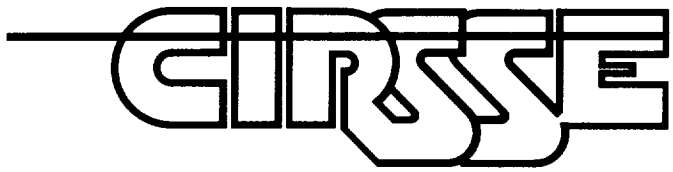

Center for Intelligent Robotic Systems for Space Exploration

Rensselaer Polytechnic Institute
Troy, New York 12180-3590

A DISTRIBUTED PETRI NET CONTROLLER FOR A DUAL ARM TESTBED

by

Atle Bjanes

Rensselaer Polytechnic Institute
Electrical, Computer, and Systems Engineering
Troy, New York 12180-3590

January 1991

CIRSSE REPORT #82

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENT

I would like to thank Professor Alan Desrochers for the help and direction given to me in this research. I also wish to thank Professor Frank DiCesare and the other members of the Petri net research group at RPI for providing ideas as well.

My special thanks go to Laura, my dear wife, for supporting me while pursuing the degree. David, born at the onset of the work, has given me much inspiration not to work on this project - you are a joy.

I am grateful to the Norway-America Association and to the New York State Center for Advanced Technology in Automation and Robotics at RPI for sponsoring my work.

A final thank you to the people at the CIRSSE labs for the soccer matches and fun while being there.

ABSTRACT

This thesis describes the design and functionality of a Distributed Petri Net Controller (DPNC). The controller runs under X Windows to provide a graphical interface. The DPNC allows users to distribute a Petri net across several host computers linked together via a TCP/IP interface. A sub-net executes on each host, interacting with the other sub-nets by passing a token vector from host to host. One host has a command window which monitors and controls the distributed controller. The input to the DPNC is a net definition file generated by GreatSPN. Thus, a net may be designed, analyzed and verified using this package before implementation. The net is distributed to the hosts by tagging transitions that are host-critical with the appropriate host number. The controller will then distribute the remaining places and transitions to the hosts by generating the local nets, the local marking vectors and the global marking vector. Each transition can have one or more preconditions which must be fulfilled before the transition can fire, as well as one or more post-processes to be executed after the transition fires. These implement the actual input/output to the environment (machines, signals etc.). The DPNC may also be used to simulate a GreatSPN net since stochastic and deterministic firing rates are implemented in the controller for timed transitions.

CHAPTER 1
INTRODUCTION

1.1 Problem Description

Petri nets are becoming popular for modeling manufacturing systems [13]. This project attempts to facilitate the use of Petri Nets for controlling discrete event dynamic systems such as manufacturing systems and flexible manufacturing systems incorporating robots. The growing complexity and emphasis on productivity forces the issues of reliability, ease of implementation, prototyping and verification. Since Petri nets can be analytically proven to be free of deadlock and bounded, the reliability issue is helped using the Petri net technique. Since it is costly to do prototyping using actual machines, a Petri net model of a manufacturing system may be analyzed for its performance, thus further enhancing the value of this approach.

Discrete event dynamic systems are characterized by concurrency and asynchronous operation and Petri nets can capture these traits easily. To reduce the complexity of using Petri nets, design tools and analysis tools are necessary for building correct controllers.

The distributed Petri net controller (DPNC) was designed with the above in mind. To further enhance the Petri net property of concurrency, a distributed architecture was chosen whereby the Petri net is distributed across several host computers which are linked together via a local area network. The controller was demonstrated using a model of the dual arm testbed in Rensselaer's Center for Intelligent Robotic Systems for Space Exploration (CIRSSE).

Other, more practical requirements for a controller include a need for flexibility in allowing the designer to easily interface with machines, devices, signals, etc. that are connected to the individual host. Since the Petri net is distributed, there must

be some scheme for distributing the places and transitions of the net on the various hosts. For operation of a system controlled by the DPNC, an interface that provides control and display of status in real-time must be provided. The status of the net (and thereby the machines) must be available on the operator screen.

1.2 Thesis Organization

Chapter 2 describes briefly what a Petri net is and how it can be applied to control. The effort of this project is compared to earlier work. Chapter 3 describes the internal design of the controller, the various programs and implementation details. Chapter 4 describes how the DPNC may be used to control a physical system and chapter 5 gives a specific application of the controller. A model of the CIRSSE testbed [8] is used to demonstrate the utility of the DPNC. Chapter 6 discusses the results obtained. Chapter 7 has a discussion on future directions of this project. Appendix A contains source code listings of all programs included in the controller package. Appendix B lists the programs and code written to implement the example of chapter 5.

CHAPTER 2
PETRI NETS AND MANUFACTURING

2.1 Introduction

This chapter gives a description of the Petri net theory on which the project is based. Uses of Petri nets in manufacturing are also discussed and some of the software tools available are reviewed. Existing implementations of Petri net based controllers are discussed.

2.2 What is a DEDS?

Discrete-event dynamic systems (DEDS) comprise a growing segment of the manufacturing base today as automation is becoming more widespread. This includes unmanned systems for space exploration and work in other hazardous environments. A DEDS is any system where resources and materials are quantized in discrete units. The opposite would be a continuous flow process plant, which could be described using differential equations. Petri nets are well suited to describing and modeling DEDS and are thus chosen as the basis for a controller suitable for DEDS.

2.3 What is a Petri Net?

A Petri net is a directed, weighted, bipartite graph. The graph consists of two kinds of nodes, transitions and places, which are interconnected by arcs from either a place to a transition or vice-versa. The arcs have weights representing the number of parallel arcs between two nodes.

A marking of the net represents a particular state of the net. The places are marked with a number of tokens, and a marking vector represents the number of

Table 2.1: Common Interpretations of Transitions and Places

Input Places	Transitions	Output Places
Preconditions	Event	Postconditions
Input data	Computation Step	Output data
Input signals	Signal Processor	Output Signals
Resource request	Task or Job	Resource released
Conditions	Clause in logic	Conclusions
Buffers	Processor	Buffers

tokens in each place of the net. A marking is denoted by M, where M is an $m x 1$ vector (the net has m places). The pth component of M, denoted M(p), is the number of tokens currently assigned to place p.

The marking of the net determines the firing of the net, or the flow of tokens through the net. A transition can fire when there is at least k tokens in each of its input places with multiplicity k. When a transition can fire, it is enabled. It may, or may not fire, depending on the physical significance attached to the particular transition. When a transition fires, it consumes k tokens corresponding to the multiplicity of the arc from each of its input places and deposits a number of tokens in each of its output places, depending on the multiplicity of the output arcs to each output place.

The places and transitions hold some physical significance; a place may represent the state of a machine, an input signal, a request by one machine to another, a resource, a part or a buffer. Table 2.1 gives some common interpretations[1].

2.4 Petri-Nets in Manufacturing

A simple example of a Petri Net model of a manufacturing system is a 2 machine one buffer model (Fig 1.) representing a producer machine and a consumer machine. In this model, the place labeled *Piece Made* represents a piece made by the

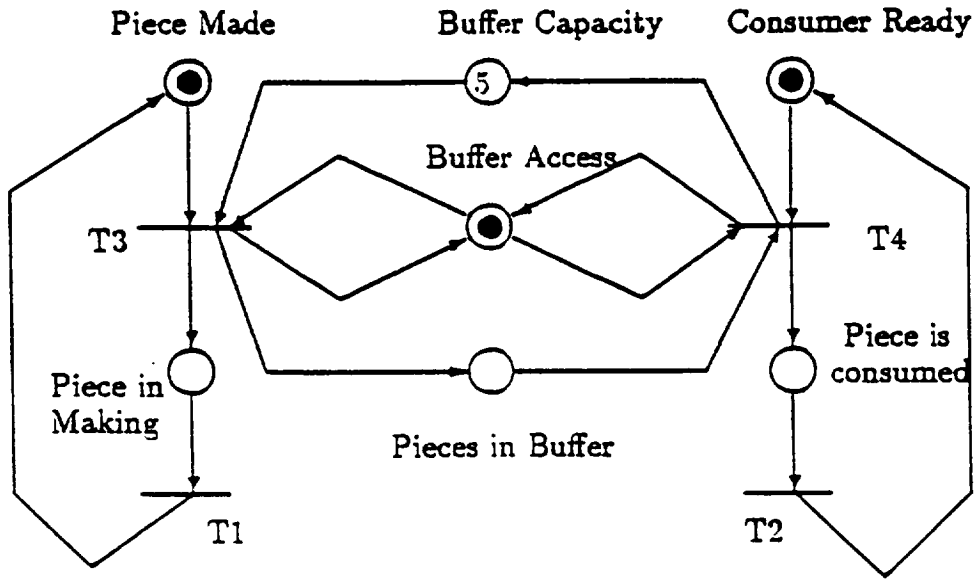

Figure 2.1: Simple Producer/Consumer Petri Net Model

producer. Place *Piece in Making* represents the producer is currently making a piece. The transition *T1* fires when the machine is done producing, taking the input (raw piece) and depositing a finished piece in the input and output places, respectively. Conversely, the place *Consumer Ready* signifies (when marked) that the consumer is ready to accept another piece. Transition *T4* fires when the consumer is ready and the buffer *Pieces in Buffer* contains at least one piece. Transition *T2* represents the consumption of a piece.

The utility of Petri nets in modeling DEDS may be seen from the fact that the buffer size can be changed by simply changing the initial marking of place *Buffer Capacity*. The number of producers (or consumers) may be changed simply by changing the initial marking of places *Piece Made* or *Consumer Ready*.

An example of why Petri nets are useful for control of DEDS is that the Petri net may be analyzed for deadlocks, cycles and boundedness. A Petri net which is free from deadlock and bounded, will ensure that the Petri net control system logic

will have these desired properties as well. The model of the system may also be used for performance analysis given performance data for the individual machines and components in the system. Of course, the accuracy of the analysis is dependent on the accuracy of the assumptions made.

The above simple system may be used to implement a distributed Petri net controller. Input and output procedures associated with the transitions can be implemented on the controller in some chosen language and can be interfaced with the controller hardware.

2.5 Petri-Net Design Tools

There exist a number of software tools for designing and analyzing Petri nets. One of the most comprehensive tools is GreatSPN[2]. This program may be used to graphically design a Petri net using a Sun workstation. The net may be analyzed for its P- and T-invariants, as well as transition throughput and token probability distribution.

SPNP is a set of 'C' subroutines[3] that are used to solve the steady-state Markov chain to find token distribution and transition throughput. It is very powerful, but somewhat cryptic to use. A tool has been written at RPI, called GreatSPN2SPNP[4], which translates the net information files of GreatSPN to an SPNP program. This allows for much easier building of SPNP programs.

There are other programs as well, but the above two may be the most widely used and known.

2.6 Comparisons with Earlier Work

Three previous Petri net control schemes have been designed and implemented at RPI. The first of these by Crockett[5] is an application-independent controller using Petri nets to describe the sequencing of operations. A hierarchical structure

was used by defining *macro* nodes to embed sub-Petri nets in a large model.

This control scheme was further enhanced by Kasturia[6] by including colored Petri nets to describe the controller. Rudolph[7] improved on [5] by simplifying the Petri net description files. The controller in [7] was used to control a miniature flexible manufacturing plant.

Outside of RPI, Petri nets have been used for fault-tolerant systems [9], as programmable logic controllers [10], [11] and for controlling flexible automation systems [12] [14]. Petri nets were used in [12] as the basis for control programs for general purpose factory automation (FA) controllers. A total FA system has been implemented in [14] with Petri net based station controllers networked over a local area network.

This project addresses three major issues compared to the above references:

1). Allow seamless integration of the controller with a Petri net design package (GreatSPN).

2). Use only generalized stochastic Petri nets to allow analysis using algorithms for generalized stochastic Petri nets (GSPN), even if this may mean loss of the hierarchical features in [5].

3). To use distributed processing as a means to facilitate the concurrency and asynchronous features of Petri nets.

2.7 Summary

Petri nets have many desirable features for control of discrete event dynamic systems. The ability to analytically determine deadlocks, boundedness and finding cycles as well as performance analysis of the system contributes to the design of both the system to be controlled and the controller. Past work both at RPI and

outside have taken advantage of these features. Chapter 3 describes the DPNC and how Petri nets are incorporated in the controller.

CHAPTER 3
CONTROLLER DESIGN

3.1 Introduction

This chapter describes the architecture of the distributed Petri net controller. Figure 3.1 shows the architecture for an *n-1* node controller. In the depicted scenario there are *n-1* host computers, each connected to a local area network. It is assumed that host computer 0 is a SUN workstation with Suntools for running GreatSPN and X windows (X11R4) for running the display and command programs.

The DPNC consists of off-line and on-line programs. The Petri net forming the basis for the controller is designed off-line using GreatSPN. An editor is used to write the various host specific driver routines, and the 'C' compiler is used to compile the run-time programs. The run-time token players and supporting display and command programs are run on the various hosts. Each host also runs a device driver process that interfaces the individual token player to the physical machines connected to the host computer.

3.2 Major Programs of the DPNC

The DPNC is composed of several programs, each of which is responsible for the implementation of a major function. These are

build A shell script program that prompts the user for the hostnames the controller is to run on. The command `build contr` will use the GSPN file *contr.net* residing in the *$HOME/greatspn/nets* directory of the user. *build* will call all appropriate programs such as *assign* and *make* to generate the token players.

net2n_m This program is a modified version of *net2n.c* written by Andreas Nowatzyk at Carnegie-Mellon University, School of Computer Science, March 1989. The

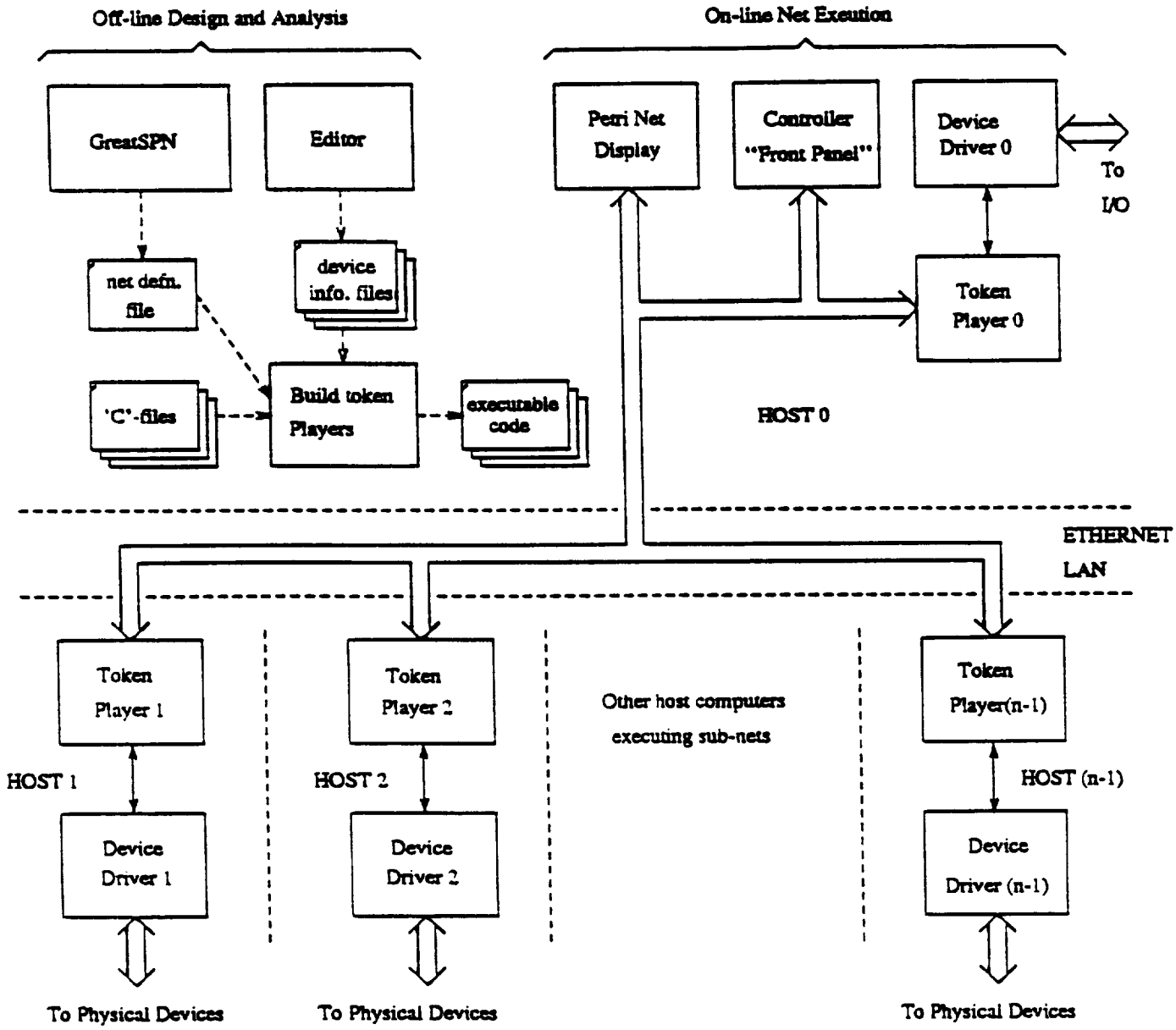

Figure 3.1: Architecture of Distributed Petri Net Controller

input to the program is a *.net* file and the output is a *.n* file containing the structure of the net only. The modification allows inhibitor arcs to have multiplicity greater than one.

assign The net structure as given in the *.n* file is used by *assign* to distribute the Petri net across the sub-nets defined by assigning transitions in GreatSPN. The file *hostnames* generated by *build* contains the host names and numbers and is used by *assign* to give the token player the correct socket addresses. *assign* outputs the net definition files for all token players.

pn*n* Each token player is compiled with unique 'C' `include` data files generated by *assign*. The *n* programs (one for each sub-net) implement one sub-net where each token player runs asynchronously and fires any transitions that are enabled. The places that are common to transitions residing on different token players are called *global*. Global places are passed from token player to token player in a token-ring fashion using sockets during net execution.

XCommand Commands to the DPNC are issued using *XCommand* by moving the mouse to the appropriate button in the command window and clicking the left mouse button. The program will then send that command to each token player and the net display program via sockets.

XDisplay The Petri net is displayed in an X window made by this program. The net appears as it was made in GreatSPN. Each token player sends a firing vector to *XDisplay* during net execution and transitions are highlighted when they fire and token counts are shown in the places.

3.3 System Files

The DPNC is composed of several programs. Each program depends on input files and produces output files. The system files, shown in figure 3.2, are : (note

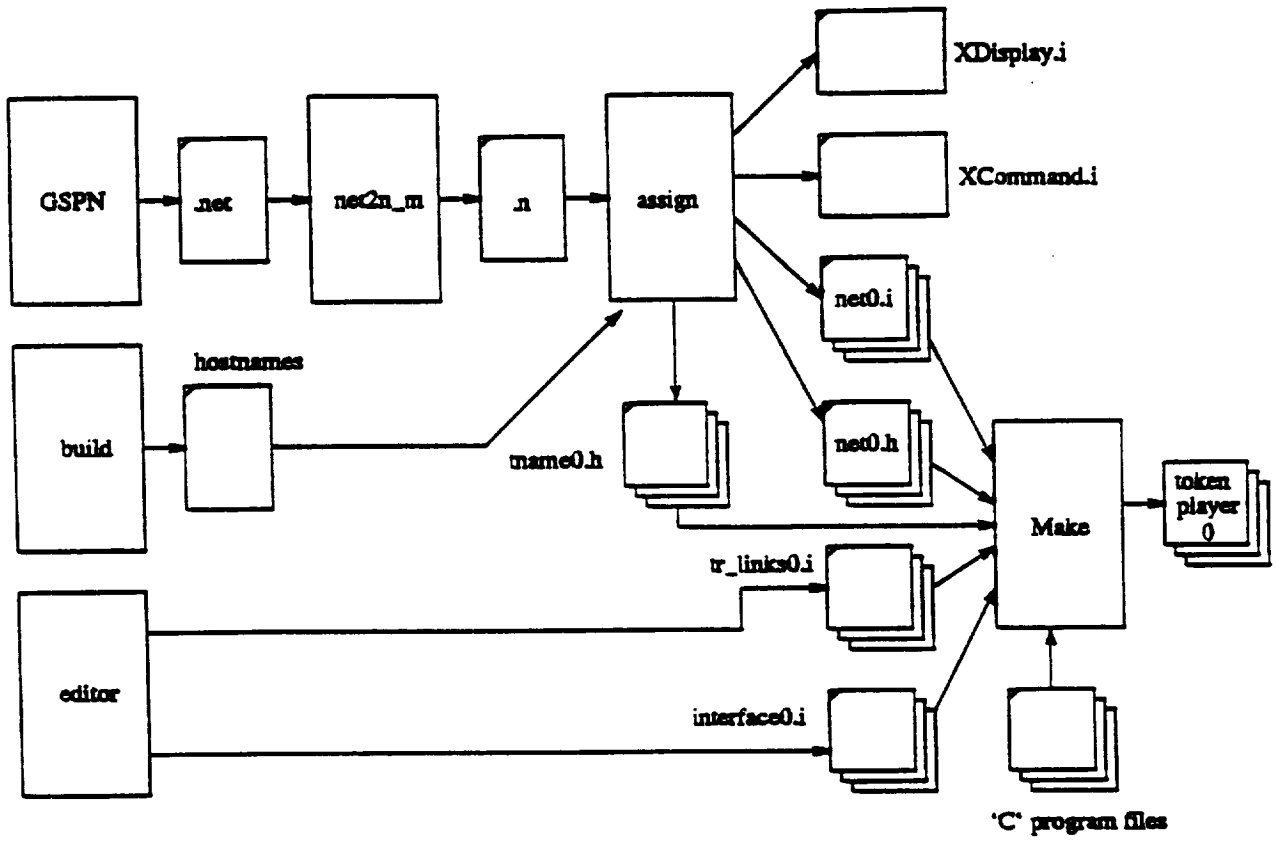

Figure 3.2: System Files

that the notation `name`*n*`.i` means that one file for each token player is required)

test.net The net definition file generated by GSPN. It contains the structure of the net as well as the geographical information for drawing the net. This file is copied from the users Petri net directory, e.g., */greatspn/nets* every time *build* is run.

test.n The net structure file. Note that *build* generates this file every time it is run. The file stores the most recent net to be built.

hostnames* This file contains the Internet host name and the host number (0 — n-1) for each of the *n* token players.

XCommand.i Contains the Internet address of each token player to connect to the command sockets and the display socket.

XDisplay.i Contains information about how the net has been partitioned. Also contains hostnames and addresses.

net*n*.i Sub-net definition files with 'C' code initialization of the data structures for each token player.

net*n*.h Contains the sizes of net data structures for each token player (the number of transitions and local and global places).

interface*n*.i* A library of routines to implement the interface to any machines connected to the individual host computer.

tr_links*n*.i* An initialization file with pointer assignments linking transitions to the input/output routines defined in *interfacen.i*.

tname*n*.h These include files contain the transition names and the transition indexing number at each token player. These numbers are necessary when the transitions are used for input/output. The files may be used as include files in the device driver programs so that instead of looking up the indexing number of a transition at a particular token player (which may change as the net is modified), the name of the transition may be used. This is possible since *tnamen.i* contains `# define` statements such as `# define T29_IN_AT_0 2`. Note that the symbols @, ! and ? are changed to *_AT_*, *_OUT_* and *_IN_* respectively to conform to allowed 'C' constant naming convention.

The designer is responsible for writing the files marked with * using a text editor (like EMACS or vi). These files contain the information for connecting the token players to any devices connected to their respective host computers.

3.4 Assigning Places and Transitions

assign distributes the Petri net into n sub-nets based on the transitions that are preassigned by the user in GSPN. To preassign a transition, the name of the transition is appended with *@h* where h is the host number. It is required to preassign only the transitions that will be enabled externally or will execute some command since these will be linked to a 'C' routine in the file *tr_links.c*.

assign will first read in the structural information from *name.n*. The places and transitions are assigned to the hosts by successively looking at each place to count the number of dependencies to already assigned (preassigned) transitions. If a place has only one host dependency, the place is local, if the place has more than one host dependency (i.e., a place has an arc to transitions residing at different hosts), the place is global.

The unassigned transitions are then checked to see how many dependencies (i.e., input, output or inhibitor places) it has on each host. The transition is placed at the highest host count.

The above two steps are repeated until all places and transitions are assigned to a host.

The places that are global become the *global marking vector* which is passed from token player to token player using the token-ring structure set up using sockets.

3.5 Data Structures

The sub-net structure is stored in an array of transitions in each token player. Some of the key elements of the data structure are shown in table 3.1.

Each token player also has data structures for the local- and global marking vectors. The local marking vector stores the marking of the local places at each token player whereas the global marking vector stores the marking of the global places when the global marking vector is available at the token player. Each data

Table 3.1: Data Structure in Token Players

flag	immediate	Immediate transition flag
flag	timed	Deterministic timed transition
int	(*preprocess) ()	Pointer to procedure
int	(*postprocess)()	pointer to procedure
long	LInputMask	Input mask for local places
long	GInputMask	Input mask for global places
short	LInputVector	Local input incidence matrix
short	LOutputVector	Local output incidence matrix
short	GInputVector	Global input incidence matrix
short	GOutputVector	Global output incidence matrix
short	LInhibitVector	Local inhibitor arc multiplicity
short	GInhibitVector	Global inhibitor arc multiplicity

structure also includes a mask bit for each place which is set when the place has a token.

3.6 Petri Net Execution Algorithm

The token players are independent processes running on separate host computers. Each token player executes a sub-net of the controller Petri net. The token player will evaluate each transition sequentially and fire the ones that are enabled. Whenever a transition fires, all transitions are evaluated over again. This is performed until no transitions can fire. The token player is then suspended until an external event occurs. An external event includes a socket message from either the command window, *XCommand*, the proceeding token player in the token ring, or a socket message from an external device driver. This makes the token player event-driven and helps reduce the load on the host computer.

A watchdog timer also generates periodical time-out events in case the token player should have lost a socket interrupt, which may happen when the host is heavily loaded (in the SUN Sparc Station case).

Each token player stores both a local marking vector and a global marking vector. Since the global marking vector is passed from token player to token player, the transitions that are dependent on input from the global marking vector are disabled when the global marking vector is not available at that particular token player. All other transitions at that token player may fire if otherwise enabled. If a transition has an output arc to a global place, the output tokens to the global place are temporarily held at the token player until the global marking vector again is available at that token player. These tokens are then added to the global places.

