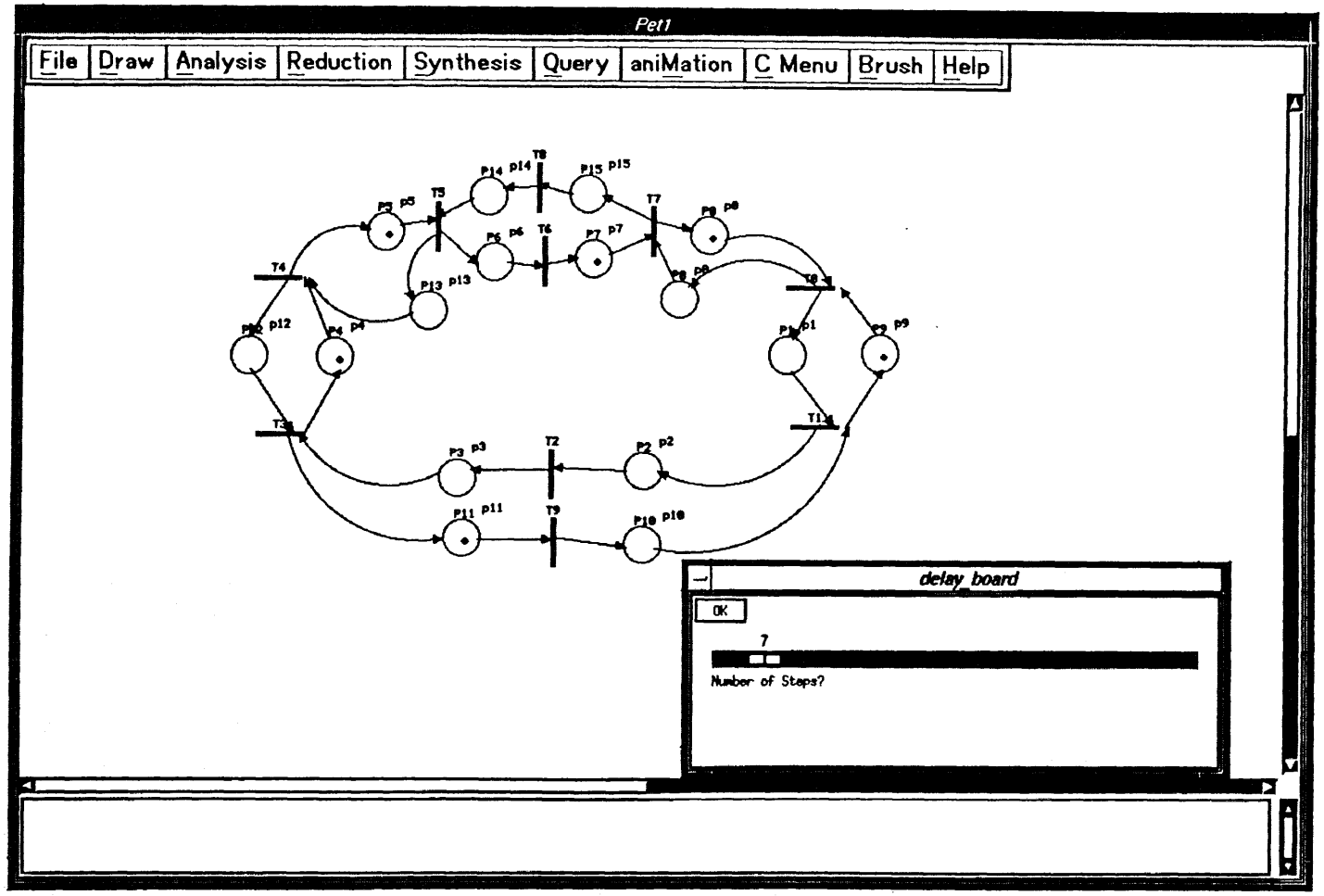

Figure 4 A Typical Scale Widget Display.