Each token player is allowed to retain the global marking vector for a fixed number of iterations of checking for enabled transitions. As soon as a token player has used its quota, it must be passed on to the next token player.

The token players are identical except from the sub-net executed. Token Player 0 is also unique since it provides the server socket for token player 1, and waits to open a client socket to token player *n-1*. A token player has a simplified structure which looks like the following:

- `initialize sub-net`
- `initalize socket connections`
- `set command to HALT`
- `while command not STOP`
    - `wait until event occurs`
    - `get_event() --- Read all sockets`
    - `switch (command)`
        - `case RUN : token_player() -- execute net`
        - `case HALT :`
        - `case RESET : re-initialize net`
        - `case STOP :`
- `close all sockets.`

token_player is the actual subroutine that executes the net at each token player. It has a simplified structure

```
•while command = RUN and get_next_transition() ≠ FALSE
        • get_event()
        • if transition enabled
            • fire the transition
            • set eventflag TRUE
        • if global marking period expired
            • release global marking vector to next token player
```

get_next_transition() returns the next transition to be checked. If all transitions have been checked to see if they can fire and none did, this routine will return FALSE, thus causing the routine *token_player()* to exit by releasing the global marking vector (if token player had the global marking vector in the first place in this iteration). The program is then suspended until the next event occurs.

get_event() is called to check the sockets in case a command, the global marking vector or a device driver message is available.

A transition is checked by *enable(tr)*. This routine returns true if the transition is enabled by having enough tokens in all input places, not more than the allowed tokens in the inhibitor places and all external preconditions satisfied. *fire(tr)* simply removes the tokens in the input places, executes the postprocess, if any, and deposits tokens in the output places. The *event_flag* is set to enable the token player to check all transitions again.

3.7 Run-Time Structure

A sequence of programs must be started to set up the DPNC correctly. The requirements are that an X-windows server has one terminal connected to each host

computer. A 3 node controller topology is given as an example of the run-time structure of a distributed Petri net controller in figure 3.2.

The X server may be any one of the host computers, the only critical part is that the processes run on their respective hosts. The sequence of processes to be started and which host terminal window to start it on is as follows:

HOST 0: Start XDisplay by entering `XDisplay`. This program will open a window on the X server where the net will be shown. XDisplay then waits for socket connections from each of the token players and the command program.

HOST 0: Start token player 0 by entering `pn0`.

HOST 1: Start token player 1 by entering `pn1`.

HOST 2: Start token player 2 by entering `pn2`.

HOST 0: Token Player 0 is at this point waiting to connect to token player 2 to close the virtual token network for global marking vector passing. By entering any character and pressing enter, token player 0 will connect to token player 2.

HOST 0: The final step is to start the command program. All other processes are now waiting for this program to open a socket to each of them. Enter `XCommand` to start the command program. A command menu will appear as soon as all sockets are connected.

After following the above sequence of starting processes, the DPNC would have several socket connections between the various processes. These sockets are depicted in figure 3.3.

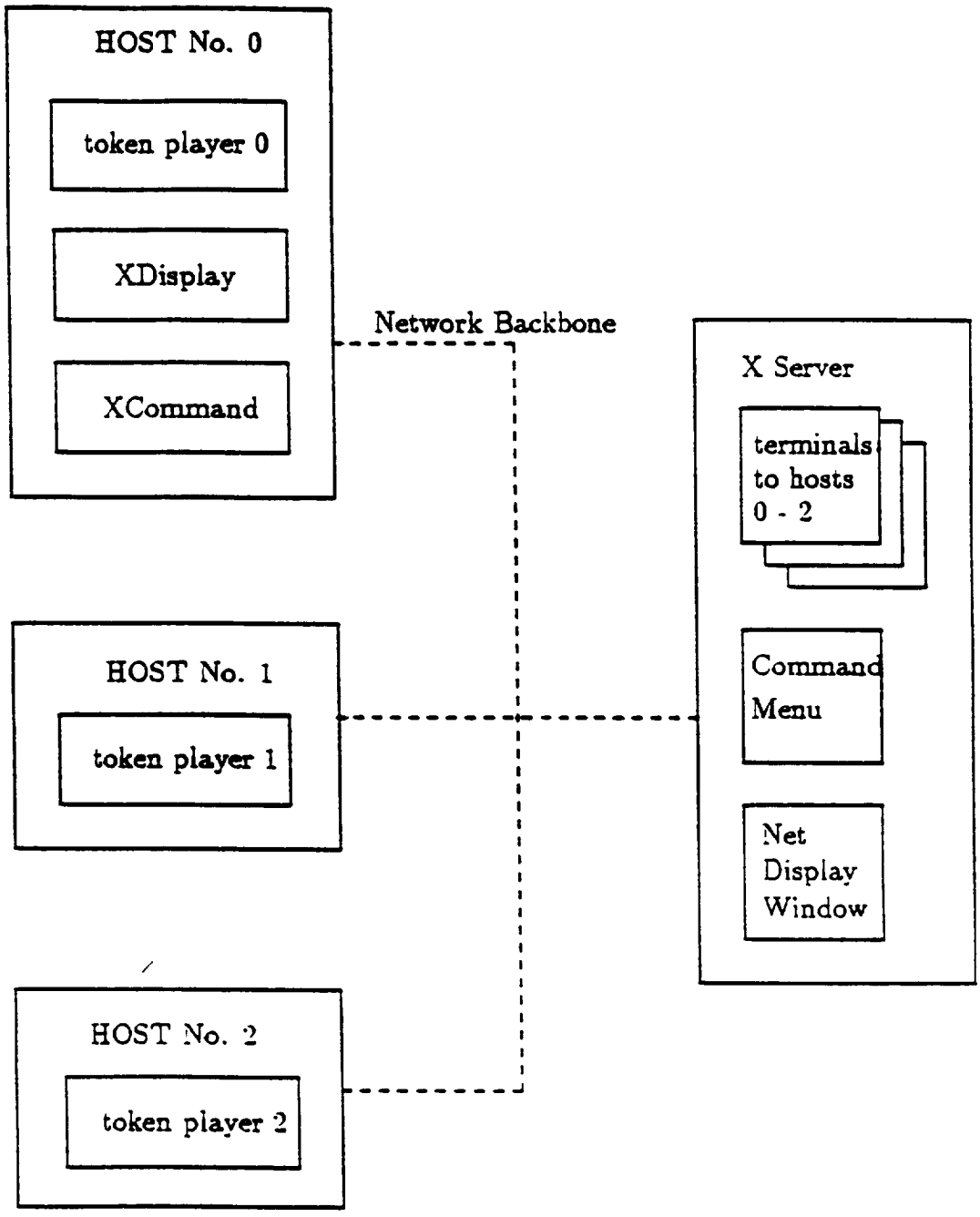

Figure 3.3: Host Computers with Associated Processes

The names listed at each computer is the Internet node name of the host computers used for running the testbed controller. All computers used are SUN SparcStation 1.

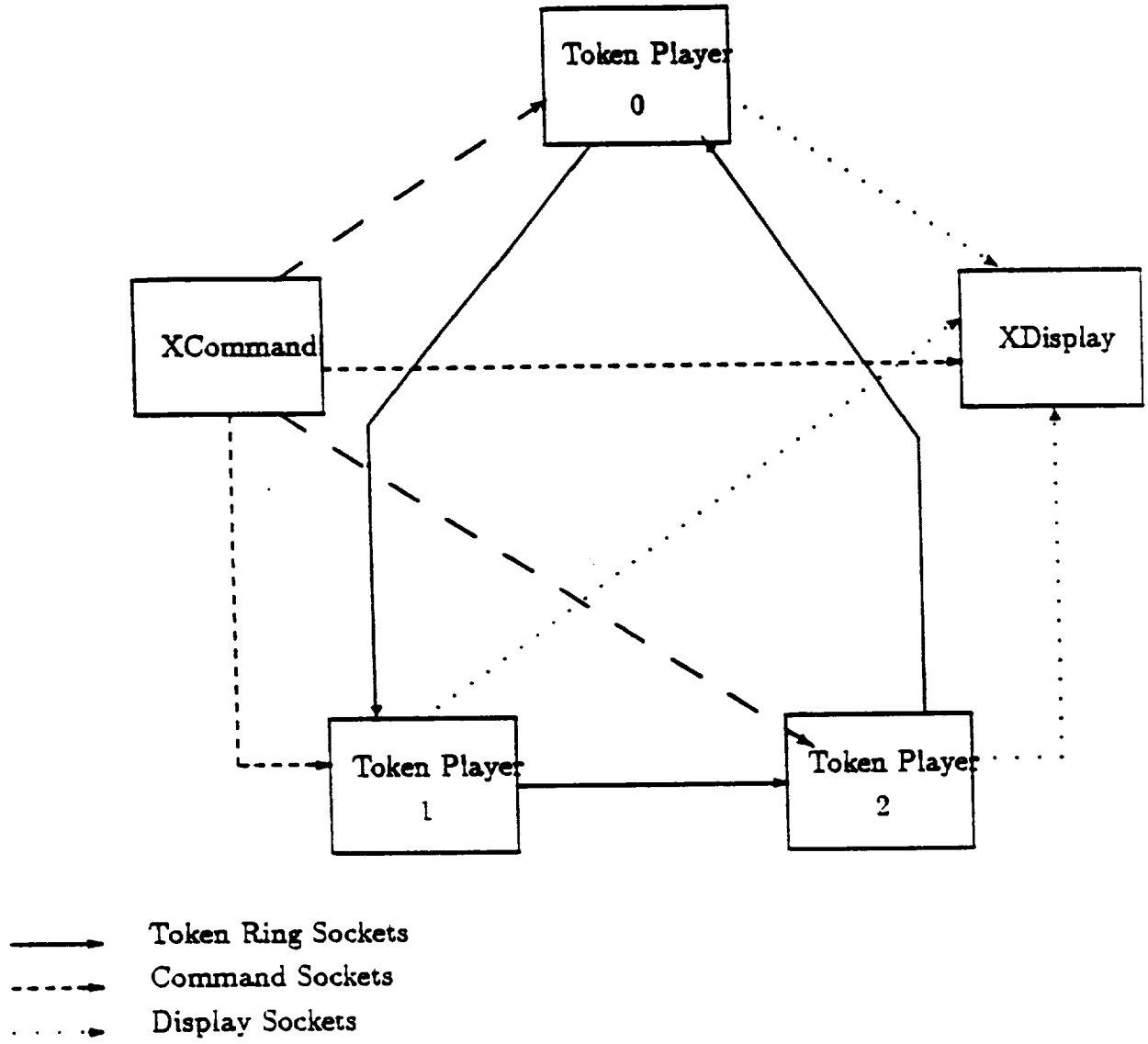

Figure 3.4: Socket Communication Structure

Not shown in this figure are socket connections to other processes that implement the device drivers at each host. Typically, one would have two unidirectional sockets for each device driver. Since the number of devices might be large, it would furthermore be advisable to write one device driver program handling all devices.

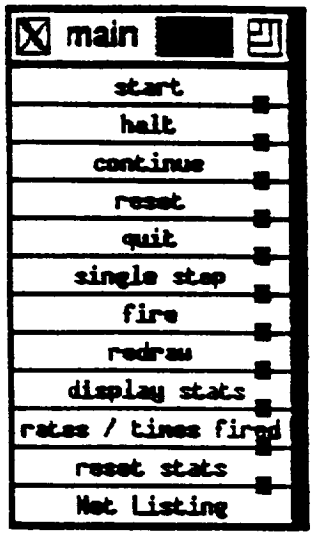

Figure 3.5: Command Window

3.8 X Interface for Controller Commands - *XCommand*

The X interface to control the execution of the net is implemented in the program *XCommand*. This program presents the operator with a menu (fig. 3.4) with 12 different buttons.

The buttons have the following effect:

Start Enables continuous firing of all enabled transitions at each token player. Pressing the start button also resets the transition firing counter and the timer showing how long since execution was started.

Halt Disables firing of all transitions. The timer continues running.

Continue Enable continued firing without resetting the timer and firing count.

Reset Resets the marking vector to the initial marking vector. Should *not* be used when the system is executing a net connected to any real devices.

Quit Stops all token players and the interface programs.

Single-Step Toggles single-stepping the firing of transitions. When **Single-Step** is enabled, only one transition at a time may fire (if any enabled) at each token player.

Fire Enables one transition to fire at each token player.

Display Stats Turns on the display of statistics at each transition.

Rate/Times Fired Toggles between the running average firing rate of each transition and displaying the number of times each transition has fired.

Reset Stats The timer and the firing count is set to zero.

Net Listing Dumps the current status of all data structures at each token player to the terminal window where the token player was started (stdout).

XCommand sends a command message to each token player and to the display program when a button is pressed. The Athena widget set is used to implement the X command menu.

3.9 X Interface for Displaying the Net - *XDisplay*

XDisplay is the program used for showing (in near real-time) the firing of the net in an X-window. The net is drawn as it was designed in GreatSPN, thus ensuring familiarity with the graphical representation of the net.

XDisplay is suspended until either an event occurs on the input sockets or a one-second timer interrupt is issued. A socket message from any of the token players or the command window causes the program to wake up and read the sockets. A socket message from a token player consists of a count of how many times each transition has fired, the local marking vector and the global marking vector.

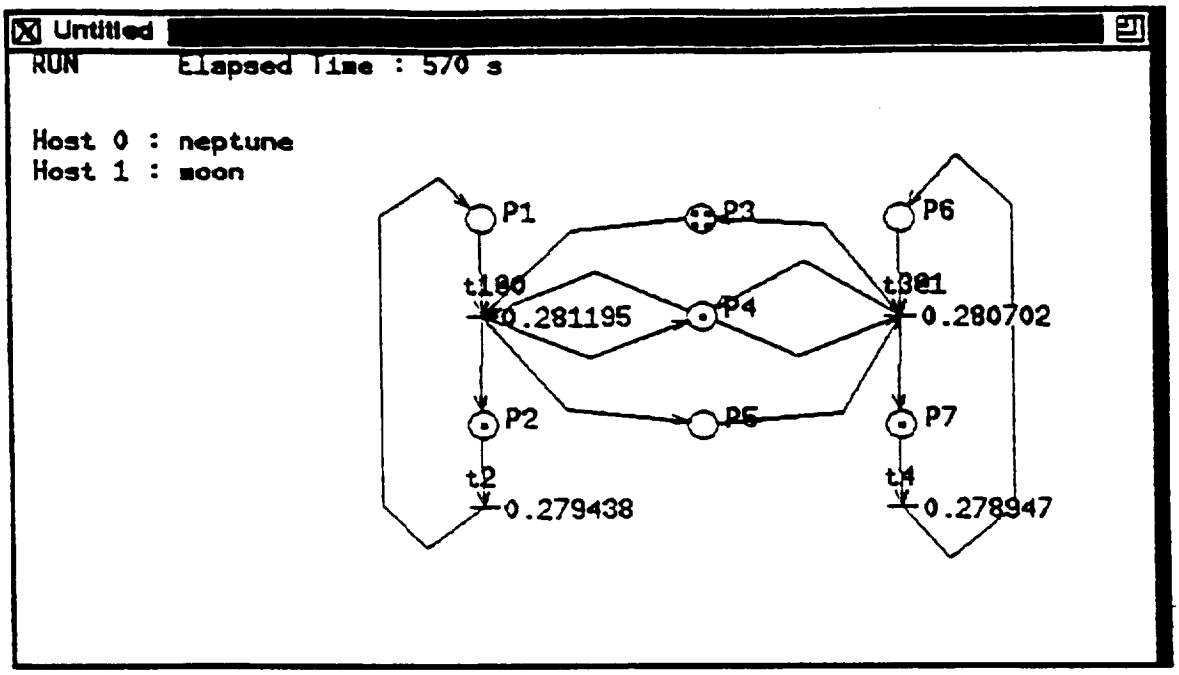

Figure 3.6: Display Window

Any transition firing counts that have changed since the last update cause those particular transitions to be highlighted for approximately 350 ms. The program is suspended for this time, but if a new socket message appears, the program wakes up and services this, updating the display as needed. The tokens are drawn in the center of each place as a dot. If there are more than 4 tokens in a place, the number is written directly for making it as easy as possible to read the marking.

When the net is executing, the elapsed time is displayed as well as the current status of the DPNC (running, halted or single-step mode). Statistics for all transitions may be displayed next to each transition. The host names also appear at the left side of the screen. In this case, host number 0 is `neptune` (one of the many stellar names used in the CIRSSE lab at RPI) and host number 1 is moon (Internet node names may be written in full: i.e. `neptune.ral.rpi.edu`).

The user may select which sub-net to display by pointing the mouse in the

display window and pressing the character 0 to toggle displaying the net on token player 0. 1 will toggle the display of token player 1's net and so on. G toggles the displaying of the global marking vector whereas A will turn on the displaying of the entire net if any nets are turned off.

3.10 'C' Language Implementation

3.10.1 Introduction

The entire controller is implemented in 'C' on Sun workstations using BSD4.2 UNIX operating system. The following is a description of each module comprising each of the main programs. A module is a file with one or more subroutines, usually logically connected.

3.10.2 net2n_m

This program is listed in Appendix A.1. The program consists of the following file:

net2n_m.c All supporting subroutines are found in this file. The program is compiled using *cc* only.

3.10.3 Assign

Assign Appendix A.2 contains the program listing for *assign*. A makefile *masg* is used for making the program with the following supporting files:

assign.c Main as well as all procedures to manipulate and assign the places and transitions are contained in this file.

syserr.c This routine is called in most places where the program performs I/O. If an I/O call should fail. this procedure will print the reason it failed and terminate the program.

3.10.4 Token Player

The token player is compiled using the makefile *mpn*. The following support files are used:

player.c The net execution routines as well as *main()* are implemented here. The socket initialization routine is also implemented in this file.

setblock.c A routine to set or reset blocking of a file descriptor (a file descriptor may point to a socket or a file).

server_intr.c This file contains the routine used to set up a server socket (client sockets connect to a server socket) with an interrupt signal to be issued when a socket message is received. This mechanism is used to wake up a program that has been suspended.

intr_timer.c A watchdog timer is used to wake up the token players in the case an interrupt is lost. This sometimes occurs when the workstation is heavily used. The timeout is set for 4 seconds when the controller is not used for simulation (no timed transitions) and for 0.1 second when there are timed transitions.

client.c This procedure is used to connect to an already open server socket.

timed_trans_handler.c The routines used to handle timed transitions are contained in this file. These start and check the timers for each timed transition during net execution.

timer.c The system clock is initialized and read using the routines in this file.

xrand.c This file contains routines for random number generators and has been obtained from the same source as *net2n.c*. The random number generator is used for simulating stochastic firing times of exponential transitions.

event_handler.c External communication with other token players and with the command program is implemented here.

init_net.c Each token player executes a sub-net. These nets are defined in this routine using an include file *netni* which initializes the data structures.

interface.c Device drivers are implemented in this routine.

dump.c The net listing is done by calling this routine.

3.10.5 XCommand

The command program is compiled using the makefile *mXc*. The program is compiled using the following support files:

XCommand.c This file contains *main()* and the procedures associated with each button in the menu.

setblock.c, client.c, syserr.c have already been described.

3.10.6 XDisplay

The net display program consists of a large number of support files for implementing the X interface as well as routines to decode and display the firing vectors received from the token players. The program is built using the makefile *mXd*.

XDisplay.c This file contains *main()* as well as routines to read the net structure, read and draw the geographical information and read the sockets.

Xdraw.c All primitive drawing routines are contained in this file. The sizing of the window, symbol sizes etc. may be scaled by changing the constants in em Xdraw.h. the include file used by the drawing routines.

Xroutines.c Support routines for initializing, opening and exiting X are contained in this file.

stats.c The running averages are computed and displayed using routines in this file.

eventx.c X events (window size changes. overlaps etc.) are handled by this routine.

timer.c, setblock.c, server_intr.c, intr_timer.c, syserr.c are described previously.

3.11 Summary

The internal workings of the DPNC have been described along with the files and programs used in the controller. It should be noted that a large number of data and program files are generated. This allows not only for custom tailoring of the controller, but also creates a need on the designers part to carefully document the implementation of a specific controller.

CHAPTER 4
USING THE DISTRIBUTED PETRI NET CONTROLLER

4.1 Introduction

A successful implementation of the controller requires careful thought to how the DPNC interfaces to the real machines and devices it is supposed to control. The Petri net itself must also be designed to facilitate decentralized control. This chapter discusses how the DPNC may be used and what issues need to be addressed.

4.2 Special Petri Net Design Considerations

The DPNC places a few restrictions on how the Petri net control logic is designed. Firing order is not random, but each transition is evaluated in a specific order. Thus, if there is a choice place (one place is the enabling input place to two or more transitions), one transition will always fire first and consume the token(s), thereby preventing the other transitions from firing.

Firing order is also dependent on how the net is partitioned and how the token players are arranged. Since the global marking vector is passed from token player to token player, the control logic embedded in the net must take the order of receiving the marking vector into consideration.

Finally, the number of global places should be minimized. The best method of doing this is to limit global places to represent only the exchange of information or resources between different token players. The more global places, the more transitions will be dependent on the global marking vector and the controller will in effect become less decentralized.

4.3 Design using GreatSPN

The basis of any distributed controller is a Petri net model of the control logic made using GreatSPN [2]. GreatSPN provides graphical tools to design and analyze Petri nets on SUN workstations using Suntools[1].

It is also possible to obtain performance data for Petri nets. Information such as transition firing rates and token probability distributions will give the analyst insight into the performance on the modeled system. This is of course only possible if performance data are known for the individual machines and devices in the system to be controlled. It is also possible to translate the GreatSPN file to a SPNP[3][4] file for further performance analysis.

4.4 Input / Output using the Controller

The DPNC facilitates I/O by linking procedures to transitions. A transition may have an enabling procedure and a firing procedure. The enabling procedure (called *precondition*) is a subroutine that returns true when a particular external condition holds true. Conversely, the routine must return false if the condition is not true. The subroutine is recommended to be short as it will be executed every time the transition is otherwise enabled. It is therefore not advisable to use a precondition subroutine to directly query a device using sockets, as that would introduce a 400 ms delay for each query. The preferred method is to use an external program that sends a status change to the token player whenever a change occurs. The precondition routine may then only check for a particular bit-pattern to be set.

The external program must communicate with the token player using sockets(fig 4.1). The socket mechanism allows the token player to be event driven. The token player will then be woken up whenever the device driver reports a change in status of a machine if the token player is suspended.

[1] An X version is reportedly being prepared

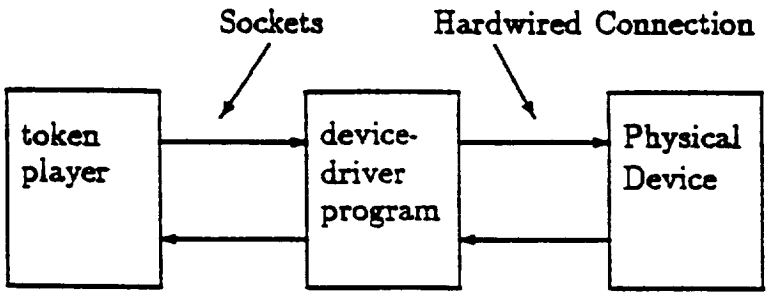

Figure 4.1: Device Driver I/O

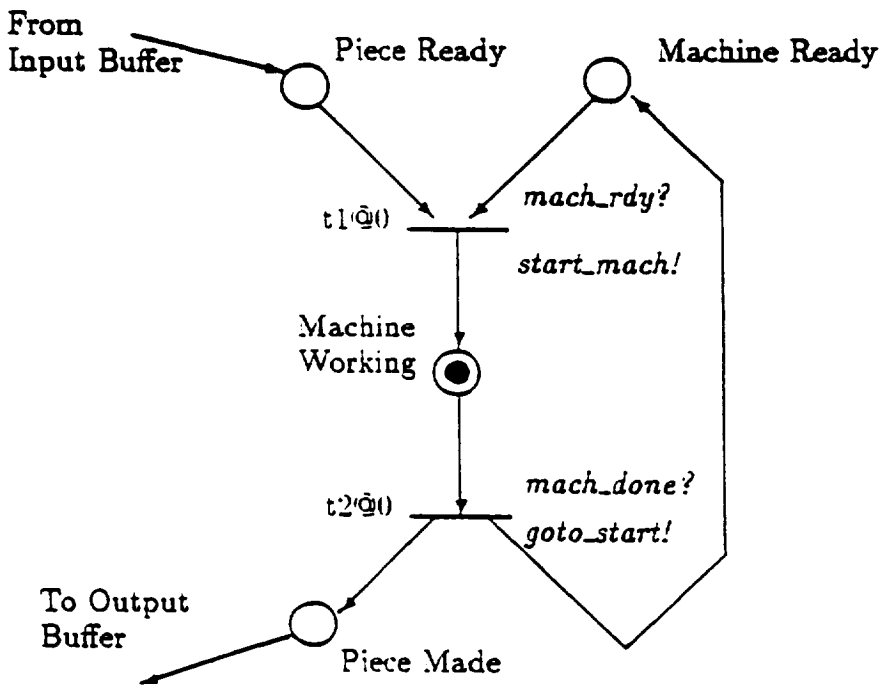

Figure 4.2: Typical Use of Transition I/O

A design example for controlling the operation of one machine is shown in figure 4.2 where a machine is started when a workpiece is present and the machine is ready. The machine is ordered to return to start position when it is finished producing a piece. Note the italicized strings next to the transitions represent the precondition (*?* appended to string) and the firing process (*!* appended to string). This provides an easy method of labeling the transitions. Also note the *@0* appended to the transition names assigns the transitions to host number 0.

4.4.1 Input using Transitions — Example

To implement this interface, a procedure for checking the machine status and a procedure for issuing a command to a machine are needed. These may be written as follows:

```
int check_status(trans_no, mach_no)
int trans_no;
int mach_no;
{
     switch (trans_no) {
          case 1 :
               if (M_STATUS[mach_no] && M_RDY) return 1;
               else return 0;
               break;
          case 2 :
               if (M_STATUS[mach_no] && M_DONE) return 1;
               else return 0;
               break;
     }
}
```

To link this procedure to the transition *t1@0* in figure 4.2, the line `Transition[0].precondition = check_status;` should be added in the file *tr_linkn.i*. This line would assign the pointer address to that procedure. The number assigned to the transition at each token player is found in the *netn.i* files. The procedure is written as general as possible with `M_STATUS` being a global array of status words for all machines handled by the token player. `M_RDY` and `M_DONE` are constants with the bit corresponding to that condition set. It is assumed that another procedure sets and resets these flags when appropriate. This procedure would be called every time a machine issued a status change socket message. A typical implementation might be :

```
void set_status()
{
     int buf[10], nread;
     if (nread = read(in_socket, buf, length(buf)) = -1)
          syserr("in_socket")
     M_STATUS[ buf[0] ] = buf[ 1 ];
}
```

The above code segment assumes that `buf[0]` contains the machine or device number and `buf[1]` the status word. For a SUN workstation, this particular implementation would allow status 32 bits per machine (since each `int` is a 32 bit word). The designer may use several words per device as needed, thus expanding the number of bits.

4.4.2 Output using Transitions — Example

To issue commands to devices, *interface.c* could contain a routine to issue commands as follows:

```
void send_command(trans_no, mach_no)
BOOLEAN trans_no;
int mach_no;
{
     int comd;
     switch (mach_no) {
          case 1 :  /* -- Machine number 1 */
               switch (trans_no) {
                    case 1 :  comd = 12; break;
                    case 2 :  comd = 14; break;
                    ...
                    case n :  comd = x; break
               }
          ...
     }
     buf[0] = mach_no;
     buf[1] = comd;
     if (write(out_socket, buf, length(buf)) = -1 )
          syserr("write");
}
```

The pointers of each output transition need to be initialized similarly to the input pointers. In addition to the procedures above, the procedure *init_sockets* in *player.c* need to include code to initialize the sockets for communicating with the device driver program.

4.5 Building the Token Players

The controller design starts with the Petri Net obtained from GreatSPN. The host dependent transitions are assigned using the *@host* convention. Driver software for the specific devices must be added next in the file *interface.c* with appropriate pointer links to the I/O procedures in the file *tr_linksn.i*. The final step is to execute `build` *name* where *name* is the given net name in GreatSPN.

4.5.1 Building a Distributed Controller

The DPNC is designed for distributing a Petri net over several host computers. To facilitate this, *build* will prompt the user for each host's Internet name. When all host names have been entered, the user should hit `return` and *build* will then proceed to make all files and executable units for the controller.

4.5.2 Building a Single-Processor Controller

The DPNC may also be used as a single-host controller by not assigning any transitions a higher host number than 0. Socket communication for passing the global marking vector will then be disabled. All processes (token player, XCommand and XDisplay) may then be run on one computer. The programs should be started in the same sequence as listed in chapter 3.

4.6 Using the Controller to Simulate Petri Nets

To facilitate simulation and verification of a Petri net model, it is possible to use the controller as a simulator. Immediate transitions may be converted to deterministic or stochastic using GreatSPN[2] with firing rates specified by the user. The example used in the I/O section (fig. 4.2) is shown in figure 4.4 with timed transitions. The numbers next to the transitions in figure 4.4, `10.0000` and `0.35000`,

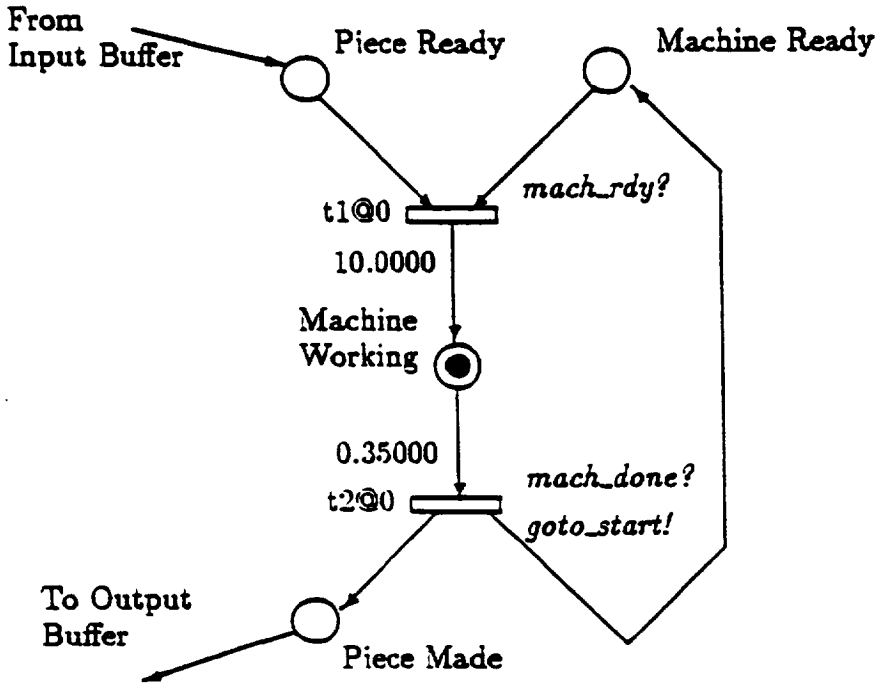

Figure 4.3: Timed Transition Simulation

are the firing rates assigned to these transitions. These reflect the expected time for the machine to go to the ready position and to process one piece, respectively.

Note that when using the system as a simulator, all transition procedure links must be commented out in *tr_linksn.i* to disable using the device driver interface routines. The transitions that were made timed for testing purposes must be made immediate when using the controller again.

4.7 Using the Controller

A few words of caution would be appropriate when using the controller with any real machines requiring fail-safe operation and emergency stop: Any emergency stop must be implemented in hardware (i.e., kill power to electric motors, engage brakes etc.). No operation required to be fail-safe should be implemented using this

controller (or any other for that matter). The controller introduces a lag of approximately 200 ms per token player in the token ring. Thus, any device requiring immediate action when its status changes should be implemented using local places only. Transitions requiring the global marking vector will experience the aforementioned delay before they may fire. Note that the socket mechanism may be modified to use datagrams or raw sockets which are much faster (there is very little delay) but do not have the TCP/IP delivery reliability.

The next chapter will describe one application of the DPNC in controlling a model of a robotic testbed for assembly of structures in space.

4.8 Summary

This chapter has attempted to describe interfacing the controller to an application using sockets and dedicated driver programs that handle the physical input/output signals. The next chapter describes a test case where these concepts have been put into use.

CHAPTER 5
TEST CASE — CIRSSE TESTBED CONTROLLER

5.1 Introduction

The previous two chapters described the internal structure of the controller and how to interface it to external systems. This chapter gives an example of how to actually control a system using the DPNC. The system chosen is the NASA CIRSSE (Center for Intelligent Robotic Systems for Space Exploration) testbed at Rensselaer Polytechnic Institute. The Petri net model for the system is found in [8].

To focus the design example on the actual controller and not the specific device drivers for any particular machine, several DPNCs are used; one acts as the central controller distributed across three host computers. One DPNC at each host acts as a simulator and device driver of the machines and devices connected to the token player at that particular host. These three DPNCs are single token players only, executing simple Petri net models of the actual machines. Figure 3.1 gives a picture of this setup with the processes marked *Device Driver* n representing single token player simulators.

5.2 Dual Arm Testbed

The system contains 2 PUMA robots, a mobile platform on which the robots are mounted, a vision system and 3 host computers for path planning and generation, and robot control. The host computers are tied together via Ethernet. The purpose of the testbed is for research of autonomous assembly of large space structures for the NASA orbiting space station "Freedom".

5.3 Petri Nets Describing the Testbed and Simulators

The Petri net model of the testbed control system (figure 5.1) is partitioned into three subsystems: Operator interaction and offline path generation, vision system, and robot and platform control. Each subsystem uses one host computer, a SUN4, a SUN3/260 and a SUN3/150 respectively. The transitions modeling each subsystem are preassigned using the notation described in chapter 3 (appending *@h* where *h* is the host computer number). There are six global places: *SUN4, OnLinePlan, OffPredo?, Traj5, TrajReq6* and *TrajReqP*. These places are shared between transitions assigned to different host computers. The partitioning can be seen in figure 5.1 by studying where the transitions are preassigned. The upper third of the net executes on token player 0 and the middle part of the net (the vision system) executes on token player 1. The lower third of the net executes on token player 2.

To illustrate the use of external device driver programs, two or three stochastic transitions in each subsystem are modeled and simulated by the device driver/simulator Petri nets at each token player.

5.4 Implementation of the Controller

The third token player, *tp2*, will be discussed as an example of how to interface a token player to an actual device. The original Petri net was modified slightly to facilitate control of the devices. Three stochastic transitions model the two robots and the platform, *T23–T25*. These were expanded from one transition each, to one output transition to issue a start command, one place to indicate machine working, and one transition to query if the machine was done (see figure 5.1). The transitions were called *T2iStart!@2* and *T2iDone?@2* for i = 3, 4 and 5 respectively.

The *Start!@2* transitions signal to the device that the operation may start, while the *Done?@2* transitions only fire when the devices signal that they are done.

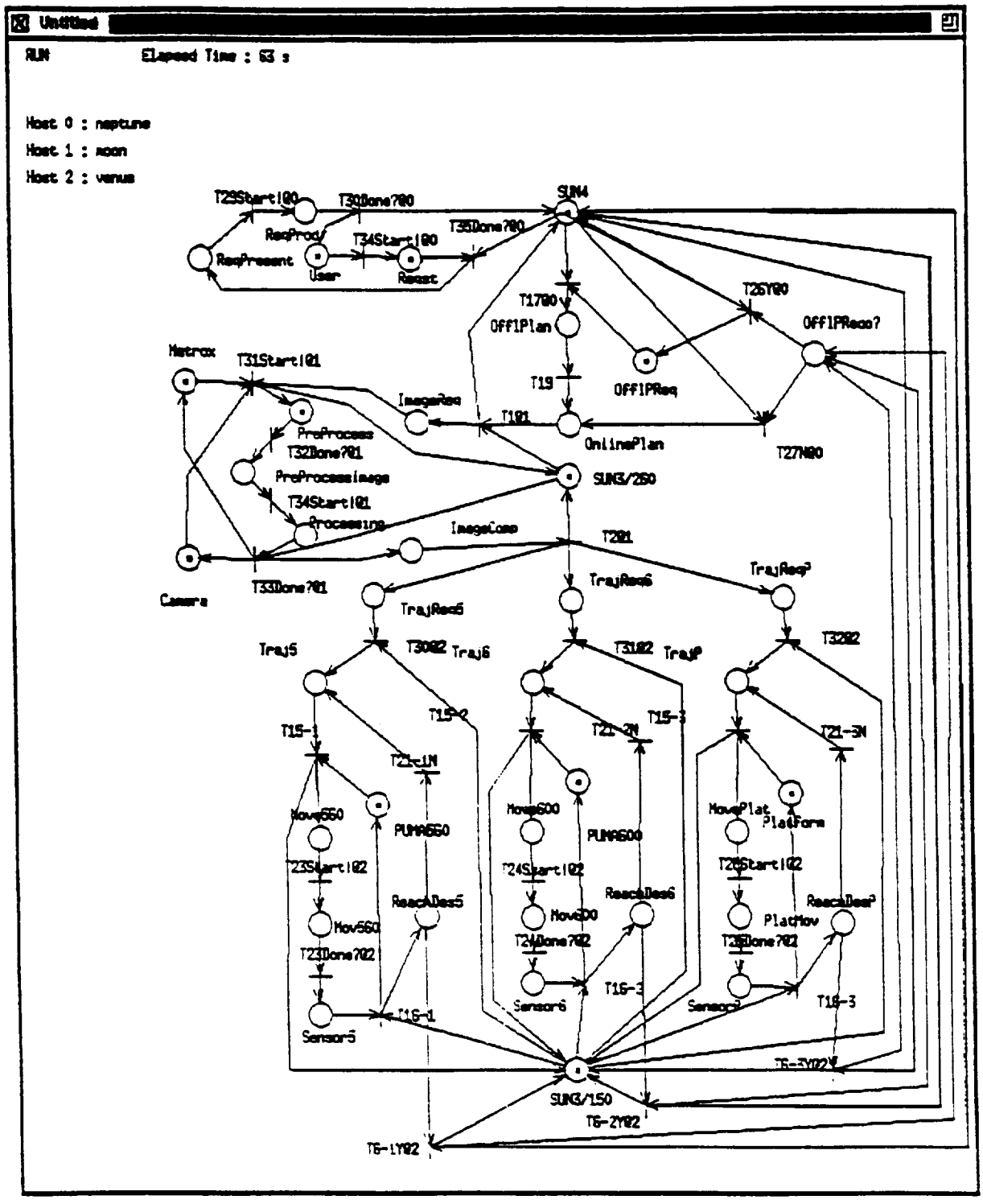

Figure 5.1: CIRSSE Testbed Petri Net Controller

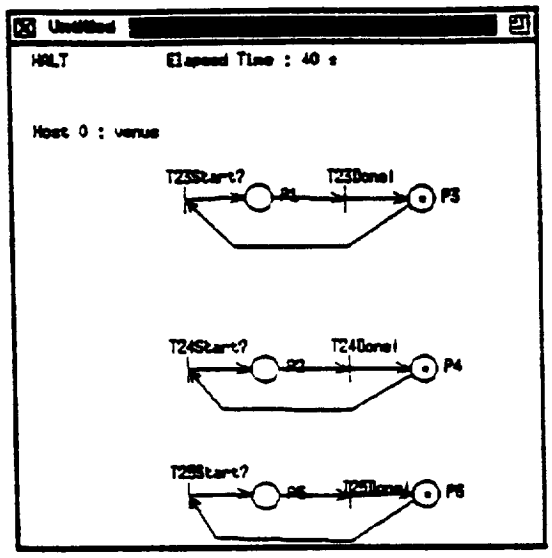

Figure 5.2: Simulator Petri Net for SUN3/150 Devices

5.5 Implementation of Simulators

To simulate the two robots and the platform, a simple Petri net was designed and implemented as a single token player simulator at each host. For host number 2, this net consists of 6 transitions and six places (figure 5.2). The transitions *T23Start?* are enabled by the start signal from the controller transitions described previously. The transitions *T23Done!* signal to the controller that the devices are done. These latter transitions have stochastic firing rates modeling the time it takes to perform the action.

5.6 Discussion of 'C' Code

Since four different DPNCs were used, three subdirectories were made under the directory containing the controller. Each subdirectory contained one simulator for one host. Appendix B. contains listings of the code written to implement simulator/device driver number 2. This code was placed in subdirectory *SIM_2*.

Note that the code for the controller is numerated *2* while the code for the simulator is enumerated *0* in the appendix. The reason is that the controller token player is number 2, while there is only one simulator token player at each host, thus giving that simulator the number *0*.

5.6.1 Controller Code

A brief discussion of the controller code for implementing the interface to the device driver/simulator is included here. For further information, please study the procedures themselves.

tname2.h This file is an include file containing the name of the transitions of token player 2 with the transition number at that token player. By including this file in the device driver program file, *interface2.i*, it is possible to reference a transition by name (which does not change), instead of by number (which does change according to how the Petri net is partitioned and modified).

interface2.i The procedures for testing preconditions, for executing postprocesses and decoding the input from the device driver are contained in this file. Also note the procedure *conflict_resolution* which returns true or false with the probability given by the transition firing rate assigned in GreatSPN (note — this limits the rate number between 0 and 1 and is only meant to be used for immediate transitions).

tr_links2.i This file is the include file that sets up the links for the transitions to call the appropriate procedures described above.

5.6.2 Simulator # 2 Code

The simulator can easily be modified to interface with real machines. The I/O procedures would need to be modified to include communication mechanisms

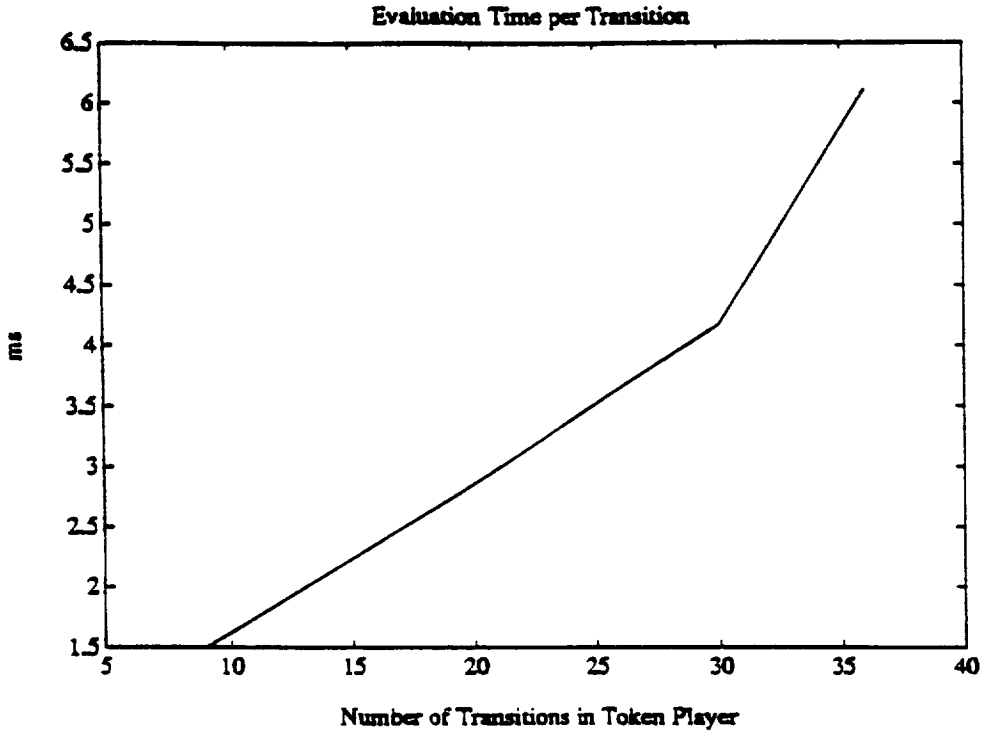

Figure 5.3: Transition Evaluation Time

A study of the time to evaluate a transition was done by executing several nets on a SUN SparcStation 1. The nets were all a sequence of places and transitions with the first place marked with 100 tokens. The time to fire all transitions 100 times was recorded and averaged over 5 runs. Figure 5.3 shows how the firing time varies with the number of transitions in the net. From 10 to 32 transitions, the relationship is linear. For more than 32 transitions, the curve becomes steeper. The reason is that the data structure required to store the token marking mask is a 'C' `long int` which is 32 bits long. For more that 32 places, two words are required to store the mask and this accounts for the increased evaluation time per transition.

The controller is very CPU friendly since it is event-driven. The average CPU time consumed as shown by `ps` is less than one percent. For this net, approximately one percent of available memory on a SUN Sparcstation 1 (8 MB RAM) is consumed.

to the actual devices, and the command and status messages would also need to be encoded for the particular device.

tname0.h contains the transition numbers as discussed above. Note that *assign* modifies the names to conform with 'C' constant declaration rules (described in the comments in *assign.c*).

interface0.i The procedures are the same as for the controller, except that the transition names are for the simulator. Note that the file *../tname2.h* is # included. This allows the procedure writing to the controller to specify which transition should be signalled to by name instead of by number.

tr_links0.i Same as above, but for the transition names of the simulator.

init_sockets() The socket initialization is different than for the controller token player and thus listed in the appendix. This procedure belongs in the file *player.c*.

get_event() Code for reading the socket to the controller token player is unique to the token player and thus listed here.

5.7 Socket Communication between Controller and Simulators

Each subdirectory containing a simulator has a unique *portnums.h* file. These files have socket port numbers for the simulators that are different than the port numbers used by the controller token players.

5.8 Net Execution and Controller Performance

The controller was found to control the operation of the (simulated) devices as planned. The system executed one testbed subtask in about 15.6 seconds using the firing rates given in [8]. This is very similar to the results obtained in [8].

5.9 Summary

A controller has been implemented for a robotic testbed using a Petri net model to facilitate the control logic. Petri net models of the machines have been used to implement simulators for the same machines, thereby demonstrating the use of device driver programs.

CHAPTER 6
CONCLUSIONS AND FUTURE DIRECTIONS

6.1 Conclusions

A distributed Petri net based controller has been designed and implemented. The major features of the system are:

1. Seamless integration of the controller with GreatSPN for generating Petri net data files.
2. Use of generalized stochastic Petri nets to allow performance analysis using known algorithms and software packages such as SPNP.
3. Distributed processing facilitating the concurrency and asynchronous features of Petri nets.
4. X Window graphics used to display the actual execution of the net.

6.2 Future Directions

This project has attempted to address the basics of a Petri net controller using as many available tools and programming standards as possible. Each choice has then, necessarily, been a compromise between speed of execution and speed of implementation. Speed of implementation has been prioritized in order to reach the goals listed in chapter 2 in a finite amount of time.

There are a number of improvements that may be made to further enhance the utility and practicality of this project. Among these are:

Token Coloration One immediate improvement would be to include colored Petri nets. This would also increase the complexity of net analysis (since the colored net would need to be unfolded).

Automated Startup Sequence This sequence may be automated so that the operator only would execute one command, instead of 2 + n-1 for an n-1 node token player.

Selective Enabling/Disabling and Firing of Transitions A menu could be presented to the user where each transition could be disabled, enabled and fired manually.

Improved File Naming Convention Data files should have the Petri net name included in the file name with file name extensions indicating what kind of file it is, e.g. *controller1.dspy* could be the initialization file for the display program for the Petri net *controller1.net*.

Faster Socket Communication As it is now, a 200 ms delay is introduced by each token player in the virtual token ring chain. This is due to the UNIX scheduler which handles TCP/IP messages. The handler, or daemon, has a turnaround time of 200ms. Other mechanisms exist that do not have this delay (raw sockets and datagrams), but these do not have the guaranteed delivery of messages that TCP/IP offers.

Using GreatSPN Layers It would be possible to use the layer mechanisms of GreatSPN. One possible use would be to have the controller net in one layer and models of the devices in different layers. Simulations could then be performed using GreatSPN or SPNP for the entire system, and the controller structure could be extracted for the distributed Petri controller implementation.

LITERATURE CITED

[1] T. Murata, "Petri Nets: Properties, Analysis and Application" *Proceedings of the IEEE*, vol. 77, no.4, April 1989.

[2] G. Chiola. GreatSPN User's Manual, Version 1.3, February 1989.

[3] G. Ciardo, "Manual for the SPNP Package", Duke University, July 1988.

[4] J. Peck. GreatSPN2SPNP Manual Page, Electrical, Computer and Systems Engineering Department. Rensselaer Polytechnic Institute, August 1990.

[5] D. Crockett. "Manufacturing Workstation Control Using Petri Nets", Master's Thesis. Electrical. Computer and Systems Engineering Department, Rensselaer Polytechnic Institute. August 1986.

[6] E. Kasturia. "Real Time Control of Multilevel Manufacturing Systems Using Colored Petri Nets". Master's Thesis. Electrical, Computer and Systems Engineering Department, Rensselaer Polytechinc Istitute. May 1988.

[7] D. Rudolph. "Petri Net-Based Control of a Flexible Manufacturing System", Master's Thesis. Electrical. Computer and Systems Engineering Department, Rensselaer Polytechinc Istitute. May 1989.

[8] J. Robinson. "Performance Analysis of a Robotic Testbed Control Architecture", Master's Thesis. Electrical. Computer and Systems Engineering Department. Rensselaer Polytechnic Institute, December 1989.

[9] J. Ayache. J. Couriat and M. Diaz. "REBUS. a Fault Tolerant Distribution System for Industrial Real Time Control". *IEEE Transactions on Computers*, Vol C.-31. No. 7. July 1982. pp. 637–647.

[10] M. Courvoisier. R. Valette. J.M. Bigou. and P. Esteban. "A Programmable Logic Controller Based on a High Level Specification Tool." *Proceedings of the IEEE IECON Conference on Industrial Electronics,"* IEEE Press, New York. 1983. pp. 174–179.

[11] D. Chocron and E. Cherny. "A Petri-Net Based Industrial Sequencer." *Proceedings of the IEEE International Conference and Exhibition on Industrial Control and Instrumentation*. IEEE Press. New York. 1980, pp. 18–22.

[12] T. Murata. N. Komoda. and K. Matsumoto, "A Petri-Net Based FA (Factory Automation) Controller for Flexible and Maintainable Control Specifications." *Proceedings of the IEEE IECON Conference on Industrial Electronics*. IEEE Press. New York. 1984. pp. 362–366.

[13] M. Kamath, N. Viswanadham, "Applications of Petri net Based Models in the Modeling and Analysis of Flexible Manufacturing Systems," *Proceedings of the 1986 IEEE International Conference on Robotics and Automation*, 1986, pp. 312–317.

[14] N. Komoda, K. Kera, T. Kubo, "An Autonomous, Decentralized Control System for Factory Automation," *IEEE Computer Magazine*, December 1984.