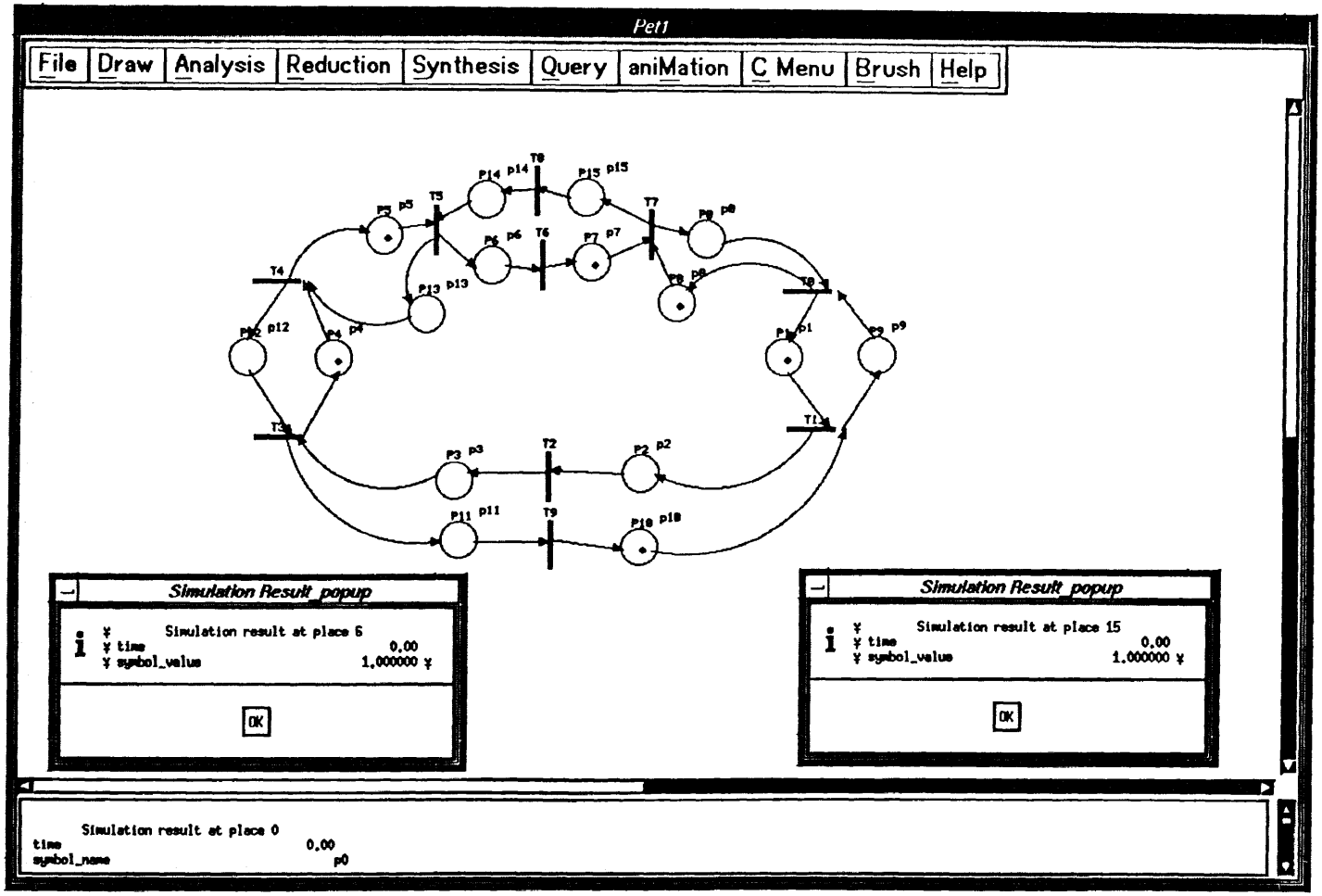

Figure 5 An Example Simulation Result Display.