APPENDIX A
'C' SOURCE CODE LISTINGS

A.1 *build* — Building the controller

```
#!/bin/sh
# Use Bourne shell

echo
echo "Distributed Petri Net Control System"
echo
# Test if a net name has been passed as an argument
if [ $# -ne 1 ]
then
  echo "Usage: build <Petri Net Name>"
  exit
fi

# Test if this file exits
if [ ! -f $HOME/greatspn/nets/$1.net ]
then
  echo "Petri Net $1 does not exist"
  exit
fi

# Test if hostnames file exists, if not, prompt for new file
if [ -f hostnames ]
then
  echo "Current hostname assignments are :"
  cat hostnames
  echo
  echo "Hit <Return> to keep current assignments"
  echo "or enter new information at the prompt"
  echo
else
  echo "Hostname file definitions does not exist"
  echo
fi
echo "Enter host information in the following format:"
echo "0 name <Return>     /* For host 0 /*"
echo "1 name <Return>     /* For host 1 /*"
echo "  ."
echo "  ."
echo "n name <Return>     /* For the last host /*"
echo "<Return>            /* To stop or keep current values/*"
echo
hostno=0
echo "Hostname $hostno :"
read inp
if [ "$inp" != "" ]
then
  if [ -f hostnames ]
  then
    rm hostnames
  fi
fi
until [ "$inp" = "" ]
```

```
do
  echo $hostno $inp >> hostnames
  hostno=`expr $hostno + 1`
  echo "Host $hostno : "
  read inp
done

if [ ! -f hostnames ]
then
  echo "FATAL - hostname definition file does not exist"
  echo
  exit
else
  echo
  echo "Current hostname assignments are :"
  cat hostnames
  echo
fi

# Convert .net file to .n file
echo
echo "Converting .net file to .n file ..."
echo
unset noclobber
mv test.n test.n~
mv test.net test.net~
cp $HOME/greatspn/nets/$1.net test.net
cat test.net | net2nm > test.n

rm net*.i
# Assign places and transitions, generate net definition files
echo
echo "Generating net definition files ..."
assign

# Compile XDisplay program
echo
echo "Compiling XDisplay server ..."
make -f mXd

# Compile XCommand program
echo
echo "Compiling XCommand client ..."
make -f mXc

# Compile token players
i=0
while [ -f net$i.h ]
do
   echo
   echo "Compiling token player $i ..."
   rm player.h
   echo "#include        \"net$i.h\"" > player.h
   cat master_player.h >> player.h
   make -f mpn
   mv pn pn$i
   i=`expr $i + 1`
done
echo
echo "done"
```

A.2 *net2n_m* — Extracting the Net Structure

```
/*************************************************************************\
*                                                                         *
*  GS: A Generalized, stochastic petri net Simulator                      *
*      V0.01       March 1989      Andreas Novatzyk (agn@unh.cs.cmu.edu)  *
*      Carnegie-Mellon Univerity, School of Computer Science              *
*      Schenley Park, Pittsburgh, PA 15213                                *
*                                                                         *
*      MODIFIED 1/3/91 by Atle Bjanes to acommodate inhibitor arcs of     *
*                      multiplicity greater than 1.                       *
*                                                                         *
\*************************************************************************/

/* Converts GreatSPN's net-files into a more resonable format */

#include <stdio.h>
#define TTY_IMM                 0                    /* transition types  */
#define TTY_EXP                 1
#define TTY_DET                 2

char        *trns_types[3] = {"imm", "exp", "det"};

/******************** Data structures ********************/

struct place {                                       /* A place is... */
    char                *name;
    int                 tokens;
};
struct place_list {                                  /* list of places */
    struct place        *pl;
    struct place_list   *next;
};
struct transition {                                  /* A transition is... */
    char                *name;
    unsigned char       type;
    double              rate;
    int                 dep;                         /* enabling dependency */
    struct place_list       *inpts;
    struct place_list       *outpts;
    struct place_list       *inhibs;
};
struct parameter {                                   /* parameters */
    char                *name;
    double                deflt;
};

/******************** Net storage ********************/

struct place                *PLACES;
int                         n_PL;
struct transition           *TRANSITIONS;
int                         n_TR;
struct parameter            *RATES, *MARKS;
int                         n_RT, n_MK;

main(argc, argv)                                     /* boring stuff */
    int         argc;
    char *argv[];
{
     read_net();
     print_net();
```

```
}

read_net ()                                          /* read a network */
{
    register int    i, j;
    int             n_GR, k, l, m;
    register struct place_list *t;
    char            buf[1024], tmp[1024];

    while (gets(buf))                                /* skip preamble */
        if (buf[0] == '|' && buf[1] == 0)
            break;

    if (!gets(buf) || 5 != sscanf(buf, "%*s%d%d%d%d%d", &n_MK, &n_PL, &n_RT,
&n_TR, &n_GR) || n_MK < 0 || n_PL < 1 || n_RT < 0 || n_TR < 1
|| n_GR < 0) err("Bogus parameters");
    /* allocate storrage */
    if (n_MK) MARKS = (struct parameter *)
      malloc(n_MK * sizeof(struct parameter));
    if (n_RT) RATES = (struct parameter *)
      malloc(n_RT * sizeof(struct parameter));
    PLACES = (struct place *) malloc(n_PL * sizeof(struct place));
    TRANSITIONS = (struct transition *)
      malloc(n_TR * sizeof(struct transition));

    for (i = 0; i < n_MK; i++) {                 /* read mark-parameters */
        if (!gets(buf)) err("Premature end of file");
        if (2 != sscanf(buf, "%s%lf", tmp, &(MARKS[i].deflt)))
            err("Param problem");
        strcpy(MARKS[i].name = (char *) malloc(strlen(tmp) + 1), tmp);
    }
    for (i = 0; i < n_PL; i++) {                 /* read places */
        if (!gets(buf)) err("Premature end of file");
        if (2 != sscanf(buf, "%s%d", tmp, &(PLACES[i].tokens)))
            err("Place def problem");
        strcpy(PLACES[i].name = (char *) malloc(strlen(tmp) + 1), tmp);
    }
    for (i = 0; i < n_RT; i++) {                 /* read rate-parameters */
        if (!gets(buf)) err("Premature end of file");
        if (2 != sscanf(buf, "%s%lf", tmp, &(RATES[i].deflt)))
            err("Param problem");
        strcpy(RATES[i].name = (char *) malloc(strlen(tmp) + 1), tmp);
    }
    for (i = 0; i < n_GR; i++)                           /* skip groups */
        if (!gets(buf)) err("Premature end of file");

    for (i = 0; i < n_TR; i++) {/* read transitions */
        if (!gets(buf)) err("Premature end of file");
        if (5 != sscanf(buf, "%s%lf%d%d%d", tmp, &(TRANSITIONS[i].rate),
 &(TRANSITIONS[i].dep), &k, &l) || l < 1)
            err("Transition def problem");
        switch (k) {
        case 0:
            TRANSITIONS[i].type = TTY_EXP;
            break;
        case 1:
            TRANSITIONS[i].type = TTY_IMM;
            break;
        case 127:
            TRANSITIONS[i].type = TTY_DET;
            if (TRANSITIONS[i].rate < 1e-10)
```

```
                err("0-delay, deterministic transition");
            TRANSITIONS[i].rate = 1.0 / TRANSITIONS[i].rate;
            break;
        default:
            err("Unknown transition type");
        }
        strcpy(TRANSITIONS[i].name = (char *) malloc(strlen(tmp) + 1), tmp);
        TRANSITIONS[i].inpts = 0;
        TRANSITIONS[i].outpts = 0;
        TRANSITIONS[i].inhibs = 0;

        for (j = 1; j--;) {                          /* read inputs */
            if (!gets(buf)) err("Premature end of file");
            if (3 != sscanf(buf, "%d%d%d", &k, &l, &m) || !k || --l < 0 || l
>= n_PL || m < 0)
                err("Transition input def problem");
            if(k < 0) k = -k;

            while (k--) {
      t = (struct place_list *) malloc (sizeof(struct place_list));
      t->pl = &PLACES[l];
      t->next = TRANSITIONS[i].inpts;
      TRANSITIONS[i].inpts = t;
            }

            while (m--)                                  /* skip geo-info */
      if (!gets(buf)) err("Premature end of file");
  }

        if (!gets(buf)) err("Premature end of file");
        if (1 != sscanf (buf, "%d", &k) || k < 1) err("TR output problem");
        for (j = k; j--;) {                          /* read outputs */
  if (!gets(buf)) err("Premature end of file");
  if (3 != sscanf(buf, "%d%d%d", &k, &l, &m) || !k || --l < 0 || l
      >= n_PL || m < 0)
    err("Transition output def problem");
  if(k < 0) k = -k;

            while (k--) {
      t = (struct place_list *) malloc (sizeof(struct place_list));
      t->pl = &PLACES[l];
      t->next = TRANSITIONS[i].outpts;
      TRANSITIONS[i].outpts = t;
            }

            while (m--)                                  /* skip geo-info */
      if (!gets(buf)) err("Premature end of file");
        }

        if (!gets(buf)) err("Premature end of file");
        if (1 != sscanf (buf, "%d", &k) || k < 0) err("TR inhibt problem");
        for (j = k; j--;) {                          /* read inhibits */
  if (!gets(buf)) err("Premature end of file");
  if (3 != sscanf(buf, "%d%d%d", &k, &l, &m) || !k || --l < 0 || l
      >= n_PL || m < 0)
    err("Transition inhibit def problem");
  /* -- MODIFICATION INSERTED HERE  */
  while (k--) {
    t = (struct place_list *) malloc (sizeof(struct place_list));
    t->pl = &PLACES[l];
    t->next = TRANSITIONS[i].inhibs;
```

```
    TRANSITIONS[i].inhibs = t;
  }
  /* -- END OF MODIFICATION
  if (k) {
    t = (struct place_list *) malloc (sizeof(struct place_list));
    t->pl = &PLACES[l];
    t->next = TRANSITIONS[i].inhibs;
    TRANSITIONS[i].inhibs = t;
  }
  */
            while (m--)                                    /* skip geo-info */
      if (!gets(buf)) err("Premature end of file"); } }
}

err(s)
   char *s;
{
    fprintf (stderr, "%s\n", s);
    exit(1);
}

print_net ()                                        /* print the garbage */
{
    register int i, j;
    register struct place_list *t;

    for (i = 0; i < n_MK; i++)
        printf ("PARAM %s %.10e;\n", MARKS[i].name,  MARKS[i].deflt);
    for (i = 0; i < n_RT; i++)
        printf ("PARAM %s %.10e;\n", RATES[i].name,  RATES[i].deflt);
    for (i = 0; i < n_PL; i++) {
        j = PLACES[i].tokens;
        if (j < 0) {
            j = -1 - j;
            if (j >= n_MK)
                err ("Marking parameter problem");
            printf ("PLACE %s #marks= %s;\n", PLACES[i].name, MARKS[j].name);
        } else
            printf ("PLACE %s #marks= %d;\n", PLACES[i].name, j);
    }
    for (i = 0; i < n_TR; i++) {
        if (TRANSITIONS[i].rate < 0) {
  j = -0.5 - TRANSITIONS[i].rate;
  if (j >= n_RT)
    err ("Rate parameter problem");
  printf ("TRANS %s %s rate= %s, dep= %d : ", TRANSITIONS[i].name,
  trns_types[TRANSITIONS[i].type], RATES[j].name,
  TRANSITIONS[i].dep);
 } else
  printf ("TRANS %s %s rate= %e, dep= %d : ", TRANSITIONS[i].name,
  trns_types[TRANSITIONS[i].type], TRANSITIONS[i].rate,
  TRANSITIONS[i].dep);

        for (t = TRANSITIONS[i].inpts; t; t = t->next)
            printf ("%s ", t->pl->name);
        for (t = TRANSITIONS[i].inhibs; t; t = t->next)
            printf ("!%s ", t->pl->name);
        printf ("-->");
        for (t = TRANSITIONS[i].outpts; t; t = t->next)
            printf (" %s", t->pl->name);
        printf(";\n");
```

```
    }
    printf("$end\n");
}
```

A.3 *assign* — Place and Transition Assignment

A.3.1 *masg*: Make file for *assign*

```
##   Make file for assign
##
##
EXEC= assign
##
## -g = Debugging info
## -O = Optimize
CFLAGS= -g
##
## Libraries
## X11 X11 graphics library
##
LIBS=
#
#
OBJECTS= assign.o \
        syserr.o
##
## Compile & link
##
$(EXEC): $(OBJECTS)
cc $(CFLAGS) -o $(EXEC) $(OBJECTS) $(LIBS)
##
##
## End of make file
##
```

A.3.2 *assign.c* – 'C' program Code

```
#include <stdio.h>
#include <string.h>
#include "size_limits.h"
#include "defs.h"
#define LONGSIZE sizeof( long ) * 8

int longsize = LONGSIZE;

struct tr {    /* -- Holds info on transitions  */
  char name[ MAXLEN ];
  unsigned assigned : 1;
  float firing_rate;
  char type[ MAXLEN ];
  int host;
  long LInputMask[ MAXTRANS / LONGSIZE + 1 ];
  long GInputMask[ MAXTRANS / LONGSIZE + 1 ];
  char in_dep[ MAXDEP ] [ MAXLEN ];
  char out_dep[ MAXDEP ] [ MAXLEN ];
  char inhibit[ MAXDEP ] [ MAXLEN ];
} T[ MAXTRANS ];
```

```
struct pl {    /* -- Place info  */
  char name[ MAXLEN ];
  unsigned assigned : 1;
  unsigned global : 1;
  int host;
  int marking;
} P[ MAXTRANS ];

struct hosts {   /* -- Host info  */
  char HOSTNAME[ MAXLEN ];
  char CLIENTNAME[ MAXLEN ];
  char SERVERNAME[ MAXLEN ];
  int LOCAL_MASK_SIZE;
  int LOCAL_PLACES;
  int NO_TRANSITIONS;
} H[ MAXHOST ];

int GLOBAL_MASK_SIZE;
int GLOBAL_PLACES;

long GinputMask[ MAXTRANS /  LONGSIZE + 1 ];
long LinputMask[ MAXTRANS /  LONGSIZE + 1  ];

FILE *in_f, *hostf, *fopen();

int p_no, t_no, h_no;   /* -- Counters for places, transitions & hosts.  Has
                              the # of respective items when readinfo() is
                              finished   */

int assigned_places = 0, assigned_trans = 0;
BOOLEAN distributed = FALSE;

/*
**      readinfo()
**
**      Reads the input file and scans each line to determine if it is a
**      transition or a place.  The info is then assigned to the respective
**      arrays T and P for each element.  The "hostname file is read and
**      the info is placed in the H array.
*/

void
readinfo()
{
  char buf[ 1024 ], temp[ 1024 ], temp2[ 1024 ], temp3[ 1024 ],
       type[ 5 ], *p1, *p2;
  char *strchr(), *strtok();
  int d_no, inh_no, i, j, mark, pos;
  float rate;

  p_no = t_no = 0;
  while ( fgets( buf, sizeof( buf ), in_f ) )
    {
      sscanf( buf, "%s", temp );
      if ( !strcmp( temp, "PLACE" ) )     /* -- PLACE info is read */
        {
          sscanf( buf, "%*s%s%*s%d", temp, &mark );  /* -- get name & marking*/
          for( i = 0; ( P[ p_no ].name[ i ] = temp[ i ] ) != '\0'; i++ );
          P[ p_no ].marking = mark;
          P[ p_no ].assigned = 0;
          P[ p_no ].global = 0;
```

```
    P[ p_no ].host = -1;
    p_no++;
  }
else if ( !strcmp( temp, "TRANS" ) )   /* -- transition  */
  {
    sscanf( buf, "%*s%s%s%*s%f", temp, type, &rate );
    /* -- Get any host assignment  */
    T[ t_no ].assigned = 0;
    if ( strrchr( temp, '@' ) )
      {
        for( pos = 0;  temp[pos] != '@' ; pos++);
        for ( i = pos + 1, j = 0; (temp2[ j] = temp[ i ]) != '\0';
              j++, i++ );
        if ( sscanf( temp2, "%d", &i ) != -1 )
          {
            T[ t_no ].host = i;
            T[ t_no ].assigned = 1;
            assigned_trans++;
          }
      }
    strcpy( T[ t_no ].name, temp );
    if ( !T[ t_no ].assigned )
      T[ t_no ].host = -1;
    strcpy( T[ t_no ].type, type );
    T[ t_no ].firing_rate = rate;

    /* -- Get input dependencies  */
    p1 = strchr( buf, ':' );
    p2 = buf;
    pos = p1 - p2;
    /* -- Copy in dependency part of buf  */
    for (i = pos + 1, j = 0;  ( temp[j] = buf[i] ) != '-'; i++, j++ );
    temp[ j - 1 ] = '\0';
    d_no = 0;
    inh_no = 0;
    strcpy( temp2, strtok( temp, " " ) );
    if ( p1 = strchr( temp2, '!' ) )  /* -- Inhibitor arc found  */
      {
        for (i =  1, j = 0; ( temp3[j] = temp2[i] ) != '\0';
              i++, j++);
        strcpy( T[ t_no ].inhibit[ inh_no ], temp3 );
        inh_no++;
      }
    else
      {
        strcpy( T[ t_no ].in_dep[ d_no ], temp2 );
        d_no++;
      }
    for( ; ( p1 = strtok( NULL, " " ) )   /* -- Get rest of inputs  */
        != '\0'; )
      {
        strcpy( temp2, p1 );
        if ( p1 = strchr( temp2, '!' ) )  /* -- Inhibitor arc found  */
          {
            for (i =  1, j = 0; ( temp3[j] = temp2[i] ) != '\0';
                  i++, j++);
            strcpy( T[ t_no ].inhibit[ inh_no ], temp3 );
            inh_no++;
          }
        else
          {
```

```
                strcpy( T[ t_no ].in_dep[ d_no ], temp2 );
                d_no++;
              }
          }
        /* -- Get ouput dependencies  */
        p1 = strchr( buf, '>' );
        p2 = buf;
        pos = p1 - p2;
        /* -- Copy out dependency part of buf  */
        for (i = pos + 1, j = 0;  ( temp[j] = buf[i] ) != ';'; i++, j++ );
        temp[ j ] = '\0';
        d_no = 0;
        strcpy( T[ t_no ].out_dep[ d_no ], strtok( temp, " " ) );
        d_no++;
        for( ; ( p1 = strtok( NULL, " " ) )
            != '\0'; d_no++ )
           strcpy( T[ t_no ].out_dep[ d_no ], p1 );
        t_no++;
        if ( t_no > MAXTRANS || p_no > MAXTRANS )
          syserr( "Recompile with larger MAXTRANS constant" );
      }
    else   /* -- Input is #end  */
      close( in_f );
  }
 /* -- Read in "hostname" file */
 h_no = 0;
 while ( fgets (buf, sizeof( buf ), hostf ) )
   {
     temp[ 0 ] = '\0';
     sscanf( buf, "%d%s", &h_no, temp );
     strcpy( H[ h_no ].HOSTNAME, temp );
     h_no++;
   }
 close( hostf );
}

/*
**       assign()
**
**       The places and transitions are assigned to the hosts by this procedure
**       by successively looking at each place to count # of dependecies
**       to already assigned (preassigned) transitions.  If there is one
**       dependency, the place is local, if there is more than one, the place
**       is global.
**
**       The unassigned transitions are then checked to see how many dependen-
**       cies it has on each host.  The transition is placed at the highes
**       host count.
**
**       The above two steps are repeated until all places & transitions are
**       assigned to a host.
*/

void assign()
{
  char buf[ 1024 ], temp[ 1024 ];
  int host_count[ MAXHOST ];       /* -- Count # of references to host  */
  int p, t, tmp, i, no_dep;
  BOOLEAN done = FALSE;

  for ( i = 0; i < h_no; i++ ) host_count[ i ] = 0;
```

```
while ( !done )
  {
    /* -- For each place, see if it is free, if so, count # of dependencies
          on each host and assign it to host with highest count  */
    for ( p = 0; p < p_no; p++ )
      {
        if ( !P[ p ].assigned )
          {
            for ( t = 0; t < t_no; t++ )    /* -- For all transitions  */
              {
                if ( T[ t ].assigned )
                  {
                    /* -- Check if p is an input dependency   */
                    for ( i = 0; strcmp ( T [ t ].in_dep[ i ], "" ); i++ )
                      if ( !strcmp ( P[ p ].name, T[ t ].in_dep[ i ] ) )
                        host_count[ T[ t ].host ]++;
                    /* -- Check if p is an output dependency   */
                    for ( i = 0; strcmp ( T [ t ].out_dep[ i ], "" ); i++ )
                      if ( !strcmp ( P[ p ].name, T[ t ].out_dep[ i ] ) )
                        host_count[ T[ t ].host ]++;
                    /* -- Check if p is an inhibitor dependency   */
                    for ( i = 0; strcmp ( T [ t ].inhibit[ i ], "" ); i++ )
                      if ( !strcmp ( P[ p ].name, T[ t ].inhibit[ i ] ) )
                        host_count[ T[ t ].host ]++;
                  }
              }
            /* -- Count # of occurrences of each transition to place the
                  place at the host with the most dependecies  */
            no_dep = 0;
            tmp = 0;
            for ( i = 0; i < h_no; i++ )
              {
                if ( host_count[ i ] > 0 )
                  {
                    no_dep++;
                    tmp = i;
                  }
                host_count[ i ] = 0;
              }
            switch ( no_dep )   /* -- Check if place is a dependent of .. */
              {
              case 0 :          /* -- 0 transitions ->  free place  */
                break;
              case 1 :          /* -- Transitions on 1 host -> local place */
                P[ p ].host = tmp;
                assigned_places++;
                P[ p ].assigned = 1;
                break;
               default :         /* -- Transitions on > 1 hosts -> global  */
                P[ p ].global = 1;
                P[ p ].host = -1;
                assigned_places++;
                P[ p ].assigned = 1; } } }
    /* -- For each transtion, check if it assigned, if not, count # of
          dependecies and assign it to the host with the highest count  */
    for ( t = 0; t < t_no; t++ )
      {
        if ( !T[ t ].assigned )
          {
            /* -- For each input dependency, count # of occurrences on host*/
```

```
            for ( p = 0; strcmp ( T[ t ].in_dep[ p ] , "" ); p++ )
              for ( i = 0; i < p_no; i++ )
                if ( !strcmp ( T[ t ].in_dep[ p ], P[ i ].name ) )
                  if ( P[ i ].assigned && P[i].host >= 0 )
                    host_count[ P[ i ].host ]++;

            /* -- For each output dependency, count # of times on host */
            for ( p = 0; strcmp ( T[ t ].out_dep[ p ], "" ); p++ )
              for ( i = 0; i < p_no; i++ )
                if ( !strcmp ( T[ t ].out_dep[ p ], P[ i ].name ) )
                  if ( P[ i ].assigned && P[i].host >= 0 )
                    host_count[ P[ i ].host ]++;

            /* -- For each inhibitor dependency, count # of times on host */
            for ( p = 0; strcmp ( T[ t ].inhibit[ p ], "" ); p++ )
              for ( i = 0; i < p_no; i++ )
                if ( !strcmp ( T[ t ].inhibit[ p ], P[ i ].name ) )
                  if ( P[ i ].assigned && P[i].host >= 0 )
                    host_count[ P[ i ].host ]++;

            tmp = 0;
            for ( i = 0; i < h_no; i++ )
              {
                if ( host_count[ i ] > tmp )
                  tmp = i;
                host_count[ i ] = 0;
              }
            T[ t ].host = tmp;
            T[ t ].assigned = 1;
            assigned_trans++;
          }
        }
      if ( ( assigned_places >= p_no ) && (assigned_trans >= t_no ) )
        done = TRUE;
    }
}

/*
**      fill_host()
**
**      The #define statements for each token player is generated in this
**      procedure by counting # of local and global places and # of
**      transitions at each host.
**
**      The token ring succession is determined and the CLIENTNAME and
**      SERVERNAMEs are assigned.
*/

void fill_host()
{
  int i, j;

  for ( i = 0; i < h_no; i++ )   /* -- Count # of places & trans @ host i */
    {
      for ( j = 0; j < t_no; j++ )
        if ( T[ j ].host == i )
          H[ i ].NO_TRANSITIONS++;

      for ( j = 0; j < p_no; j++ )
        if ( P[ j ].host == i )
          H[ i ].LOCAL_PLACES++;
```

```
      H[ i ].LOCAL_MASK_SIZE = H[ i ].LOCAL_PLACES /
                               longsize + 1;
    }

  for ( i = 0; i < p_no; i++ )    /* -- Count # of global places  */
    if ( P[ i ].global )
      GLOBAL_PLACES++;

  GLOBAL_MASK_SIZE = GLOBAL_PLACES / longsize + 1;

  for ( i = 1; i < h_no; i++ )    /* -- Assign hostnames  */
    {
      strcpy( H[ i ].SERVERNAME, H[ i - 1 ].HOSTNAME );
      strcpy( H[ i - 1 ].CLIENTNAME, H[ i ].HOSTNAME );
    }
  strcpy( H[ 0 ].SERVERNAME, H[ h_no - 1 ].HOSTNAME );
  strcpy( H[ h_no - 1 ].CLIENTNAME, H[ 0 ].HOSTNAME );
}

/*
**        write_defs()
**
**        The information generated by fill_host() is written to the net<i>.h
**        definition files for compilation of token players.
**
*/

void
write_defs()
{
  int i;
  char c;
  char f[ 100 ];
  int t_count, t, h;
  FILE *fp, *fopen();

  /* --  Write initialization for XCommand client  */
  sprintf( f, "XCommand.i" );
  if ( ( fp = fopen( f, "w" ) ) == NULL )
    syserr( "fopen XCommand.i" );
  fprintf( fp, "%d\n", h_no );
  for ( i = 0; i < h_no; i++ )
    fprintf( fp, "%s\n", H[ i ].HOSTNAME );
  fprintf( fp, "%s\n", H[ 0 ].HOSTNAME );
  close( fp );

  /* --  Write initialization for XDisplay server  */
  sprintf( f, "XDisplay.i" );
  if ( ( fp = fopen( f, "w" ) ) == NULL )
    syserr( "write XDisplay.i" );
  /* -- Print # of token players  */
  fprintf( fp, "%d ", h_no );
  /* -- Print # of global places  */
  fprintf( fp, "%d ", GLOBAL_PLACES );
  fprintf( fp, "%d ", p_no );
  fprintf( fp, "%d\n", t_no );
  for ( i = 0; i < h_no; i++ )
    {
      fprintf( fp, "%s ", H[ i ].HOSTNAME );
      fprintf( fp, "%d %d\n", H[ i ].LOCAL_PLACES, H[ i ].NO_TRANSITIONS );
```

```
  }
for ( i = 0; i < p_no; i++ )
  fprintf( fp, "%s %d\n", P[ i ].name, P[ i ].host );
for ( i = 0; i < t_no; i++ )
  fprintf( fp, "%s %d\n", T[ i ].name, T[ i ].host );

/* -- Note - make room for inh. arcs  */
close( fp );

/* -- Write net<i>.h  definition files for compilation of token players  */
for ( i = 0; i < h_no; i++ )
  {
    sprintf( f, "%net%d.h", i );
    if ( ( fp = fopen( f, "w" ) ) == NULL )
      syserr( "fopen" );
    fprintf( fp, "#define\tHOSTNAME\t\"%s\"\n", H[ i ].HOSTNAME );
    fprintf( fp, "#define\tHOSTNO\t%d\n", i );
    fprintf( fp, "#define\tNO_HOSTS\t%d\n", h_no );
    fprintf( fp, "#define\tCLIENTNAME\t\"%s\"\n", H[ i ].CLIENTNAME );
    fprintf( fp, "#define\tSERVERNAME\t\"%s\"\n", H[ i ].SERVERNAME );
    fprintf( fp, "#define\tHOSTONAME\t\"%s\"\n", H[ 0 ].HOSTNAME );
    fprintf( fp, "#define\tLOCAL_PLACES\t%d\n", H[ i ].LOCAL_PLACES );
    fprintf( fp, "#define\tNO_TRANSITIONS\t%d\n", H[ i ].NO_TRANSITIONS );
    fprintf( fp, "#define\tLOCAL_MASK_SIZE\t%d\n", H[ i ].LOCAL_MASK_SIZE );
    fprintf( fp, "#define\tGLOBAL_PLACES\t%d\n", GLOBAL_PLACES );
    fprintf( fp, "#define\tGLOBAL_MASK_SIZE\t%d\n", GLOBAL_MASK_SIZE );
    sprintf ( f, "net%d.i", i );
    fprintf( fp, "#define\tNETFILE\t\"%s\"\n", f );
    sprintf ( f, "tr_links%d.i", i );
    fprintf( fp, "#define\tLINKFILE\t\"%s\"\n", f );
    sprintf ( f, "interface%d.i", i );
    fprintf( fp, "#define\tDRIVERFILE\t\"%s\"\n", f );
    if ( i == 0 )
      fprintf( fp, "#define\tHOST0\t1\n" );
    close( fp );
  }

/* -- Write tname<i>.h  definition files for obtaining the number of each
      transition at each host -- will replace all nonletter and digits with
      _AT_ for @, _IN_  for ?, and _OUT_ for !.  All other characters are
      replaced with _.  This makes the #define names legal in 'C'  */

for ( h = 0; h < h_no; h++ )
  {
    sprintf( f, "tname%d.h", h );
    if ( ( fp = fopen( f, "w" ) ) == NULL )
      syserr( "fopen" );
    t_count = 0;
    for ( t = 0; t < t_no; t++ )
      {
        if (T[t].host == h)
          {
            fprintf( fp, "#define\t");
            for (i = 0; (c = T[t].name[i]) != '\0'; i++)
              {
                if ((c >= '0' && c <= '9') || (c >= 'a' && c <= 'z') ||
                    (c >= 'A' && c <= 'Z'))
                  fputc(c, fp);
                else
                  switch (c)
                    {
```

```
                        case '@' : fprintf(fp, "_AT_"); break;
                        case '?' : fprintf(fp, "_IN"); break;
                        case '!' : fprintf(fp, "_OUT"); break;
                        default  : fprintf(fp, "_"); break;
                        }
                    }
                  sprintf( f, "%d", t_count++ );
                  fprintf( fp, "\t%s\n", f);
                }
            }
          close( fp );
        }
}

/*
**      write_inits()
**
**      The initialization code (in 'C') is generated as include files for
**      each token player.  The extended incidence matrix is generated,
**      assigning to each transition the input and output arcs.  The
**      local marking vector is generated and for HOST0 token player,
**      the global marking vector is initialized as well.
**
*/

void
write_inits()
{
  int i, j, k, p, t, h, flag;
  char temp[ 1024 ], buf[ 1024 ], f[ 100 ];
  FILE *fp, *fopen();

  /* -- Initialize all masks */
  for ( t = 0; t < t_no; t++ )
    {
      for ( j = 0; j < (int) MAXTRANS / longsize + 1; j++ )
        {
          T[ t ].GInputMask[ j ] = 0;
          T[ t ].LInputMask[ j ] = 0;
          GinputMask[ j ] = 0;
          LinputMask[ j ] = 0;
        }
    }
  /* -- For each host, make include file net<n>.i for init.c  */
  for ( h = 0; h < h_no; h++ )
    {
      sprintf ( f, "net%d.i", h );
      if ( ( fp = fopen( f, "w" ) ) == NULL )
        syserr( "fopen" );

      /* -- For each transition, initialize input incidence matrix  */
      t = 0;
      for ( i = 0; i < t_no; i++ )
        {
          if ( T[ i ].host == h )
            {
              fprintf( fp, "strcpy( Transition[ %d ].tag, \"%s\" );\n",
                      t, T[ i ].name );
              fprintf( fp, "Transition[ %d ].firing_rate = %f;\n",
                      t, T[ i ].firing_rate );
              if ( !strcmp( T[ i ].type, "imm" ) )
```

```
  fprintf( fp, "Transition[ %d ].immediate = 1;\n", t );
if ( !strcmp( T[ i ].type, "det" ) )
  fprintf( fp, "Transition[ %d ].timed = 1;\n", t );

/* -- Check each input dependency to generate LInputVector   */
p = 0;
for ( j = 0; j < p_no; j++ )
  {
    if ( P[ j ].host == h )
      {
        for ( k = 0; strcmp( T[ i ].in_dep[ k ], "" ); k++ )
          if ( !strcmp( P[ j ].name, T[ i ].in_dep[ k ] ) )
            {
              fprintf( fp,
                "Transition[%d].LInputVector[%d]++;\n",t, p );
              T[ i ].LInputMask[ (int) ( p / longsize ) ]
                |= ( 1 << p );
            }
        p++;
      }
  }
/* -- Write mask to file  */
for ( j = 0; j <= ( int )( p / longsize ); j++ )
  fprintf( fp, "Transition[ %d ].LInputMask[ %d ] = 0x%x;\n",
           t, j, T[ i ].LInputMask[ j ] );
/* -- Check each output dependency to generate LOutputVector   */
p = 0;
for ( j = 0; j < p_no; j++ )
  {
    if ( P[ j ].host == h )
      {
        for ( k = 0; strcmp( T[ i ].out_dep[ k ], "" ); k++ )
          if ( !strcmp( P[ j ].name, T[ i ].out_dep[ k ] ) )
            {
              fprintf( fp,
                "Transition[%d].LOutputVector[%d]++;\n", t, p );
            }
        p++;
      }
  }
/* -- Check each inhibit dependency to generate LInhibitVector */
flag = 0;
p = 0;
for ( j = 0; j < p_no; j++ )
  {
    if ( P[ j ].host == h )
      {
        for ( k = 0; strcmp( T[ i ].inhibit[ k ], "" ); k++ )
          if ( !strcmp( P[ j ].name, T[ i ].inhibit[ k ] ) )
            {
              fprintf( fp,
                "Transition[%d].LInhibitVector[%d]++;\n",t, p );
              flag = 1;
            }
        p++;
      }
  }
if ( flag )
  fprintf( fp, "Transition[ %d ].LInhArc = 1;\n", t );
/* -- Generate global input vector   */
flag = 0;
```

```
p = 0;
for ( j = 0; j < p_no; j++ )
  {
    if ( P[ j ].global )
      {
        for ( k = 0; strcmp( T[ i ].in_dep[ k ], "" ); k++ )
          if ( !strcmp( P[ j ].name, T[ i ].in_dep[ k ] ) )
            {
              fprintf( fp,
                "Transition[%d].GInputVector[%d]++;\n", t, p );
              T[ i ].GInputMask[ (int) ( p / longsize ) ]
                |= ( 1 << p );
              flag = 1;
            }
        p++;
      }
  }
/* -- Write mask to file  */
for ( j = 0; j <= ( int )( p / longsize ); j++ )
  fprintf( fp, "Transition[ %d ].GInputMask[ %d ] = 0x%x;\n",
           t, j, T[ i ].GInputMask[ j ] );
if ( flag )
  fprintf( fp, "Transition[ %d ].glob_in = 1;\n", t );
/* -- Generate global output vector   */
flag = 0;
p = 0;
for ( j = 0; j < p_no; j++ )
  {
    if ( P[ j ].global )
      {
        for ( k = 0; strcmp( T[ i ].out_dep[ k ], "" ); k++ )
          if ( !strcmp( P[ j ].name, T[ i ].out_dep[ k ] ) )
            {
              fprintf( fp,
               "Transition[%d].GOutputVector[%d]++;\n", t, p );
               flag = 1;
            }
        p++;
      }
  }
if ( flag )
  fprintf( fp, "Transition[ %d ].glob_out = 1;\n", t );
/* -- Generate global inhibitor vector   */
flag = 0;
p = 0;
for ( j = 0; j < p_no; j++ )
  {
    if ( P[ j ].global )
      {
        for ( k = 0; strcmp( T[ i ].inhibit[ k ], "" ); k++ )
          if ( !strcmp( P[ j ].name, T[ i ].inhibit[ k ] ) )
            {
              fprintf( fp,
                "Transition[%d].GInhibitVector[%d]++;\n", t, p );
              flag = 1;
            }
        p++;
      }
  }
if ( flag )
  {
```

```
            fprintf( fp, "Transition[ %d ].GInhArc = 1;\n", t );
            fprintf( fp, "Transition[ %d ].glob_in = 1;\n", t );
          }
        t++; /* -- Increment trans. counter for this host  */
      }
  }
/* -- Generate local marking vector  */
p = 0;
for ( j = 0; j < p_no; j++ )
  {
    if ( P[ j ].host == h )
      {
        fprintf( fp, "strcpy( LocalMarkingV.tag[ %d ], \"%s\" );\n",
                 p, P[ j ].name );
        if ( P[ j ].marking )
          {
            LinputMask[ (int) ( p / longsize ) ]
              |= ( 1 << p );
            fprintf( fp, "LocalMarkingV.marking[ %d ] = %d;\n", p,
                         P[ j ].marking );
          }
        p++;
      }
  }
/* -- Write LocalMarkingV mask  */
for ( j = 0; j < (int) ( p / longsize + 1 ); j++ )
  fprintf( fp, "LocalMarkingV.LinputMask[ %d ] = 0x%x;\n", j,
           LinputMask[ j ] );
/* -- If this is HOST0, initialize global marking vector  */
if ( h == 0 )
  {
    p = 0;
    for ( j = 0; j < p_no; j++ )
      {
        if ( P[ j ].global )
          {
            fprintf( fp, "strcpy( GlobalMarkingV.tag[ %d ], \"%s\" );\n",
                     p, P[ j ].name );
            if ( P[ j ].marking )
              {
                GinputMask[ (int) ( p / longsize ) ]
                  |= ( 1 << p );
                fprintf( fp, "GlobalMarkingV.marking[ %d ] = %d;\n", p,
                         P[ j ].marking );
              }
            p++;
          }
      }
    /* -- Write gmv mask  */
    for ( j = 0; j < (int) ( p / longsize + 1 ); j++ )
      fprintf( fp, "GlobalMarkingV.GinputMask[ %d ] = 0x%x;\n",
               j, GinputMask[ j ] );
    fprintf( fp, "GlobalMarkingV.GM_av = 1;\n" );
    fprintf( fp, "GlobalMarkingV.update = 0;\n" );
  }
 else   /* -- Initialize tags only  */
  {
    p = 0;
    for (  j = 0; j < p_no; j++ )
      {
        if ( P[ j ].global )
```

```
                {
                  fprintf( fp, "strcpy( GlobalMarkingV.tag[ %d ], \"%s\" );\n",
                           p, P[ j ].name );
                  p++;
                }
            }
          fprintf( fp, "GlobalMarkingV.GM_av = 0;\n" );
          fprintf( fp, "GlobalMarkingV.update = 0;\n" );
        }
      close ( fp );
    }
}

main(argc, argv)   /* -- Reads the net info in the file given to main and
                         distributes the places and transitions across the
                         different nets on the hosts in the "hostname" file */
int argc;
char * argv;
{
  char name[100];
  if ( ( in_f = fopen("test.n", "r" ) ) == -1 )
    syserr( "open in_f" );
  if ( ( hostf = fopen("hostnames", "r" ) ) == -1 )
    syserr( "open hostnames" );

  readinfo();     /* -- Read in net info and put in P, T and H arrays  */
  assign();       /* -- Assign transitions and places to hosts  */
  fill_host();    /* -- Get host names and assign server and client names  */
  write_defs();   /* -- Write net<n>.h include files   */
  write_inits(); /* -- Write net<n>.i net definition files  */
}
```

A.4 *pnn* — Token Player Implementation

A.4.1 *mpn*: Make file for *pnn*

```
##
##
##
EXEC=        pn
##
##        -g = Debugging info
##        -O = Optimize
CFLAGS=        -g
##
##        Libraries
##        X11        X11 graphics library
##
LIBS= -lm
#
#
OBJECTS=        player.c        \
                syserr.o        \
                setblock.o        \
                server_intr.o        \
                intr_timer.o        \
                client.o        \
                timed_trans_handler.c        \
                timer.o                \
```

```
                xrand.o                 \
                event_handler.c         \
                init_net.c      \
                dump.c
##
##      Compile & link
##
$(EXEC):        $(OBJECTS)
                cc $(CFLAGS) -o $(EXEC) $(OBJECTS) $(LIBS)
##
##
##      End of make file
##
```

A.4.2 *master-player.h* — Data Structure Definition File

```
#include "size_limits.h"                /* -- Contains maximum sizes  */

#define EXTERNAL_IO 1                   /* -- Defined when the DPNC uses a socket
                                           for communication with a device
                                           driver program  */

#define GMV_LIMIT NO_TRANSITIONS+10     /* -- Number of iterations of the token
                                           player before releasing the gmv */

#define TR_LIMIT NO_TRANSITIONS         /* -- Number of fired transitions before
                                           sending the firing sequence to the
                                           XDisplay server (defines update
                                           rate) */

#define LONGSIZE sizeof( long ) * 8     /* -- # of bytes in a long int  */
#define MASK_BUFSIZE GLOBAL_MASK_SIZE /* -- Size of mask for global m.v. mask*/
#define GMV_BUFSIZE  1 + MASK_BUFSIZE + GLOBAL_PLACES   /* -- Size of gmv buf*/
#define NO_BYTES ( sizeof ( long ) * GMV_BUFSIZE )      /* -- # of bytes in
                                                           gmv buffer  */

/*#define DEBUG 1 */    /* -- Print general info  */
#define DEBUG1 1        /* -- Print transition fired */
#define DEBUG2 1        /* -- Print timed transitions duration  */

struct trans {          /* -- The struct for the transition info.    */
  unsigned        firing        : 1;    /* -- Currently firing            */
  unsigned        fire          : 1;    /* -- Fire (timed)              */
  unsigned        glob_in       : 1;    /* -- Input from global m.v. */
  unsigned        glob_out      : 1;    /* -- Output to  global m.v. */
  unsigned        LInhArc       : 1;    /* -- Local inh. arc          */
  unsigned        GInhArc       : 1;    /* -- Global inh. arc         */
  unsigned        immediate     : 1;    /* -- Immediate transition    */
  unsigned        timed         : 1;    /* -- Exponentially timed tr.*/
  unsigned        preconditions : 1;    /* -- Set if preconditions
                                           exist (procedure)  */
  unsigned        postprocessing : 1;   /* -- Set if postproc. is
                                           to be done (procedure
                                           call)              */
  unsigned long   end_time;             /* -- The expiration time of
                                           the timer for timed tr.*/
  float    firing_rate;                 /* -- Firing rate for timed or
                                           deterministic tr.      */
  int     (*preprocess) ();             /* -- Procedure to be called
     to see if transition is
```

```
    enabled.  Returns true if
    conditions are ok      */
  int     (*postprocess)();            /* -- Procedure to be called
    when transition fires  */
  char    tag[MAXLEN];                   /* -- name of transition     */
  long    LInputMask[LOCAL_MASK_SIZE];   /* -- Local mask     */
  long    GInputMask[GLOBAL_MASK_SIZE];  /* -- Global mask    */
  short   LInputVector[LOCAL_PLACES];    /* Input incidence matrix */
  short   LOutputVector[LOCAL_PLACES];   /*Output inc. matrix     */
  short   GInputVector[GLOBAL_PLACES];   /* Input incid. matrix   */
  short   GOutputVector[GLOBAL_PLACES];  /*Output inc. matrix    */
  short   LInhibitVector[LOCAL_PLACES];  /*  Inhibitor arc mult. */
  short   GInhibitVector[LOCAL_PLACES];  /*  Inhibitor arc mult. */
};

struct LocalMV {     /* -- the struct for the local marking vector        */
  short       marking[LOCAL_PLACES];        /* -- Marking of local places*/
  long        LinputMask[LOCAL_MASK_SIZE];  /* -- Local mask     */
  char        tag[LOCAL_PLACES] [MAXLEN];   /* -- Tag to be printed   */
};

struct GlobalMV { /* -- The struct for the global marking vector       */
  unsigned        update : 1;        /* -- Set if update necs.    */
  unsigned        GM_av : 1;         /* -- Set if GM available     */
  long            GinputMask[ GLOBAL_MASK_SIZE];  /* -- Global mask    */
  short           marking[GLOBAL_PLACES];         /* -- temporary marking */
  short           old[GLOBAL_PLACES];             /* -- temporary storage */
  char            tag[GLOBAL_PLACES] [MAXLEN];    /* -- Tag for local
        places   */
};

enum comd_type { RUN, HALT, RESET, STOP };
```

A.4.3 *defs.h* — Macro Definitions

```
typedef enum { FALSE, TRUE } BOOLEAN;

#define lowbyte(w) ((w) & 0377)
#define highbyte(w) lowbyte((w) >> 8)
```

A.4.4 *size_limits.h* — Sizes of Data Structures

```
#define MAXDEP   50            /* -- # of dependencies of each transition  */
#define MAXHOST  10            /* -- Max # of hosts */
#define MAXTRANS 200           /* -- Max # of transitions/places in net   */
#define MAXLEN 30              /* -- Max string length for any name    */

#define MAXPROCESS 4           /* -- maximum number of processes forked that
                                      must be killed before exiting main()  */
```

A.4.5 *portnums.h* — Socket Port Numbers

```
#define PLAYERPORT 1500    /* -- Port number for token ring sockets  */
#define COMMANDPORT 2500   /* -- Port number for command sockets  */
#define DISPLAYPORT 3500   /* -- Port number for display sockets  */
#define IOPORT 4500        /* -- Port number for i/o with device driver  */
#define OFFSET1 50         /* -- Offset for simulator 1  */
#define OFFSET2 75         /* -- Offset for simulator 2  */
```

A.4.6 *player.c* — Token Player Routines and main()

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#include <fcntl.h>
#include <errno.h>
#include <signal.h>
#include "portnums.h"
#include "player.h"
#include "size_limits.h"
#include "defs.h"

extern BOOLEAN single_step;
extern BOOLEAN event_flag;

struct trans    Transition[NO_TRANSITIONS]; /* -- Array of transitions  */
struct LocalMV  LocalMarkingV;  /* -- The local marking v.   */
struct GlobalMV GlobalMarkingV;       /* -- The global m.v.   */

int    longsize = LONGSIZE;
int    token_in;           /* -- File descriptors for use to receive and send */
int    token_out;           /*    the token vector */
int    out_X_win;          /* -- File descriptor for writing info to the
                              X window for this player  */
int    command_in;         /* -- File descriptor for command socket.  */
int    io_socket;           /* -- File descriptor used for receiving and sending
                              external event info */
enum   comd_type command; /* -- Type for possible command modes of player */

/* ------------------------------------------------------------------------
**
**        fire_timed_transition()
**
**        Fires a timed transition.
*/

fire_timed_transition( tr_no )
int tr_no;        /* -- Holds the integer # transition   */
{
  int (*exec) ();       /* -- A temporary pointer to the address of the
                              procedure to be called.  */
#ifdef DEBUG1
      printf( "\n%s fired timed ", Transition[tr_no].tag );
#endif
      remove_tokens( tr_no );
      if ( Transition[ tr_no ].postprocessing )
        {
          exec = Transition[ tr_no ].postprocess; /* -- Assign address
                                                        to exec  */
          (*exec) ( tr_no );                     /* -- Call procedure   */
        }
      add_tokens( tr_no );
      update_Xwindow( tr_no, 1 );
      event_flag = TRUE;
      if ( single_step ) command = RUN;
}

/* ------------------------------------------------------------------------
**        token_player()
```

```
**
**        is called when an event occurs (RUN, external event, timed transition
**        enabled or the gmv is available.
**
**      The player will run until
**         a. No more transitions fire, or
**        b. a command different than RUN is received.
**
**        The global marking vector is kept for GMV_LIMIT # of iterations until
**        it is released to the netx token player.
**
**        When no more transitions fire and the mode is RUN, blocking is
**        turned on for the eventpipe so that the token player will not be
**        invoked again until an event occurs.  This makes the token player
**        event driven.
**
**      When a transition is enabled, it fires if it is an immediate
**        transition.  Timed transitions are fired later (if still enabled).
**
*/

void
token_player()
{
  int tr_cnt = 1;                          /* -- Counter for transitions  */
  int tr;

  while ( ( ( tr = get_next_transition() ) != -1 ) && ( command == RUN ) )
    {
      get_event();                         /* -- Any new event ?  */
      if ( enable ( tr ) )
fire( tr );                      /* -- Fire transition tr   */

      if ( GlobalMarkingV.GM_av )
        {
          tr_cnt += 1;
          if ( tr_cnt > GMV_LIMIT  )   /* -- GMV_LIMIT iterations done  */
            {
              release_gmv();                   /* -- Release gmv to next token pl.*/
              tr_cnt = 1;  } } }

  if ( command == RUN )                    /* -- Release gmv when no other
                                                 transisiton can fire */
    release_gmv();
}

/* ------------------------------------------------------------------------
**     fire()
**
**     fire will call procedure for immediate or timed transitions (if any).
**     If the transition has a procedure and is immediate, this procedure is
**     called immediately and executed sequentially (within same context as
**     fire() ).  The data structures in lmv and gmv are updated immediately,
**     as is the Transition[ t ] data structure.
**
**     If the transition is timed, the transition # is given to timed_trans_
**     handler().  The transition may fire if still enabled when the timer
**     expires for the transition.
**
*/
```

```
fire( tr_no )
int tr_no;
{
  int (*exec) ();       /* -- A temporary pointer to the address of the
                              procedure to be called.  */
  if ( Transition[ tr_no ].immediate )
    {
      remove_tokens( tr_no );                  /* -- Remove tokens  */
      add_tokens( tr_no );                     /* -- Add tokens     */
#ifdef DEBUG1
      printf("\n%s fired",Transition[tr_no].tag);
#endif
      if ( Transition[ tr_no ].postprocessing )
        {
          exec = Transition[ tr_no ].postprocess; /* -- Assign address
                                                        to exec  */
          (*exec) ( tr_no );                      /* -- Call procedure   */
        }
      update_Xwindow( tr_no, 1 );
      event_flag = TRUE;
      if ( single_step ) command = HALT;
    }
   else         /* -- Timed transition  */
    {
      if ( Transition[ tr_no ].fire )          /* -- Timer has finished */
        {
          Transition[ tr_no ].fire = 0;
          fire_timed_transition( tr_no );       /* -- Fire transition if still
                                                     enabled  */
        }
       else
        {
          Transition[ tr_no ].firing = 1;      /* -- No timer is running for
                                                     this transition, so start
                                                     the timer  */
          start_timer( tr_no );
        }
    }
}    /* -- fire()   */


/* ------------------------------------------------------------------------
**      enable(tr_no)
**
**      2.1) If .global AND global marking vector is avaliable
**      2.2) If AND global input masks to see if this transition may fire
**      2.3) Check # of tokens available ==> check_globmv
**      3.1) If AND local input masks to see if this transition may fire
**      3.2) Check # of tokens available ==> check_localmv
**      3.3) If preconditions set, check if these are satisfied
**      4) Return true if all conditions are satisfied
**
*/

int
enable(t)
int t;   /* -- t is the transition number to be checked  */
{
  int (*exec) ();       /* -- A temporary pointer to the address of the
                              procedure to be called.  */
  int i;
```

```
/* -- If a timer is running for this transition, do not check again  */
if ( Transition[ t ].firing && Transition[ t ].end_time )
  return ( 0 );

/* -- Clear .firing flag such that the transition may be tested again */
if ( Transition[ t ].firing && !Transition[ t ].end_time )
  {
    Transition[ t ].firing = 0;
    Transition[ t ].fire = 1;
  }

/* -- Both conditions must be true    */
if ( Transition[ t ].glob_in )
  if ( !(GlobalMarkingV.GM_av ) )                    /* 2.)        */
    return( 0 );
  else
    {
      /* -- Check Global marking mask  */
      for ( i = 0; i < GLOBAL_MASK_SIZE; i++ )
        if ( !( ( Transition[ t ].GInputMask[ i ]
                  & GlobalMarkingV.GinputMask[ i ] ) ==
                Transition[ t ].GInputMask[ i ] ) )
          return( 0 );

      /* -- Check Global marking vector  */
      if ( !( check_globmv( t ) ) )
        return( 0 );                                  /* 2.3)      */

      /* -- Check Inhibitor arcs if any */
      if ( Transition[ t ].GInhArc )
        if ( !check_glob_inhibit(t) )
            return( 0 );
    }

 /* -- Check Local marking vector  */
 for ( i = 0; i < LOCAL_MASK_SIZE; i++ )
   {
     if ( !( ( Transition[ t ].LInputMask[ i ] &     /* 3.1)    */
               LocalMarkingV.LinputMask[ i ]) ==
             Transition[ t ].LInputMask[ i ]))
       return( 0 );
   }

 /* -- Check Inhibitor arcs if any */
 if ( Transition[ t ].LInhArc )
   if ( !check_local_inhibit(t) )
     return( 0 );

 if ( check_localmv( t ) )                            /* 3.3)    */
   {
     if ( Transition[ t ].preconditions )             /* 3.4)    */
       {
         exec = Transition[ t ].preprocess; /* -- Assign address    */
         if ( (*exec) ( t ) )               /* -- Call procedure    */
           return( 1 );                               /* 4)      */
         else
           return( 0 );
       }
     else
       return( 1 );
```

```
    }
  else
    return( 0 );
}       /* -- enable()   */


/*  ------------------------------------------------------------------
**
**      check_globmv(t)
**
**      1) Check each input arc to transition t to see if # of tokens in
**         the GlobalMarkingV >= the number of input arcs to t from each
**         place in the GMV
*/

check_globmv( t )

     int t;      /* -- t is the transition number to be checked  */

{
  int i;

  for ( i = 0; i < GLOBAL_PLACES; i++ )
    if ( Transition[ t ].GInputVector[ i ] >
         GlobalMarkingV.marking[ i ] )
      return( 0 );
  return( 1 );            /*  Return TRUE only if all arcs & places checked  */

}


/*  ------------------------------------------------------------------
**
**      check_localmv(t)
**
**      1) Check each input arc to transition t to see if # of tokens in
**         the LocalMarkingV >= the number of input arcs to t from each
**         place in the LMV
**
*/

check_localmv( t )
int t;      /* -- t is the transition number to be checked  */
{
  int i;

  for ( i = 0; i < LOCAL_PLACES; i++ )
    if ( Transition[ t ].LInputVector[ i ] >
         LocalMarkingV.marking[ i ] )
      return( 0 );
  return( 1 );            /*  Return TRUE only if all arcs & places checked  */
}