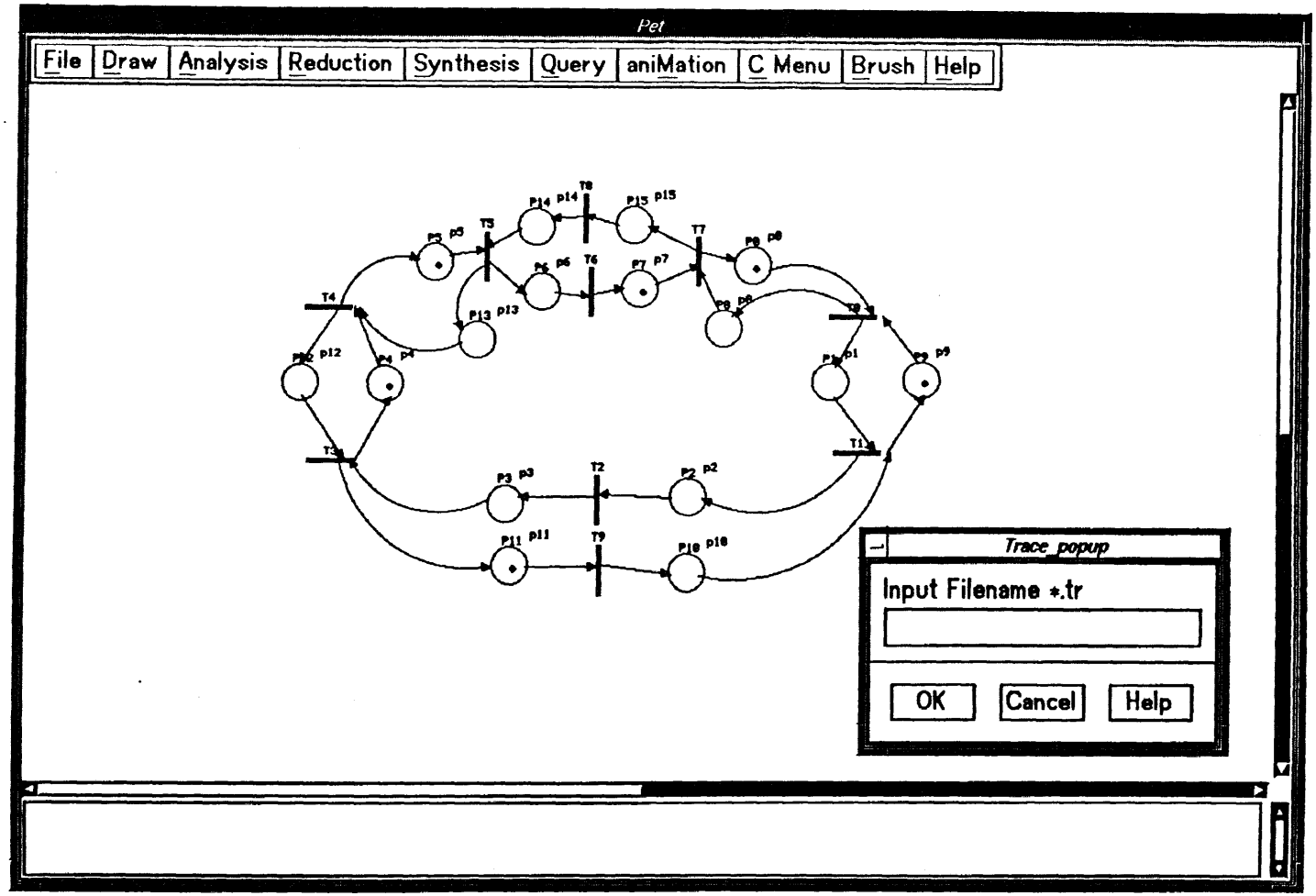

Figure 6 A Pop-Up Window to Input the Trace Filename.

REFERENCES

1. Pan, Z. D, and Mark A. Linton. "Supporting Reverse Execution of Parallel Programs." (1990).

2. Fidge, C. J. "Partial Orders for Parallel Debugging." (1990).

3. Fowler, R. J., Thomas J. Leblanc, and John M. Mellor Crummey. "An Integrated Approach to Parallel Program Debugging and Performance Analysis on Large-Scale Multiprocessors." (1990).

4. Ziya, A., and Ilya Gertner. "High-Level Debugging in Parasight." (1990).

5. Yaw, Y. "Analysis and Synthesis of distributed systems and protocols.", *Ph.D. Dissertation, University of California, Berkeley.* (1987).

6. Yaw, Y., C. V. Ramamoorthy, and W. T. Tsai. "A synthesis technique for designing concurrent systems." *Second Parallel Processing Symposium.* (1988): 143-166.

7. Yaw Y., and F. L. Foun. "The Algorithm of a synthesis technique for concurrent systems." *Third International Petri Net Conference.* (1989): 266-276.

8. Yaw Y., D. C. Hu, and C-W Huang. "An Expert System based on Petri Net Graphics Tool.", *submitted* . (1992).

9. Murata, T. "Petri Nets : Properties, analysis, and applications." *Proceedings of the IEEE, Vol. 77, No. 4.* (1989): 542.

10. Socha, D., Mary L. Bailey, and David Notkin. "Voyeur: Graphical Views of Parallel Programs." (1991).

11. Miller, P. B., and Jong-Deok Choi. "A Mechanism for Efficient Debugging of Parallel Programs." (1991).

12. Petri, C. "Concepts of Net Theory.", *Proceedings of the Symposium and Summer School on Mathematical Foundation of Computer Science, High Tatras, Mathematical Institute of the Slovak Academy of Science.* (1973).

13. Peterson, J.L. "Petri Nets." (1989).