/*  ------------------------------------------------------------------
**
**      check_glob_inhibit(t)
**
**      1) Check each inhibitor arc not -1 to see if
**         multiplicity of inh. arc >= marking in place + output mult. to place
*/
```

```
check_glob_inhibit( t )
int t;     /* -- t is the transition number to be checked  */
{
  int i;

  for ( i = 0; i < GLOBAL_PLACES; i++ )
    if ( Transition[ t ].GInhibitVector[ i ] > 0 )
      if ( Transition[ t ].GInhibitVector[ i ] <
          ( GlobalMarkingV.marking[ i ] + Transition[ t ].GOutputVector[ i ] ))
        return( 0 );
  return( 1 );           /*  Return TRUE only if all arcs & places checked  */
}


/*   ------------------------------------------------------------------------
**
**      check_local_inhibit(t)
**
**      1) Check each inhibitor arc not -1 to see if
**         multiplicity of inh. arc >= marking in place + output mult. to place
*/

check_local_inhibit( t )
int t;     /* -- t is the transition number to be checked  */
{
  int i;

  for ( i = 0; i < LOCAL_PLACES; i++ )
    if ( Transition[ t ].LInhibitVector[ i ] > 0 )
      if ( Transition[ t ].LInhibitVector[ i ] <
          ( LocalMarkingV.marking[ i ] + Transition[ t ].LOutputVector[ i ] ) )
        return( 0 );
  return( 1 );           /*  Return TRUE only if all arcs & places checked  */
}

/*   ------------------------------------------------------------------------
**
**        remove_tokens( transition )
**
**        remove_tokens() is called after a transition is enabled and is fired.
**        The procedure has a critical section because the marking vectors may
**        be accessed by other processes concurrently.
**
**        1) For each input arc to the transition - remove # of tokens
**           corresponding to # of output arcs to each output place.
**        2) Set the masks too.
*/

remove_tokens( t )
int t;   /* The transition */
{
  int i;                          /* -- Counter      */

  /* -- Remove tokens from local places  */
  for ( i = 0; i < LOCAL_PLACES; i++ )
    if ( ( LocalMarkingV.marking[ i ] -= Transition[ t ].LInputVector[ i ] )
        == 0 )
      LocalMarkingV.LinputMask[ (int) i / longsize ] =
        LocalMarkingV.LinputMask[ (int) i / longsize ] &
          ~( 0x1 << ( i % longsize ) );
```

```
  /* -- Remove tokens from global places  */
  if ( Transition[ t ].glob_in )
    for ( i = 0; i < GLOBAL_PLACES; i++ )
      if ( ( GlobalMarkingV.marking[ i ] -=
          Transition[ t ].GInputVector[ i ] ) == 0 )
        GlobalMarkingV.GinputMask[ (int) i / longsize ] =
          GlobalMarkingV.GinputMask[ (int) i / longsize ] &
          ~( 0x1 << ( i % longsize ) );
} /* -- remove_tokens()  */

/*
**        add_tokens( transition )
**
**        add_tokens() is called after a transition is enabled and is fired.
**      1) For each input arc to the transition - add # of tokens
**           corresponding to # of output arcs to each output place.
**      2) Set the masks for global and local marking vectors
*/

add_tokens( t )
int t;   /* The transition */
{
  int i;                        /* -- Counter      */

  for ( i = 0; i < LOCAL_PLACES; i++ )        /* -- Add local tokens  */
    if ( ( LocalMarkingV.marking[ i ] +=
      Transition[ t ].LOutputVector[ i ] ) > 0 )
      LocalMarkingV.LinputMask[ (int) i / longsize ] =
        LocalMarkingV.LinputMask[ (int) i / longsize ] |
          ( 0x1 << ( i % longsize ) ); /* -- Set mask          */

  if ( Transition[ t ].glob_out )              /* -- Add global tokens  */
    {
      for ( i = 0; i < GLOBAL_PLACES; i++ )
        if ( ( GlobalMarkingV.marking[ i ] +=
            Transition[ t ].GOutputVector[ i ] ) > 0 )
          GlobalMarkingV.GinputMask[ (int) i / longsize ] =
            GlobalMarkingV.GinputMask[ (int) i / longsize ] |
            (0x1 << ( i % longsize ) );
      GlobalMarkingV.update = 1;
    }
} /* -- add_tokens()  */

static int ctr = 0;                /* -- points to the next output buffer for
                                        of tr_buffer  */

/* -- Buffer for sending the # of firings and local and global marking  */
short info_buffer[ 1 + NO_TRANSITIONS + LOCAL_PLACES + GLOBAL_PLACES ];
/* ------------------------------------------------------------------------
**
**        flush_bufs()
**
**        Writes the tr_buffer and pl_buffer to the XDisplay server.
*/

flush_bufs()
{
  int i;

  if ( !( ctr ) )
    return( 0 );
```

```
  info_buffer[ 0 ] = HOSTNO;

  /* -- Copy local marking info  */
  for( i = 0; i < LOCAL_PLACES; i++ )
    info_buffer[ i + NO_TRANSITIONS + 1 ] = LocalMarkingV.marking[ i ];

  /* -- Copy global marking info  */
  for( i = 0; i < GLOBAL_PLACES; i++ )
    info_buffer[ i + NO_TRANSITIONS + LOCAL_PLACES + 1 ] =
      GlobalMarkingV.old[ i ];

  if ( ( write( out_X_win, info_buffer, sizeof( info_buffer ) ) ) == -1 )
    syserr( "write out_X_win" );
  ctr = 0;
#ifdef DEBUG
  printf("LM : ");
  for(i=0;i<LOCAL_PLACES;i++)
    printf("%d ", LocalMarkingV.marking[ i ]);
  printf("\n");
  printf("GM : ");
  for(i=0;i<GLOBAL_PLACES;i++)
    printf("%d ", GlobalMarkingV.old[ i ]);
  printf("\n");
#endif
}

/* ------------------------------------------------------------------------
**
**      reset_firing_count()
**
**      Sets the firing counter to zero.
**
*/

void
reset_firing_count()
{
  int i;

  for ( i = 0; i < NO_TRANSITIONS; i++ )
    info_buffer[ i + 1 ] = 0;
}

/* ------------------------------------------------------------------------
**
**        update_Xwindow()
**
**        Stores the firing sequence vector and sends it every TR_LIMIT firing
**      or when the token player is finished.
*/

update_Xwindow( tr_no, mode )
int tr_no, mode;
{
  switch ( mode )
    {
    case 1 :   /* -- Add tr_no to output buffer  */
      ctr++;
      if ( ctr <= TR_LIMIT )
info_buffer[ tr_no + 1 ]++;
      else
```

```
        {
          info_buffer[ tr_no + 1]++;
          flush_bufs();
        }
      break;
    case 2 :   /* -- Flush buffers */
      flush_bufs();
      break;
    default :
      break;
    }
}

/* ------------------------------------------------------------------------
**
**        close_all()
**
**        Send termination signal to timed_trans_handler and close all open
**      file descriptors.
*/

void
close_all()
{
  close( token_in );
  close( token_out );
}

/* ------------------------------------------------------------------------
**
**      alarm_handler()
**
**      Called every time the interrupt timer goes off.
*/

void alarm_handler()
{
  printf("+");
}


/* ------------------------------------------------------------------------
**
**        input_interrupt_handler()
**
**        When an I/O interrupt is issued, this routine is called.
**      The procedure sets a flag.
*/

static int count = 0;
void
input_interrupt_handler()
{
  switch ( count++ )
    {
    case 0 : printf( "\b-" ); break;
    case 1 : printf( "\b/" ); break;
    case 3 : printf( "\b|" ); break;
    case 4 : printf( "\b\\" ); break;
    case 5 : count = 0; break;
    }
```

```
}

/* ------------------------------------------------------------------------
**
**      init_socket()
**
**      Initializes the sockets for global marking vector token ring.
**
**      token_in is socket for the server of SERVERNAME that runs on this host
**      and reads the token message from SERVERNAME.
**
**      token_out is the socket for the client to CLIENTNAME and writes to
**      this.
*/

init_sockets()
{
  char temp;

  /* -- Connect to XDisplay  */
  out_X_win = client_setup( HOSTONAME, DISPLAYPORT );

#if NO_HOSTS > 1
#ifdef HOST0   /* -- This is token player # 0   */
  /* -- Setup SERVER connection for token player # 1   */
  token_in = serv_setup_intr( HOSTNAME, PLAYERPORT,
                              "Server waiting for pn1 ..." );
  setblock( token_in, FALSE );

  /* -- Wait until the last token player has opened its server for token
        player 0   */
  printf(" Hit Enter when last token player server is ready : \n");
  scanf( "%s", &temp );
  token_out = client_setup( SERVERNAME, PLAYERPORT );
#else    /* -- This is not token player 0  */
  token_out = client_setup( SERVERNAME, PLAYERPORT);
  token_in = serv_setup_intr( HOSTNAME, PLAYERPORT,
                              "Server waiting for pn(i+1)" );
  setblock( token_in, FALSE );
#endif
#endif
  command_in = serv_setup_intr( HOSTNAME, COMMANDPORT,
                                "Server waiting for XCommand ..." );
  setblock( command_in, FALSE );                  /*    receive commands from
                                                        XCommand */
#if HOSTNO == 0
#ifdef EXTERNAL_IO
  io_socket = serv_setup_intr( HOSTNAME, IOPORT,
                               "Server waiting for device driver program ..." );
  setblock(io_socket, FALSE );                 /*    receive commands from
                                                    device driver */
#endif
#endif

#if HOSTNO == 1
#ifdef EXTERNAL_IO
  io_socket = serv_setup_intr( HOSTNAME, IOPORT+OFFSET1,
                               "Server waiting for device driver program ..." );
  setblock(io_socket, FALSE );                 /*    receive commands from
                                                    device driver */
#endif
```

```
#endif

#if HOSTNO == 2
#ifdef EXTERNAL_IO
  io_socket = serv_setup_intr( HOSTNAME, IOPORT+OFFSET2,
                               "Server waiting for device driver program ..." );
  setblock(io_socket, FALSE );                    /*    receive commands from
                                                       device driver */
#endif
#endif

  signal( SIGIO, input_interrupt_handler );  /* -- Set up interrupt to be
                                                   issued when i/o occurs */
}       /* -- socketinit()   */

/* ------------------------------------------------------------------------
**        main()
**
**        main has the following structure:
**
**        1) Initialize net data structures
**        2) Establish communication to other players, XDisplay & XCommand
**      2.1) Establish communication with external machines to be controlled
**      3) Set up interrup timer for watchdog operation
**      4) Wait for event
**        4.1) If command = RUN; Run token player
**             until system is quiet (no further transitions fire)
**        4.2) Goto wait for event
**        5) If quit command issued, close sockets and exit
*/

main()
{       /* -- main() */

  init_net();                          /* -- Initialize net   */
  init_sockets();                      /* -- Initializes sockets for interhost
                                           token message handling  */
  init_elapsed_time();                 /* -- Set signal for interrupt timer  */
  signal(SIGALRM, alarm_handler);

#if NO_HOSTS > 1                       /* -- Set interrupt intervals   */
  set_intr(5,0);
#else
  set_intr(0,100000);
#endif

  command = HALT;
  while ( command != STOP )            /* -- Do until command == STOP  */
    {
      pause();
      fflush( stdout );
      get_event();
      switch ( command )
        {
        case RUN   : token_player();
                     break;
        case HALT :  break;
        case RESET : init_net();
                     command = HALT;
                     break;
        default : ;
```

```
        }
    }
  close_all();
  exit( 0 );
}
```

A.4.7 *event_handler.c* — Command and Marking Vector Handling

```
#include          <stdio.h>
#include          "player.h"
#include         "defs.h"

BOOLEAN single_step = FALSE;

extern struct        trans         Transition; /* -- Array of transitions  */
extern struct  GlobalMV GlobalMarkingV;  /* -- The global m.v.        */
extern int longsize;
extern int endpipe[];        /* -- File descriptors for pipe.  This pipe is
                                used for signalling end of timed trans.  */
extern int token_in;         /* -- File descriptor for receiving the gmv
                                the token from the previous player  */
extern int token_out;        /* -- File descriptor for sending the gmv to
                                the next token player */
extern int command_in;       /* -- Socket for receiving commands */
extern int io_socket;         /* -- Socket for receiving external events */
extern BOOLEAN timer_running;

enum   comd_type command; /* -- Type for possible command modes of player */
BOOLEAN event_flag = TRUE;
static int tr_pointer = 0;
long io_buffer[2];

/* ------------------------------------------------------------------------
**
**         get_next_trans()
**
**         returns the next transition # to be evaluated for firing.
**
**         First checks if a transition is on the event queue and returns
**         this if so.  Else returns the transition pointed to by tr_pointer.
**
**         If tr_pointer > NO_TRANSITIONS, then all transitions have been
**         evaluated without any firing.  (-1) is then returned to signal that
**         token player may hibernate until another event occurs.
*/

int get_next_transition()
{
  int tr_no;

  if ( event_flag )       /* -- Event has occured, check all transitions again*/
    {
      event_flag = FALSE;
      tr_pointer = 0;
    }

  if ( timer_running )                /* -- A timed transition may be ready to
                                           fire, check all transitions  */
    if ( ( tr_no = timer_end() ) != -1 )   /* -- Some transition is done  */
      return ( tr_no );
```

```
  if ( tr_pointer < NO_TRANSITIONS )
    return( tr_pointer++ );          /* -- Return transition and increment
                                        pointer */
  else                               /* -- All transitions have been
                                        checked and no one fired  */
    {
      tr_pointer = 0;                /* -- Reset counter and start new cycle*/
      return( -1 );
    }
}

/* ------------------------------------------------------------------------
**
**      decode_comd( buf )
**
**      Decodes the first short of the global marking vector.  When this is
**      not zero, a command is being passed around.
*/

void decode_comd( com )
long com;
{
  switch ( com )
    {
    case 1 :
      command = RUN; break;
    case 2 :
      command = HALT; break;
    case 21 :
      command = RUN; break;
    case 3 :
      command = RESET; break;
    case 4 :
      command = STOP; break;
    case 5 :
      if ( single_step )
        single_step = FALSE;
      else
        single_step = TRUE;
      break;
    case 6 :
      if ( single_step ) command = RUN; break;
    case 7 : break;   /* -- Redraw */
    case 8 : break;  /* -- Display Stats */
    case 9 : break;  /* -- rates / times  */
    case 10 :  /* -- reset stats  */
      reset_firing_count(); break;
    case 11 :  /* -- core dump  */
      dump(); break;
    }
}       /* -- decode_comd()  */

/* ------------------------------------------------------------------------
**
**          get_gmv( buffer )
**
**          When a gmv event is detected by event_handler, this routine
**          is called to form the global marking vector.
**
**          buffer[ 1 - n ] contains the global marking vector.
```

```
*/

void get_gmv( buffer )
long buffer[];
{
  int i;

  GlobalMarkingV.GM_av = 1;        /* -- GMV is avaliable now  */
  event_flag = TRUE;

  if ( GlobalMarkingV.update )
    {
      for ( i = 0; i < GLOBAL_PLACES; i++ )
        {
          GlobalMarkingV.marking[ i ] += buffer[ i  + MASK_BUFSIZE ];
          GlobalMarkingV.GinputMask[ (int) ( i / longsize ) ] =
            ( GlobalMarkingV.marking[ i ] > 0 ) ?
              GlobalMarkingV.GinputMask[ (int) ( i / longsize ) ] |
                ( 0x1 << i ) :
                  GlobalMarkingV.GinputMask[ (int) ( i / longsize ) ] &
                    ~( 0x1 << i );
        }
      GlobalMarkingV.update = 0;
    }
   else
    {
      for ( i = 0; i < GLOBAL_PLACES; i++ )
GlobalMarkingV.marking[ i ] = buffer[  i  + MASK_BUFSIZE ];
      for ( i = 0; i < (int) GLOBAL_PLACES / longsize + 1; i++ )
GlobalMarkingV.GinputMask[ i ] = buffer[ i ];
    }
}

/* ------------------------------------------------------------------------
**
**          release_gmv()
**
**          After the token player is done with the global marking vector,
**          this is released to the next host.
*/

void release_gmv()
{
  long buffer[ GMV_BUFSIZE ];
  int i;
#if NO_HOSTS > 1
  if ( GlobalMarkingV.GM_av )  /* -- Only send gmv when it is at this
                                     player  */
    {
      buffer[ 0 ] = 0;
      GlobalMarkingV.GM_av = 0;
      GlobalMarkingV.update = 0;
      /* -- Copy mask to buffer  */
      for ( i = 0; i < (int) GLOBAL_PLACES / longsize + 1; i++ )
        {
          buffer[ i ] = GlobalMarkingV.GinputMask[ i ];
          GlobalMarkingV.GinputMask[ i ] = 0;
        }
      /* -- Copy global marking vector  */
      for ( i = 0; i < GLOBAL_PLACES; i++ )
        {
```

```
        buffer[ MASK_BUFSIZE + i ] = GlobalMarkingV.marking[ i ];
        GlobalMarkingV.old[ i ] = GlobalMarkingV.marking[ i ];
        GlobalMarkingV.marking[ i ] = 0;
      }
    /* -- Send global marking vector to next token player */
    if ( write ( token_out, buffer, sizeof( buffer ) ) == -1 )
      syserr( "write token_out" );
    update_Iwindow( 0, 2 );    /* -- Flush firing sequence and marking
                                  vectors to the XDisplay  */
  }
#endif
}

/* ------------------------------------------------------------------------
**
**         get_event()
**
**         Reads all the input sockets and pipes.  If there is any info,
**         decodes this.
**
**         Reads:
**         - command_in socket from XCommand window
**         - token_in socket from the previous token player
**         - io_socket socket from device driver program.
*/

void get_event()
{
  long token_buffer[ GMV_BUFSIZE ];/* -- Buffer for received marking vector */
  long command_buffer[ 2 ];
  int nread, i;

#if HOSTNO == 0
#ifdef EXTERNAL_IO
  /* -- Check if there is anything from the device driver program to read  */
  if ( nread = read( io_socket, io_buffer, sizeof( io_buffer ) )
       > 0 )
    decode_io();
#endif
#endif

#if HOSTNO == 1
#ifdef EXTERNAL_IO
  /* -- Check if there is anything from the device driver program to read  */
  if ( nread = read( io_socket, io_buffer, sizeof( io_buffer ) )
       > 0 )
    decode_io();
#endif
#endif

#if HOSTNO == 2
#ifdef EXTERNAL_IO
  /* -- Check if there is anything from the device driver program to read  */
  if ( nread = read( io_socket, io_buffer, sizeof( io_buffer ) )
       > 0 )
    decode_io();
#endif
#endif

  /* -- Check if there is a command available to read  */
  if ( nread = read( command_in, command_buffer, sizeof( command_buffer ) )
```

```
      > 0 )
    decode_comd( command_buffer[ 0 ] );

#if NO_HOSTS > 1
  /* -- Check if it is the global marking vector that is available  */
  if ( nread = read( token_in, token_buffer, sizeof( token_buffer ) ) > 0 )
    {
      get_gmv( token_buffer );    /* -- gmv available */
#ifdef DEBUG
      printf("gmv read %d bytes : ",nread);
      for (i = 0; i < sizeof(token_buffer)/4; i++)
        printf(" %d",token_buffer[i]);
      printf("\n");
#endif
    }
#endif
}
```

A.4.8 *init_net.c* — Intitalizes Data Structures

```
#include        "player.h"
#include        <stdio.h>
#include        <errno.h>
#include        "defs.h"
#include        <string.h>

extern struct trans Transition[NO_TRANSITIONS];   /* -- Transitions  */
extern struct LocalMV  LocalMarkingV;         /* -- The local marking v.   */
extern struct GlobalMV GlobalMarkingV;        /* -- The global m.v.        */

/*  ------------------------------------------------------------------------
**
**        init_links()
**
**        Initializes the links to transition preconditions and postprocessing
**      routines.
*/

#include DRIVERFILE

/*  ------------------------------------------------------------------------
**
**        init_net()
**
**        init initializes the datastructures that contain the net definition
*/

void init_net()
{
        int         i, j;                 /* -- Loop counter                */
        printf("init called\n");

        /* --  Use host number to specify a seed between 0 and 31 */
        init_elapsed_time();
        usleep(1000);
        rnd_init( elapsed_time() % 31 );
        init_elapsed_time();
        for (i = 0; i < NO_TRANSITIONS; i++)
          {
            Transition[i].firing = 0;
```

```
            Transition[i].glob_in = 0;
            Transition[i].glob_out = 0;
            Transition[i].LInhArc = 0;
            Transition[i].GInhArc = 0;
            Transition[i].immediate = 0;
            Transition[i].timed = 0;
            Transition[i].preconditions = 0;
            Transition[i].postprocessing = 0;
            Transition[i].firing_rate = 1.0;
            Transition[i].preprocess = NULL;
            Transition[i].postprocess = NULL;
            strcpy( Transition[i].tag, "" );
            for ( j = 0; j < LOCAL_PLACES; j++ )
              {
                Transition[i].LInputVector[j] = 0;
                Transition[i].LOutputVector[j] = 0;
                Transition[i].LInhibitVector[j] = 0;
              }
            for ( j = 0; j < LOCAL_MASK_SIZE; j++ )
              Transition[i].LInputMask[j] = 0;
            for ( j = 0; j < GLOBAL_PLACES; j++ )
              {
                Transition[i].GInputVector[j] = 0;
                Transition[i].GOutputVector[j] = 0;
                Transition[i].GInhibitVector[j] = 0;
              }
            for ( j = 0; j < GLOBAL_MASK_SIZE; j++ )
              Transition[i].GInputMask[j] = 0;
          }
        for ( i = 0; i < LOCAL_PLACES; i++ )
          {
            LocalMarkingV.marking[i] = 0;
            strcpy( LocalMarkingV.tag[i], "" );
          }
        for ( i = 0; i < GLOBAL_PLACES; i++ )
          {
            GlobalMarkingV.update = 0;
            GlobalMarkingV.GM_av = 0;
            GlobalMarkingV.marking[i] = 0;
            strcpy( GlobalMarkingV.tag[i], "" );
          }
#include NETFILE

#include LINKFILE
      }
```

A.4.9 *intr_timer.c* — Sets up the Interrupt Timer

```
#include <sys/time.h>
#include <stdio.h>
#include <signal.h>

/* ------------------------------------------------------------------------
**
**          set_intr( sec, usec )
**
**          Sets the interval for the timer to go off in sec.usec
**
*/
```

```
void set_intr( sec, usec )
long sec, usec;
{
  struct itimerval ti;

  if ( ( 10000 + usec ) >= 1000000 )
    {
      ti.it_value.tv_sec = 1 + sec;
      ti.it_value.tv_usec = 10000 + usec - 1000000;
    }
  else
    {
      ti.it_value.tv_sec = sec;
      ti.it_value.tv_usec = 10000 + usec;
    }

  ti.it_interval.tv_sec = sec;
  ti.it_interval.tv_usec = usec;

  if ( setitimer(ITIMER_REAL, &ti, (struct timeval*)0 ) )
    syserr("setitimer");
}
```

A.4.10 *timed_trans_handler.c* — Checks Timer for Transitions

```
#include        <sys/time.h>
#include        <sys/types.h>
#include        <sys/times.h>

#include        <stdio.h>
#include        <errno.h>
#include        "player.h"
#include        "defs.h"
#include        <math.h>

static double TICKS =  1000.0;  /* -- Number of ticks per sec. system clock */
static double SCALE = 1.0;  /* -- Time scale for simulator:
                               1/10 seconds : SCALE = .1
                               seconds      : SCALE = 1.0
                               minutes      : SCALE = 60.0
                               hours        : SCALE = 3600.0
                               days         : SCALE = 86400.0
                            */

extern struct trans Transition[];

extern unsigned long elapsed_time();

double log(), rnd_01d();

BOOLEAN timer_running = FALSE;

/*  ------------------------------------------------------------------------
**
**      timer_end()
**
**      Checks all firing transitions (timed or exponential) to see if the
**      end_time is less than the actual time.
**
*/
```

C-2

```
int
timer_end()
{
  int i;

  timer_running = FALSE;

  for ( i = 0; i < NO_TRANSITIONS; i++ )
    {
      if ( Transition[ i ].firing )
        {
          timer_running = TRUE;
          if ( elapsed_time() >= Transition[ i ].end_time )
            {
              Transition[ i ].end_time = 0;
              return( i );
            }
        }
    }
  return ( -1 );
}


/*  ------------------------------------------------------------------------
**
**          start_timer( tr_no )
**
**          Starts the timer for a timed transition.  The time is fixed
**          and is the inverse of the firing rate if the transition has
**          deterministic firing time.
**          For exponential transitions, the firing time is based on computing
**          the inverse of the cumulative distribution function given a random
**          value between [0, 1] and scaling this using the firing rate of the
**          transition.
**
*/

void
start_timer( tr_no )
int tr_no;
{

  unsigned long tstart;
  double rnd;
  double time, rate;

  rate = (double) Transition[ tr_no ].firing_rate;
  switch ( Transition[ tr_no ].timed )
    {
    case 0 :    /* -- Exponential  */
      tstart = elapsed_time();
      rnd = rnd_01d();
      time = SCALE * (-1) * log( rnd ) * TICKS / rate;
      Transition[ tr_no ].end_time =  tstart + (unsigned long) time;
#ifdef DEBUG2
      printf("\n%s -- R#: %f -> Exp %f ms\n", Transition[tr_no].tag, rnd, time);
#endif
      break;
    case 1 :    /* -- Timed        */
      tstart = elapsed_time();
```

```
      time = (SCALE * TICKS) / rate;
#ifdef DEBUG2
      printf("\n%s -- Det %f ms\n", Transition[tr_no].tag, time);
#endif
      Transition[ tr_no ].end_time = tstart + (unsigned long) time;
      break;
    default : ;
    }
  timer_running = TRUE;
}
```

A.4.11 *timer.c* — Timer Routines

```
#include        <sys/time.h>
#include        <sys/types.h>
#include        <sys/times.h>

#include        <stdio.h>
#include        <errno.h>
#include        "player.h"
#include        "defs.h"

/* ------------------------------------------------------------------------
**
**        init_elapsed_time()
**
**        Resets the timer.
**
*/

static struct timeval time_0;

void init_elapsed_time()
{
  gettimeofday(&time_0, (struct timezone*)0);
}

/*  ------------------------------------------------------------------------
**
**        elapsed_time()
**
**        Returns the time of the internal clock in 1/1000th of seconds.
**
*/

unsigned long
elapsed_time()
{
  struct timeval tp;
  unsigned long i;
  gettimeofday(&tp, (struct timezone*)0);
  i = (tp.tv_sec-time_0.tv_sec)*1000 + (tp.tv_usec-time_0.tv_usec)/1000;
  return i;
}


/*  ------------------------------------------------------------------------
**
**        elapsed_time_sec()
**
```

```
**        Returns the time of the internal clock in seconds.
**
*/

unsigned long
elapsed_time_sec()
{
  struct timeval tp;
  unsigned long i;
  gettimeofday(&tp, (struct timezone*)0);
  i = (tp.tv_sec-time_0.tv_sec);
  return i;
}
```

A.4.12 *rrand.c* — Random Number Generator for Simulation

```
#include <math.h>

/* Random number generators:
 *
 *  rnd_init (unsigned seed)
 *                        : initializes the generator
 *  rnd_01d ()                : returns doubles              [0.0,1.0)
 *                      Note: ")" is no typo - rnd_01d will not return a 1.0,
 *                              but can return the next smaller FP number.

 *  Algorithm M as describes in Knuth's "Art of Computer Programming",
 *  Vol 2. 1969
 *  is used with a linear congruential generator (to get a good uniform
 *  distribution) that is permuted with a Fibonacci additive congruential
 *  generator to get good independence.
 *
 *  Bit, byte, and word distributions were extensively tested and pass
 *  Chi-squared test near perfect scores (>7E8 numbers tested, Uniformity
 *  assumption holds with probability > 0.999)
 *
 *  Run-up tests for on 7E8 numbers confirm independence with
 *  probability > 0.97.
 *
 *  Plotting random points in 2d reveals no apparent structure.
 *
 *  Autocorrelation on sequences of 5E5 numbers (A(i) = SUM X(n)*X(n-i),
 *  i=1..512)
 *  results in no obvious structure (A(i) ~ const).
 *
 *  On a SUN 3/60, rnd_u() takes about 19.4 usec per call, which is about 44%
 *  slower than Berkeley's random() (13.5 usec/call).
 *
 *  Except for speed and memory requirements, this generator outperforms
 *  random() for all tests. (random() scored rather low on uniformity tests,
 *  while independence test differences were less dramatic).
 *
 *  Thanks to M.Mauldin, H.Walker, J.Saxe and M.Molloy for inspiration & help.
 *
 *  (c) Copyright 1988 by A. Nowatzyk
 * -- Routines not used removed from file , 1/29/91  A.B.
 */

/* LC-parameter selection follows recommendations in
 * "Handbook of Mathematical Functions" by Abramowitz & Stegun 10th, edi.
```

```
 */
#define LC_A 66049          /* = 251^2, ~= sqrt(2^32) */
#define LC_C 3907864577     /* result of a long trial & error series    */

#define Xrnd(x) (x * LC_A + LC_C)   /* the LC polynomial  */

static unsigned long Fib[55];  /* will use X(n) = X(n-55) - X(n-24)        */
static int Fib_ind;            /* current index in circular buffer   */
static unsigned long Xrnd_var; /* LCA - recurrence variable              */
static unsigned long auxtab[256];   /* temporal permutation table    */
static unsigned long prmtab[64] = { /* spatial permutation table   */
    0xffffffff, 0x00000000,  0x00000000,  0x00000000,  /* 3210 */
    0x0000ffff, 0x00ff0000,  0x00000000,  0xff000000,  /* 2310 */
    0xff0000ff, 0x0000ff00,  0x00000000,  0x00ff0000,  /* 3120 */
    0x00ff00ff, 0x00000000,  0xff00ff00,  0x00000000,  /* 1230 */

    0xffff0000, 0x000000ff,  0x00000000,  0x0000ff00,  /* 3201 */
    0x00000000, 0x00ff00ff,  0x00000000,  0xff00ff00,  /* 2301 */
    0xff000000, 0x00000000,  0x000000ff,  0x00ffff00,  /* 3102 */
    0x00000000, 0x00000000,  0x00000000,  0xffffffff,  /* 2103 */

    0xff00ff00, 0x00000000,  0x00ff00ff,  0x00000000,  /* 3012 */
    0x0000ff00, 0x00000000,  0x00ff0000,  0xff0000ff,  /* 2013 */
    0x00000000, 0x00000000,  0xffffffff,  0x00000000,  /* 1032 */
    0x00000000, 0x0000ff00,  0xffff0000,  0x000000ff,  /* 1023 */

    0x00000000, 0xffffffff,  0x00000000,  0x00000000,  /* 0321 */
    0x00ffff00, 0xff000000,  0x00000000,  0x000000ff,  /* 0213 */
    0x00000000, 0xff000000,  0x0000ffff,  0x00ff0000,  /* 0132 */
    0x00000000, 0xff00ff00,  0x00000000,  0x00ff00ff   /* 0123 */
};

union hack {                /* used to access doubles as unsigneds        */
    double d;
    unsigned long u[2];
};
static union hack man;      /* mantissa bit vector  */
rnd_init (seed)             /* modified: seed 0-31 use precomputed stuff */
    unsigned seed;
{
    register unsigned long u;
    register int i;
    double x, y;
    union hack t;

    static unsigned seed_tab[32] = {
                0xbdcc47e5, 0x54aea45d, 0xec0df859, 0xda84637b,
                0xc8c6cb4f, 0x35574b01, 0x28260b7d, 0x0d07fdbf,
                0x9faaeeb0, 0x613dd169, 0x5ce2d818, 0x85b9e706,
                0xab2469db, 0xda02b0dc, 0x45c60d6e, 0xf1fe49d10,
                0x7224fea3, 0xf9684fc9, 0xfc7ee074, 0x326ce92a,
                0x366d13b6, 0x17aaa731, 0xeb83a675, 0x7781cb32,
                0x4ec7c92d, 0x7f187521, 0x2cf346b4, 0xad13310f,
                0xb89cff2b, 0x12164de1, 0xa865168d, 0x32b56cdf  };

    if (seed < 32)
        u = seed_tab[seed];
    else
        u = seed ^ seed_tab[seed & 31];
    for (i = 55; i--;)      /* set up Fibonacci additive congruential        */
        Fib[i] = u = Xrnd(u);
```

```
    for (i = 256; i--;)
        aurtab[i] = u = Xrnd(u);
    Fib_ind = u % 55;      /* select a starting point                    */
    Xrnd_var = u;
    if (sizeof(x) != 2 * sizeof(unsigned long)) {
        x = 0.0;
        y = 1.0;
        y /= x;                 /*** intentional divide by 0: rnd_01d will
                                     not work because a double doesn't fit
                                     in 2 unsigned longs on your machine! ***/
    };
    x = 1.0;
    y = 0.5;
    do {        /* find largest fp-number < 2.0                           */
        t.d = x;
        x += y;
        y *= 0.5;
    } while (x != t.d && x < 2.0);
    man.d = 1.0;
    man.u[0] ^= t.u[0];
    man.u[1] ^= t.u[1];   /* man is now 1 for each mantissa bit           */
}

unsigned long rnd_u ()
/*
 * same as rnd_i, but gives full 32 bit range
 */
{
    register unsigned long i, j, *t = Fib;

    i = Fib_ind;
    j = t[i];                                          /* = X(n-55) */
    j -= (i >= 24) ? t[i - 24] : t[i + 21]; /* = X(n-24) */
    t[i] = j;
    if (++i >= 55) i = 0;
    Fib_ind = i;

    t = &aurtab[(j >> 24) & 0xff];
    i = *t;
    Xrnd_var = *t = Xrnd(Xrnd_var);
    t = &prmtab[j & 0x3c];

    j =  *t++ & i;
    j |= *t++ & ((i << 24) | ((i >>  8) & 0x00ffffff));
    j |= *t++ & ((i << 16) | ((i >> 16) & 0x0000ffff));
    j |= *t   & ((i <<  8) | ((i >> 24) & 0x000000ff));

    return j;
}

long rnd_ri (rng)
    long rng;
/*
 * randint: Return a random integer in a given Range [0..rng-1]
 *          Note:  0 < rng
 */
{
    register unsigned long  r, a;

    do {
        r = rnd_i();
```

```
        a = (r / rng) + 1;
        a *= rng;
    } while (a >= 0x7fffffff);

    a--;
    return a - r;
}

double rnd_01d ()
/*
 * returns a uniformly distributed double in the range of [0..1)
 *         or  0.0 <= rnd_01d() < 1.0 to be precise
 *
 * Note: this code assumes that 2 'unsigned long's can hold a 'double'
 *       (works on SUN-3's, SUN-4's, MIPS, VAXen, IBM RT's)
 */
{
    union hack t;

    t.d = 1.0;

    t.u[0] |= rnd_u() & man.u[0];          /* munch in 1st part   */
    t.u[1] |= rnd_u() & man.u[1];          /* munch in 2nd part   */

    return t.d - 1.0;
}
```

A.4.13 *syserr.c* and *setblock.c* — System Error and IO Blocking

```
#include <stdio.h>

void syserr( msg )  /* print system call error message and terminate */
char *msg;
{
  extern int errno, sys_nerr;
  extern char *sys_errlist[];

  fprintf( stderr, "ERROR: %s (%d", msg, errno );
  if ( errno > 0 && errno < sys_nerr )
    fprintf( stderr, "; %s)\n", sys_errlist[ errno ] );
  else
    fprintf( stderr, ")\n" );
  exit( 1 );
}

#include        <stdio.h>
#include        <errno.h>
#include        <fcntl.h>
#include        "defs.h"

/*
**      setblock()
**
**      setblock() turns blocking on or off for a given fd.
**
*/

void setblock( fd, on )
int fd;
BOOLEAN on;
```

```
{
  static int blockf, nonblockf;
  static BOOLEAN first = TRUE;
  int flags;

  if( first )
    {
      first = FALSE;
      if( ( flags = fcntl( fd, F_GETFL, 0 ) ) == -1 )
        syserr( "fcntl" );
      blockf = flags & ~O_NDELAY;     /* -- O_NDELAY is off  */
      nonblockf = flags | O_NDELAY;   /* -- O_NDELAY is on   */
    }
  if( fcntl( fd, F_SETFL, on ? blockf : nonblockf ) == -1 )
    syserr( "fcntl2" );

}       /* -- setblock()  */
```

A.4.14 *dump.c* — Prints the Current Net Data on the Terminal

```
#include        "player.h"
#include        <stdio.h>
#include        <errno.h>
#include        "defs.h"
#include        <string.h>

extern struct trans   Transition[NO_TRANSITIONS];   /* -- Transitions  */
extern struct LocalMV  LocalMarkingV;               /* -- The local marking v.   */
extern struct GlobalMV GlobalMarkingV;              /* -- The global m.v.        */

/* ------------------------------------------------------------------------
**
**        dump.c
**
**        Dumps all data to the screen.
*/

void dump()
{
  int i, j;
  for (i = 0; i < NO_TRANSITIONS; i++)
    {
      printf("T[%d].firing = %d\n",i,Transition[i].firing);
      printf("T[%d].glob_in = %d\n",i,Transition[i].glob_in);
      printf("T[%d].glob_out = %d\n",i,Transition[i].glob_out);
      printf("T[%d].LInhArc = %d\n",i,Transition[i].LInhArc);
      printf("T[%d].GInhArc = %d\n",i,Transition[i].GInhArc);
      printf("T[%d].immediate = %d\n",i,Transition[i].immediate);
      printf("T[%d].timed = %d\n",i,Transition[i].timed);
      printf("T[%d].preconditions = %d\n",i,Transition[i].preconditions);
      printf("T[%d].postprocessing = %d\n",i,Transition[i].postprocessing);
      printf("T[%d].end_time = %d\n",i,Transition[i].end_time);
      printf("T[%d].firing_rate = %f\n",i,Transition[i].firing_rate);
      printf("T[%d].tag = %s\n",i, Transition[i].tag);
      printf("T[%d].LInputVector = ",i);
      for ( j = 0; j < LOCAL_PLACES; j++ )
        printf("%d ", Transition[i].LInputVector[j]);
      printf("\n");
      printf("T[%d].LOutputVector = ",i);
      for ( j = 0; j < LOCAL_PLACES; j++ )
```

```
      printf("%d ", Transition[i].LOutputVector[j]);
    printf("\n");
    printf("T[%d].LInhibitVector = ",i);
    for ( j = 0; j < LOCAL_PLACES; j++ )
      printf("%d ", Transition[i].LInhibitVector[j]);
    printf("\n");
    printf("T[%d].GInhibitVector = ",i);
    for ( j = 0; j < LOCAL_PLACES; j++ )
      printf("%d ", Transition[i].GInhibitVector[j]);
    printf("\n");
    printf("T[%d].LInputMask = ",i);
    for ( j = 0; j < LOCAL_MASK_SIZE; j++ )
      printf("%x ",Transition[i].LInputMask[j]);
    printf("\n");
    printf("T[%d].GInputVector = ",i);
    for ( j = 0; j < GLOBAL_PLACES; j++ )
      printf("%d ", Transition[i].GInputVector[j]);
    printf("\n");
    printf("T[%d].GOutputVector = ",i);
    for ( j = 0; j < GLOBAL_PLACES; j++ )
      printf("%d ", Transition[i].GOutputVector[j]);
        printf("\n");
    printf("T[%d].GInputMask = ",i);
    for ( j = 0; j < GLOBAL_MASK_SIZE; j++ )
      printf("%x ",Transition[i].GInputMask[j]);
    printf("\n");
  }
 printf("Local Marking = ");
 for ( i = 0; i < LOCAL_PLACES; i++ )
   printf("%x ", LocalMarkingV.marking[i]);
 printf("\n");
 printf("Local Tags =  ");
 for ( i = 0; i < LOCAL_PLACES; i++ )
   printf("%s ", LocalMarkingV.tag[i]);
 printf("\nGlobal Marking = ");
 for ( i = 0; i < GLOBAL_PLACES; i++ )
   printf("%d ", GlobalMarkingV.marking[i]);
 printf("\n");
 printf("Global Tags = ");
 for ( i = 0; i < GLOBAL_PLACES; i++ )
   printf("%s ", GlobalMarkingV.tag[i]);
 printf("\n");
 printf("GLOBALMARKINGV.update = %d\n", GlobalMarkingV.update);
 printf("GLOBALMARKINGV.GM_av = %d\n", GlobalMarkingV.GM_av);
}
```

A.4.15 *server_intr.c* — Sets up Server Socket

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <fcntl.h>
#include <errno.h>
#include <signal.h>

serv_setup_intr(myname, port_num, message)
char *myname;
u_short port_num;
char *message;
```

```
{
        struct sockaddr_in self, from;
        int s, fromlen, ns;

        printf("Using port num %d\n", port_num);
        bzero((char *)&self, sizeof(self));
        self.sin_family = AF_INET;
        self.sin_addr.s_addr = INADDR_ANY;
        self.sin_port = htons((u_short)port_num);
        /* if ((self.sin_addr.s_addr = rhost(&myname)) < 0) {
                fprintf("Can not determine address of server\n");
                exit(-1);
        } */

        if ((s = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
                perror("Could not obtain socket");
                exit(-2);
        }

        if (fcntl(s,F_SETFL, FNBIO)<0){
                perror("fcntl F_SETFL, FNBIO");
                exit(1);
        }

        if (setsockopt(s, SOL_SOCKET, SO_REUSEADDR, (char *)0, 0) == -1) {
                perror("setsockopt: SO_REUSEADDR");
                close(s);
                exit(-4);
        }
        if (setsockopt(s, SOL_SOCKET, SO_DONTLINGER, (char *)0, 0) == -1) {
                perror("setsockopt: SO_REUSEADDR");
                close(s);
                exit(-4);
        }

        if(bind(s, (struct sockaddr *)&self, sizeof(self))) {
                perror("Bind failure");
                close(s);
                exit(-3);
        }

        if (listen(s, 5)) {
                perror("Error in listen");
                exit(-1);
        }

        bzero((char *)&from, sizeof(from));
        from.sin_family = AF_INET;
        from.sin_addr.s_addr = INADDR_ANY;
        fromlen = sizeof(from);
        printf( "%s\n", message );
        if ((ns = accept(s, (struct sockaddr *)&from, &fromlen)) < 0) {
                perror("Accept failed");
                exit(-3);
        }

        if (fcntl(ns,F_SETOWN, getpid())<0){
          perror("fcntl F_SETOWN, :");
          exit(1);
        }
        if (fcntl(ns,F_SETFL, FASYNC|FNBIO)<0){
```

```
            perror("fcntl F_SETFL, FASYNC");
            exit(1);
        }

        printf("Connection has been established\n");
        shutdown(s,2);
        close(s);
        return(ns);
}
```

A.4.16 *client.c* — Sets up Client Socket

```
#include <stdio.h>
#include <errno.h>
#include <netdb.h>

#include <sys/types.h>
#include <sys/socket.h>

#include <netinet/in.h>

client_setup(host,port_num)
char *host;
int port_num;
{
        struct hostent *hp;
        struct sockaddr_in to;
        int s;
        int lport = IPPORT_RESERVED;

        bzero((char *)&to, sizeof(struct sockaddr_in));

        if ((s = socket(AF_INET, SOCK_STREAM, 0))< 0) {
                perror("Can not obtain socket");
                exit(1);
        }
        printf("Client Socket is %d\n", s);

        if ((hp = gethostbyname(host)) == NULL) {
                fprintf(stderr, "Can not determine server host's address\n");
                close(s);
                exit(1);
        }

        to.sin_port = htons((u_short)port_num);
        to.sin_family = hp->h_addrtype;
        bcopy(hp->h_addr, (char *)&to.sin_addr, sizeof(to.sin_addr));

        if (connect(s, (struct sockaddr *)&to, sizeof(to)) < 0) {
                perror("Connection failed");
                close(s);
                exit(1);
        }
        return(s);
}
```

A.5 *XCommand* — Command Menu Program

A.5.1 *mXc* — Make File for *XCommand*

```
##
##
##
EXEC=        XCommand
##
##        -g = Debugging info
##        -O = Optimize
CFLAGS=        -g
##
##        Libraries
##        Xil        Xil graphics library
##
LIBS= -lXaw -lXmu -lXt -lXil
#
#
OBJECTS=        XCommand.c        \
                setblock.c        \
                client.c        \
                syserr.c
##
##        Compile & link
##
$(EXEC):        $(OBJECTS)
                cc $(CFLAGS) -o $(EXEC) $(OBJECTS) $(LIBS)
##
##
##        End of make file
##
```

A.5.2 *XCommand.c* — main() for Program *XCommand*

```
#include "size_limits.h"
#include "portnums.h"
#include <stdio.h>
#include <X11/Intrinsic.h>
#include <X11/StringDefs.h>
#include <X11/Xaw/List.h>
#include <X11/Xaw/Command.h>
#include <X11/Xaw/Cardinals.h>
#include <X11/Xaw/Paned.h>
#include <X11/Xaw/AsciiText.h>

static void Run(), Halt(), Cont(), Reset(), Quit(), SingleStep(), Fire(),
  Redraw(), StatsRates(), StatsTimes(), ResetStats(), Dump();
int command_socket;
char HOSTNAMES[ MAXHOST ] [ MAXLEN ];
int NO_HOSTS;
int out_command[ MAXHOST ];

main()
{
  int argc;
  char **argv;
  int i;
  char buf[ 100 ];
  FILE *in_f, *fopen();
```

```
Widget toplevel, run, halt, cont, reset, quit, singlestep, fire, redraw,
       paned, statsrates, statstimes, resetstats, dump;
XtAppContext app_con;
Arg args[1];

if ( ( in_f = fopen("XCommand.i", "r" ) ) == -1 )
  syserr( "open XCommand.i" );

/* -- read # of hosts  */
fgets( buf, sizeof( buf ), in_f );
sscanf( buf, "%d", &NO_HOSTS );

/* -- Read each host name  */
for ( i = 0; i < NO_HOSTS + 1; i++ )
  {
    fgets( buf, sizeof( buf ), in_f );
    sscanf( buf, "%s", HOSTNAMES[ i ] );
  }
close ( in_f );

/* -- Connect to token players  */
for ( i = 0; i < NO_HOSTS; i++ )
  out_command[ i ] =  client_setup( HOSTNAMES[ i ], COMMANDPORT );

/* -- Connect to Display window  */
out_command[NO_HOSTS] =  client_setup( HOSTNAMES[ NO_HOSTS ], DISPLAYPORT);

toplevel = XtAppInitialize(&app_con, "Xlist", NULL, ZERO,
                           (Cardinal *) &argc, argv, NULL, NULL, ZERO);
paned = XtCreateManagedWidget("paned", panedWidgetClass, toplevel,
                              NULL, ZERO);

run = XtCreateManagedWidget("start", commandWidgetClass, paned,
                            NULL, ZERO);
halt = XtCreateManagedWidget("halt", commandWidgetClass, paned,
                            NULL, ZERO);
cont = XtCreateManagedWidget("continue", commandWidgetClass, paned,
                            NULL, ZERO);
reset = XtCreateManagedWidget("reset", commandWidgetClass, paned,
                            NULL, ZERO);
quit = XtCreateManagedWidget("quit", commandWidgetClass, paned,
                           NULL, ZERO);
singlestep = XtCreateManagedWidget("single step", commandWidgetClass, paned,
                            NULL, ZERO);
fire = XtCreateManagedWidget("fire", commandWidgetClass, paned,
                            NULL, ZERO);
redraw = XtCreateManagedWidget("redraw", commandWidgetClass, paned,
                            NULL, ZERO);
statsrates = XtCreateManagedWidget("   display stats  ", commandWidgetClass,
                              paned, NULL, ZERO);
statstimes = XtCreateManagedWidget("rates / times fired", commandWidgetClass,
                              paned, NULL, ZERO);
resetstats = XtCreateManagedWidget("reset stats", commandWidgetClass, paned,
                            NULL, ZERO);
dump = XtCreateManagedWidget("Net Listing", commandWidgetClass, paned,
                            NULL, ZERO);

XtAddCallback(run, XtNcallback, Run, (XtPointer)NULL);
XtAddCallback(halt, XtNcallback, Halt, (XtPointer)NULL);
XtAddCallback(cont, XtNcallback, Cont, (XtPointer)NULL);
```

```
  XtAddCallback(reset, XtNcallback, Reset, (XtPointer)NULL);
  XtAddCallback(quit, XtNcallback, Quit, (XtPointer)NULL);
  XtAddCallback(singlestep, XtNcallback, SingleStep, (XtPointer)NULL);
  XtAddCallback(fire, XtNcallback, Fire, (XtPointer)NULL);
  XtAddCallback(redraw, XtNcallback, Redraw, (XtPointer)NULL);
  XtAddCallback(statsrates, XtNcallback, StatsRates, (XtPointer)NULL);
  XtAddCallback(statstimes, XtNcallback, StatsTimes, (XtPointer)NULL);
  XtAddCallback(resetstats, XtNcallback, ResetStats, (XtPointer)NULL);
  XtAddCallback(dump, XtNcallback, Dump, (XtPointer)NULL);
  XtRealizeWidget(toplevel);
  XtAppMainLoop(app_con);
}

/*
**      write_com( command )
**
**      Writes the command to all the sockets
*/

void write_com( command )
long command;
{
  long buffer[ 2 ];
  int i;

  /* -- write to the token players */
  buffer[ 0 ] = command;
  for ( i = 0; i < NO_HOSTS ; i++ )
    if ( write (out_command[ i ], buffer, sizeof(buffer) ) == -1 )
      syserr( "write out_command□" );

  /* -- Write to the XDisplay server  */
  buffer[ 1 ] = command;
  buffer[ 0 ] = NO_HOSTS;
  if ( write (out_command[ i ], buffer, sizeof(buffer) ) == -1 )
    syserr( "write out_command□" );
}

/*        Function Name: Run
 *        Description: Issues RUN command
 */

static void Run(widget, client_data, call_data)
Widget        widget;
XtPointer client_data, call_data;
{
  write_com( 1 );
}

/*        Function Name: Halt
 *        Description: Issues Halt command
 */

static void Halt(widget, client_data, call_data)
Widget        widget;
XtPointer client_data, call_data;
{
  write_com( 2 );
}

/*        Function Name: Cont
```

```
 *        Description: Issues Continue command
 */

static void Cont(widget, client_data, call_data)
Widget        widget;
XtPointer client_data, call_data;
{
  write_com( 21 );
}

/*        Function Name: Reset
 *        Description: Issues Reset command
 */

static void Reset(widget, client_data, call_data)
Widget        widget;
XtPointer client_data, call_data;
{
  write_com( 3 );
}

/*        Function Name: Quit
 *        Description: exits the application.
 */

static void Quit(widget, client_data, call_data)
Widget        widget;
XtPointer client_data, call_data;
{
  int i;
  write_com( 4 );
  for ( i = 0; i < NO_HOSTS + 1; i++ )
    close( out_command[ i ] );

  XtDestroyApplicationContext(XtWidgetToApplicationContext(widget));
  exit(0);
}

/*        Function Name: SingleStep
 *        Description: Issues SingleStep command
 */
static void SingleStep(widget, client_data, call_data)
Widget        widget;
XtPointer client_data, call_data;
{
  write_com( 5 );
}

/*        Function Name: Fire
 *        Description: Issues Fire command
 */

static void Fire(widget, client_data, call_data)
Widget        widget;
XtPointer client_data, call_data;
{
  write_com( 6 );
}

/*        Function Name: Redraw
 *        Description: Issues Redraw command
```

```
 */

static void Redraw(widget, client_data, call_data)
Widget        widget;
XtPointer client_data, call_data;
{
  write_com( 7 );
}

/*        Function Name: Redraw
 *        Description: Redraw the net.
 */

static void StatsRates(w, text_ptr, call_data)
Widget w;
XtPointer text_ptr, call_data;
{
    write_com( 8 );
}

/*        Function Name: StatsTimes
 *        Description: This function toggles between rates and times fired.
 */

static void StatsTimes(w, text_ptr, call_data)
Widget w;
XtPointer text_ptr, call_data;
{
    write_com( 9 );
}


/*        Function Name: ResetStats
 *        Description: Reset firing counter and timer.
 */

static void ResetStats(w, text_ptr, call_data)
Widget w;
XtPointer text_ptr, call_data;
{
    write_com( 10 );
}

/*        Function Name: Dump
 *        Description: This causes a dump of the data of net
 */

static void Dump(w, text_ptr, call_data)
Widget w;
XtPointer text_ptr, call_data;
{
    write_com( 11 );
}
```

A.6 *XDisplay* — Displaying the Petri Net Under Execution

A.6.1 *mXd* — Make File for *XDisplay*

```
##
```

```
##
##
EXEC=        XCommand
##
##        -g = Debugging info
##        -O = Optimize
CFLAGS=        -g
##
##        Libraries
##        X11        X11 graphics library
##
LIBS= -lXaw -lXmu -lXt -lX11
#
#
OBJECTS=        XCommand.c        \
                setblock.c        \
                client.c        \
                syserr.c
##
##        Compile & link
##
$(EXEC):        $(OBJECTS)
                cc $(CFLAGS) -o $(EXEC) $(OBJECTS) $(LIBS)
##
##
##        End of make file
##
```

A.6.2 *Xdraw.h* — Data Structure Definitions

```
#define IN 0                        /* -- These define arc types  */
#define OUT 1
#define INH 2


/*#define DEBUG1 1                    -- Print input from token players */
/*#define DEBUG2 2                      -- Print firing sequence & marking */
/*#define DEBUG3 3                      -- Enable testing function  */

#define XMIN 0                  /* -- These four parameters define X window */
#define XMAX 550                /*    size.   */
#define YMIN 0
#define YMAX 750
#define OffSize 1
#define UnitSize 18
#define GridSize UnitSize*3
#define GridOff 100
#define        Dia        UnitSize
#define Dia2    13
#define ARROWL 10

#include "size_limits.h"

/*-------------------------------------------------------------------
 * mag(x, off) Converts the Grid Coordinates to the real coordinates
 * on the screen. x is the grid number, off is the offset on that
 * grid.
 *-------------------------------------------------------------------*/
#define mag(x, off)        ((x * GridSize + off * OffSize + GridOff))
```

```
struct PL {                          /* -- This structure is used to store info  */
  char *name;                        /*    re. places at each host  */
  float x_pos, y_pos;
  float xt_pos, yt_pos;
  int tokens; };

struct TR {                          /* -- Info abt. transitions for each host  */
  char *name;
  float x_pos, y_pos;
  float xt_pos, yt_pos;
  float xr_pos, yr_pos;
  long times_fired;                  /* -- Counter for # of times trans. fired  */
  long old_times_fired;              /* -- Counter for # of times trans. fired  */
  float rate;                        /* -- Keeps last written avg. firing rate  */
  int highligthed;
  int type;
  int rot; };

struct obj {                         /* -- name and host assignment of all places*/
  char *name;
  int host_no; };

struct hosts {   /* -- Host info  */
  char HOSTNAME[ MAXLEN ];
  int LOCAL_PLACES;
  int NO_TRANSITIONS;
  int socket;
  struct PL P[ MAXTRANS ];
  struct TR T[ MAXTRANS ];
};

struct gmv {   /* -- Info abt. global marking vector  */
  char *name;
  int tokens;
  float x_pos, y_pos;
  float xt_pos, yt_pos;
};
```

A.6.3 *XDisplay.c* — main() and Routines for Reading Net Structure

```
#include "size_limits.h"
#include "portnums.h"
#include <stdio.h>
#include "defs.h"
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <fcntl.h>
#include <errno.h>
#include <signal.h>
#include <math.h>
#include <X11/Xlib.h>
#include "Xdraw.h"

extern Display *theDisplay;
extern XEvent theEvent;

Window theWindow, openWindow();
```

```
GC theGC;

struct obj PLACES[ MAXTRANS ];   /* -- name and host assignment of all places*/
struct obj TRANS[ MAXTRANS ];    /* -- name & host of all trans.  */
struct hosts H[ MAXHOST ];       /* -- All net infor stored here  */
struct gmv GM[ MAXTRANS ];       /* -- Info abt. global marking vector  */

int GLOBAL_PLACES;
int NO_HOSTS;
int NO_PLACES;
int NO_TRANS;

BOOLEAN display_stats = FALSE;
BOOLEAN avg_rate = TRUE;

int layer = 0x7FF;               /* -- Stores which layer is to be displayed
                                       0x400 - global places only
                                       0x7FF - everything
                                       0x7FF XOR (host #) - toggle host on/off
                                    */
BOOLEAN run = FALSE;
BOOLEAN single_step = FALSE;

/* ------------------------------------------------------------------------
**
**        input_interrupt_handler()
**
**        When an I/O interrupt is issued, this routine is called.
**      The procedure sets a flag.
*/

void
input_interrupt_handler()
{
  printf("-"); fflush( stdout );
}

void alarm_handler()
{
  printf("+"); fflush( stdout );
}

/* ------------------------------------------------------------------------
**
**        init_hosts()
**
**        Initializes the H[] datastructure by reading the include file
**      generated by assign.c
*/

init_hosts()
{
  FILE *in_f, *fopen();
  char buf[1024];
  char tmp[1024];
  int i;

  if ( ( in_f = fopen("XDisplay.i", "r" ) ) == -1 )
    syserr( "open XDisplay.i" );

  /* -- Read # of hosts & global places  */
```

```
fgets( buf, sizeof( buf ), in_f );
sscanf( buf, "%d %d %d %d", &NO_HOSTS, &GLOBAL_PLACES, &NO_PLACES,
        &NO_TRANS );

/* -- Read each host name  */
for ( i = 0; i < NO_HOSTS; i++ )
  {
    fgets( buf, sizeof( buf ), in_f );
    sscanf( buf, "%s %d %d", &H[ i ].HOSTNAME, &H[ i ].LOCAL_PLACES,
            &H[ i ].NO_TRANSITIONS );
  }

/* -- Read place info  */
for ( i = 0; i < NO_PLACES; i++ )
  {
    fgets( buf, sizeof( buf ), in_f );
    sscanf( buf, "%s %d", tmp, &PLACES[ i ].host_no );
    strcpy(PLACES[ i ].name = (char *) malloc(strlen(tmp) + 1), tmp);
  }

/* -- Read transition info  */
for ( i = 0; i < NO_TRANS; i++ )
  {
    fgets( buf, sizeof( buf ), in_f );
    sscanf( buf, "%s %d", tmp, &TRANS[ i ].host_no );
    strcpy(TRANS[ i ].name = (char *) malloc(strlen(tmp) + 1), tmp);
  }

/* -- Read inhibitor arch stuff */
/* -- NEEDS TO BE IMPLEMENTED  */

close ( in_f );

}

/* ------------------------------------------------------------------------
**
**        gethost(name)
**
**        Searches the PLACES and TRANS arrays to find the host the object is
**      assigned to.
**
*/

int gethost( name )
char *name[];
{
  int i;

  for ( i = 0; i < NO_PLACES; i++)
    if (!strcmp(name, PLACES[i].name))
      return(PLACES[i].host_no);

  for ( i = 0; i < NO_TRANS; i++)
    if (!strcmp(name, TRANS[i].name))
      return(TRANS[i].host_no);

  return(-1);
}

/* ------------------------------------------------------------------------
```

```
**
**        getnum(host, obj_name)
**
**        Returns the local object number given a host and the object name.
**      Searches the PLACES and TRANS arrays.
**
*/

int getnum(host, name)
int host;
char *name[];
{
  int i, plno;

  plno = 0;
  for ( i = 0; i < NO_PLACES; i++)
    {
      if (!strcmp(name, PLACES[i].name) && PLACES[i].host_no == host)
        return(plno);
      if(PLACES[i].host_no == host)
        plno++;
    }
  plno = 0;
  for ( i = 0; i < NO_TRANS; i++)
    {
      if (!strcmp(name, TRANS[i].name) && TRANS[i].host_no == host)
        return(plno);
      if(TRANS[i].host_no == host)
        plno++;
    }
  return(-1);
}


/* ------------------------------------------------------------------------
**
**        check_layer( host )
**
**        Checks if the given host number is currently enabled for
**      displaying.
**
*/

int check_layer( host )
int host;
{
  /* -- Convert to correct bit placement  */
  if ( host == -1 )
    host = 0x400;
  else
    host = (1<<host);

  if ( host & layer )
    return 1;
  else
    return 0;
}

/* ------------------------------------------------------------------------
**
**        get_geo_info()
```

```
**
**        Reads the test.net file - the original GSPN file to get the
**      geographical info.  This procedure is an adaption of "net2n.c",
**      written by  Andreas Nowatzyk (agn@unh.cs.cmu.edu).
**
**      If refresh is true, the markings in H are used, else, the markings in
**      the .net file.
*/

void draw_geo_info(refresh)
int refresh;
{
  FILE *in_f;
  register int i, j;
  int n_MK, n_PL, n_RT, n_TR, n_GR;
  int tokens, h, obj_no, type, no_dep, rot, mult, from, to, geo, out_dep,
      inh_arc;
  int gmv_counter;
  float x_pos, y_pos, xt_pos, yt_pos, xr_pos, yr_pos, xi_pos, yi_pos, rate;
  char            buf[1024], tmp[1024];

  for ( i = 0; i < NO_HOSTS; i++ )
    {
      sprintf(tmp, "Host %d : %s", i, H[ i ].HOSTNAME );
      write_tag(tmp, 10, 60 + 20*i );
    }

  if ( !run )
    write_tag("HALT", 10, 10);
  else
    write_tag("RUN", 10, 10);

  if ( single_step )
    write_tag("SINGLE_STEP", 10, 30);

  if ( ( in_f = fopen("test.net", "r" ) ) == -1 )
    syserr( "open test.net" );

  while (fgets( buf, sizeof( buf ), in_f ))        /* skip preamble */
    if (buf[0] == '|' && buf[1] == '\n')
      break;

  if (!fgets( buf, sizeof( buf ), in_f ) || 5 != sscanf(buf, "%*s%d%d%d%d%d",
&n_MK, &n_PL, &n_RT, &n_TR, &n_GR) ||
       n_MK < 0 || n_PL < 1 || n_RT < 0 || n_TR < 1 || n_GR < 0)
       syserr("Bogus parameters");

  for (i = 0; i < n_MK; i++)          /* Skip mark-parameters */
    if (!fgets( buf, sizeof( buf ), in_f )) syserr("Premature end of file");

  gmv_counter = 0;
  for (i = 0; i < n_PL; i++)          /* read places */
    {
      if (!fgets( buf, sizeof( buf ), in_f )) syserr("Premature end of file");
      if (6 != sscanf(buf, "%s%d%f%f%f%f", tmp, &tokens, &x_pos, &y_pos,
                      &xt_pos, &yt_pos))
        syserr("Place def problem");

      h = gethost(tmp);
      if ( h >= 0 )
        {
```

```
        obj_no = getnum(h, tmp);
        strcpy(H[h].P[obj_no].name = (char *) malloc(strlen(tmp) + 1), tmp);
        H[h].P[obj_no].tokens = ( refresh ? H[h].P[obj_no].tokens : tokens );
        H[h].P[obj_no].x_pos = x_pos;
        H[h].P[obj_no].y_pos = y_pos;
        H[h].P[obj_no].xt_pos = xt_pos;
        H[h].P[obj_no].yt_pos = yt_pos;
      }
    else
      {
        obj_no = gmv_counter++;
        strcpy(GM[obj_no].name = (char *) malloc(strlen(tmp) + 1), tmp);
        GM[obj_no].tokens = ( refresh ? GM[obj_no].tokens : tokens );
        GM[obj_no].x_pos = x_pos;
        GM[obj_no].y_pos = y_pos;
        GM[obj_no].xt_pos = xt_pos;
        GM[obj_no].yt_pos = yt_pos;
      }
    if ( check_layer( h ) )
      {
        draw_place(h, obj_no);
        draw_tokens(h, obj_no, tokens, refresh);
      }
  }

for (i = 0; i < n_RT; i++)                    /* Skip rate-parameters */
  if (!fgets( buf, sizeof( buf ), in_f )) syserr("Premature end of file");

for (i = 0; i < n_GR; i++)                          /* skip groups */
  if (!fgets( buf, sizeof( buf ), in_f )) syserr("Premature end of file");

for (i = 0; i < n_TR; i++)                    /* read transitions */
  {
    if (!fgets( buf, sizeof( buf ), in_f )) syserr("Premature end of file");
    if (11 != sscanf(buf, "%s%f%*d%d%d%d%f%f%f%f%f%f%f", tmp, &rate,
                     &type, &no_dep, &rot, &x_pos, &y_pos,
                     &xt_pos, &yt_pos, &xr_pos, &yr_pos))
      syserr("Transition def problem");

    h = gethost(tmp);
    obj_no = getnum(h, tmp);
    strcpy(H[h].T[obj_no].name = (char *) malloc(strlen(tmp) + 1), tmp);
    H[h].T[obj_no].type = type;
    H[h].T[obj_no].rot = rot;
    H[h].T[obj_no].x_pos = x_pos;
    H[h].T[obj_no].y_pos = y_pos;
    H[h].T[obj_no].xt_pos = xt_pos;
    H[h].T[obj_no].yt_pos = yt_pos;
    H[h].T[obj_no].xr_pos = xr_pos;
    H[h].T[obj_no].yr_pos = yr_pos;

    if ( check_layer( h ) ) draw_trans(h, obj_no);

    for (j = no_dep; j--;)                          /* read inputs */
      {
        if (!fgets( buf, sizeof( buf ), in_f ))
  syserr("Premature end of file");
        if (3 != sscanf(buf, "%d%d%d", &mult, &from, &geo))
          syserr("Transition input def problem");
        if(mult < 0) mult = -mult;
        while (mult--)
```

```
        {
          if (!geo)
            {
              if ( check_layer( h ) )
                draw_simplearc(from-1, h, obj_no, mult, IN);
            }
          else
            {
              geo--;
              fgets( buf, sizeof( buf ), in_f );
              sscanf(buf, "%f%f", &x_pos, &y_pos);
              if ( check_layer( h ) )
                start_arc(h, obj_no, x_pos, y_pos, IN);
              while (geo--)
                {
                  fgets( buf, sizeof( buf ), in_f );
                  sscanf(buf, "%f%f",&x1_pos, &y1_pos);
                  if ( check_layer( h ) )
                    draw_arc(x_pos, y_pos, x1_pos, y1_pos);
                  x_pos = x1_pos;
                  y_pos = y1_pos;
                }
              if ( check_layer( h ) )
                end_arc(from-1, x_pos, y_pos, IN); }}}

  if (!fgets( buf, sizeof( buf ), in_f )) syserr("Premature end of file");
  if (1 != sscanf (buf, "%d", &out_dep))
    syserr("TR output problem");
  for (j = out_dep; j--;)                          /* read outputs */
    {
      if (!fgets( buf, sizeof( buf ), in_f ))
syserr("Premature end of file");
      if (3 != sscanf(buf, "%d%d%d", &mult, &to, &geo))
        syserr("Transition output def problem");
      if(mult < 0) mult = -mult;
      while (mult--)
        {
          if (!geo)
            {
              if ( check_layer( h ) )
                draw_simplearc(to-1, h, obj_no, mult, OUT);
            }
          else
            {
              geo--;
              fgets( buf, sizeof( buf ), in_f );
              sscanf(buf, "%f%f", &x_pos, &y_pos);
              if ( check_layer( h ) )
                start_arc(h, obj_no, x_pos, y_pos, OUT);
              while (geo--)
                {
                  fgets( buf, sizeof( buf ), in_f );
                  sscanf(buf, "%f%f", &x1_pos, &y1_pos);
                  if ( check_layer( h ) )
                    draw_arc(x_pos, y_pos, x1_pos, y1_pos, OUT);
                  x_pos = x1_pos;
                  y_pos = y1_pos;
                }
              if ( check_layer( h ) )
                end_arc(to-1, x_pos, y_pos, OUT); } } }
    if (!fgets( buf, sizeof( buf ), in_f )) syserr("Premature end of file");
```

```
    if (1 != sscanf (buf, "%d", &inh_arc))
      syserr("TR output problem");
    for (j = inh_arc; j--;)                    /* read inhibitor arcs  */
      {
        if (!fgets( buf, sizeof( buf ), in_f ))
   syserr("Premature end of file");
        if (3 != sscanf(buf, "%d%d%d", &mult, &to, &geo))
          syserr("Transition inhibitor def problem");
        if(mult < 0) mult = -mult;
        while (mult--)
          {
            if (!geo)
              {
                if ( check_layer( h ) )
                  draw_simplearc(to-1, h, obj_no, mult, INH);
              }
            else
              {
                geo--;
                fgets( buf, sizeof( buf ), in_f );
                sscanf(buf, "%f%f", &x_pos, &y_pos);
                if ( check_layer( h ) )
                  start_arc(h, obj_no, x_pos, y_pos, INH);
                while (geo--)
                  {
                    fgets( buf, sizeof( buf ), in_f );
                    sscanf(buf, "%f%f", &x1_pos, &y1_pos);
                    if ( check_layer( h ) )
                      draw_arc(x_pos, y_pos, x1_pos, y1_pos, INH);
                    x_pos = x1_pos;
                    y_pos = y1_pos;
                  }
                if ( check_layer( h ) )
                  end_arc(to-1, x_pos, y_pos, INH); } } } }
}

/* ------------------------------------------------------------------------
**
**        highlight( host, obj_num, mode )
**
**        Highlights a transition selected by (host,obj_num).  Mode selects
**      on or off.
*/

void highlight(host, obj_num )
int host, obj_num;
{
  int x, y;

  x = mag(H[host].T[obj_num].x_pos, 0);
  y = mag(H[host].T[obj_num].y_pos, 0);
  XFillRectangle(theDisplay, theWindow, theGC,
                 x - Dia/2, y - Dia/2,
                 Dia, Dia);
}

/* ------------------------------------------------------------------------
**
**        firing( buf )
**
**        Highlights all fired transitions passed in buf.
```

```
*/

void firing( buf )
short buf[];
{

  int ctr;      /* -- Counter for reading the firing/marking buffer*/

#ifdef DEBUG2
  printf( "Firing sequence : " );
  for (ctr = 0 ; ctr < H[ buf[ 0 ] ].NO_TRANSITIONS; ctr++ )
    if ( buf[ ctr + 1 ] > H[ buf[ 0 ] ].T[ ctr ].old_times_fired )
      printf( "%d ", buf[ ctr + 1 ] );
  printf( "\n" );
#endif

  /* -- Set interrupt timer to flash the transitions for  ...msec */
  set_intr(0, 350000);

  /* -- Draw block over fired transitions  */
  XSetFunction(theDisplay, theGC, GXinvert);
  for ( ctr = 0; ctr < H[ buf[ 0 ] ].NO_TRANSITIONS; ctr++)
    if ( buf[ ctr + 1 ] > H[ buf[ 0 ] ].T[ ctr ].old_times_fired )
      if ( check_layer( buf[ 0 ] ) )
        highlight( buf[ 0 ], ctr );
  XFlush(theDisplay);

  /* -- Pause long enough to see the block  */
  pause();

  /* -- Invert block again to get back transition  */
  for ( ctr = 0; ctr < H[ buf[ 0 ] ].NO_TRANSITIONS; ctr++)
    if ( buf[ ctr + 1 ] > H[ buf[ 0 ] ].T[ ctr ].old_times_fired )
      if ( check_layer( buf[ 0 ] ) )
        highlight( buf[ 0 ], ctr );
  XFlush(theDisplay);
  XSetFunction(theDisplay, theGC, GXcopy);
}

/* ------------------------------------------------------------------------
**
**        local_marking( buf )
**
**        Updates the token count in each place for one host.  buf contains
**      the local marking vector for one host.
**
*/

void local_marking( buf )
short buf[];
{
  int ctr;      /* -- Counter for reading the firing/marking buffer*/
  int i;

#ifdef DEBUG2
  printf( "Local Marking : ");
  for ( ctr = 1 + H[ buf[ 0 ] ].NO_TRANSITIONS;
        ctr < 1 + H[ buf[ 0 ] ].NO_TRANSITIONS + H[ buf[ 0 ] ].LOCAL_PLACES;
        ctr++ )
    printf( "%d ", buf[ ctr ] );
  printf( "\n" );
```

```
#endif

  for ( i = 0, ctr = 1 + H[ buf[ 0 ] ].NO_TRANSITIONS;
        ctr < 1 + H[ buf[ 0 ] ].NO_TRANSITIONS + H[ buf[ 0 ] ].LOCAL_PLACES;
        i++, ctr++ )
    {
      if ( check_layer( buf[ 0 ] ) )
        {
          erase_tokens( buf[ 0 ], i);
          draw_tokens( buf[ 0 ], i, buf[ ctr ], i);
        }
      H[ buf[ 0 ] ].P[ i ].tokens = buf[ ctr ];
    }
}

/* ------------------------------------------------------------------------
**
**        global_marking( buf )
**
**        Updates the token count in each global place.
**
*/

void global_marking( buf )
short buf[];
{
  int ctr;      /* -- Counter for reading the firing/marking buffer*/
  int i;

#ifdef DEBUG2
  printf( "Global Marking : ");
  for ( ctr = 1 + H[ buf[ 0 ] ].NO_TRANSITIONS + H[ buf[ 0 ] ].LOCAL_PLACES;
        ctr < 1 + H[ buf[ 0 ] ].NO_TRANSITIONS + H[ buf[ 0 ] ].LOCAL_PLACES
              + GLOBAL_PLACES; ctr++ )
    printf( "%d ", buf[ ctr ] );
  printf( "\n" );
#endif

  for ( i = 0, ctr = 1 + H[ buf[ 0 ] ].NO_TRANSITIONS +
        H[ buf[ 0 ] ].LOCAL_PLACES;
        ctr < 1 + H[ buf[ 0 ] ].NO_TRANSITIONS + H[ buf[ 0 ] ].LOCAL_PLACES
              + GLOBAL_PLACES; i++, ctr++ )
    {
      if ( check_layer( -1 ) )
        {
          erase_tokens( -1, i);
          draw_tokens( -1, i , buf[ ctr ], 1 );
        }
      GM[ i ].tokens = buf[ ctr ];
    }
}

/* ------------------------------------------------------------------------
**
**        read_player_sockets()
**
**        Reads each socket and updates the window according to what is
**        received.
*/

void read_player_sockets()
```

```
{
  short in_buffer[ 2 + 2*MAXTRANS ];
  int i, j, host_no, nread;
  BOOLEAN empty;

  empty = FALSE;
  while (!empty)
    {
      empty = TRUE;
      /* -- Read the sockets from the token players  */
      for ( j = 0; j < NO_HOSTS; j++ )
        {
          if ( nread = read( H[ j ].socket, in_buffer, sizeof( in_buffer ) )
            > 0 )
            {
              host_no = in_buffer[ 0 ];
              empty = FALSE;

#ifdef DEBUG1
              printf( "\nHost %d : ", host_no );
              for( i = 0; i < 1 + H[ in_buffer[ 0 ] ].NO_TRANSITIONS +
                   H[in_buffer[0]].LOCAL_PLACES + GLOBAL_PLACES; i++ )
                printf("%d ",in_buffer[ i ]);
              printf("\n");
#endif
              /* -- marking vector received  */
              local_marking ( in_buffer );
              global_marking( in_buffer );
              /* -- Transition firing sequence received  */
              firing( in_buffer );
              update_stats( in_buffer ); } } }
}


/* ------------------------------------------------------------------------
**
**        read_command_socket()
**
**        Reads the socket from XCommand.
*/

void read_command_socket()
{
  short in_buffer[ 1 + 3*MAXTRANS ];
  int i, j, nread;

  /* -- Read socket from command window (Note, XCommand writes long ints
        while XDisplay reads short ints  */
  nread = read( H[ NO_HOSTS ].socket, in_buffer, sizeof( in_buffer ) );
  if ( nread > 0 )  /* -- have read a command */
    {
      if (in_buffer[ 1 ] != NO_HOSTS )
        printf( "Socket for XCommand written to by host %d\n",
                in_buffer[1] );
      else
        {
          printf( "Command received : %d\n", in_buffer[ 3 ] );
          switch ( in_buffer[ 3 ] )
            {
            case 1 :                              /* -- RUN  */
              if ( !run )
```

```
    {
      init_elapsed_time();
      run = TRUE;
      st_erase();
      write_tag("HALT", 10, 10);
      st_normal();
      write_tag("RUN", 10, 10);
    }
  else
    write_tag("RUN", 10, 10);
  break;
case 2 :                              /* -- HALT  */
  if ( run )
    {
      run = FALSE;
      st_erase();
      write_tag("RUN", 10, 10);
      st_normal();
      write_tag("HALT", 10, 10);
    }
  else
    write_tag("HALT", 10, 10);
  break;
case 21 :                             /* -- CONTINUE  */
  run = TRUE;
  st_erase();
  write_tag("HALT", 10, 10);
  st_normal();
  write_tag("RUN", 10, 10);
  break;
case 3 :                              /* -- RESET  */
  clear_net();
  draw_geo_info( 0 );
  break;
case 4 :                              /* -- QUIT  */
  printf( "XDisplay terminating \n" );
  quitX();
  for ( i = 0; i < NO_HOSTS + 1; i++ )
    close( H[ i ].socket );
  exit( 0 );
case 5 :                              /* -- Single Step */
  if ( single_step )
    {
      single_step = FALSE;
      st_erase();
      write_tag("SINGLE_STEP", 10, 30);
      st_normal();
    }
  else
    {
      single_step = TRUE;
      write_tag("SINGLE_STEP", 10, 30);
    }
  break;
case 6 :                              /* -- FIRE  */
  break;
case 7 :                              /* -- REDRAW  */
  clear_net();
  draw_geo_info( 1 );
  break;
case 8 :                              /* -- Stats ON/OFF  */
```

```
            if ( display_stats )
              {
                display_stats = FALSE;
                clear_stats();
              }
            else
              {
                display_stats = TRUE;
                update_all_stats();
              }
            break;
          case 9 :                          /* -- Select rates or times  */
            clear_stats();
            if ( avg_rate )
              avg_rate = FALSE;
            else
              avg_rate = TRUE;
            update_all_stats();
            break;
          case 10 :
            reset_stats();
            break;
          case 11 :                         /* -- dump net data to screen  */
            break; } } }
}

/* ------------------------------------------------------------------------
**
**        main()
**
**        Waits for a firing sequence from and a marking vector from a host and
**      displays the info in the Xwindow.
**
*/

main()
{
  int i, count1 = 0;

  init_hosts();         /* -- Read info abt. Petri Net  */
  initX();              /* -- Initialize X window  */
  theWindow = openWindow(XMIN,YMIN, XMAX,YMAX,0, &theGC);
  initXEvents( theWindow );   /* -- Initialize X event queue  */
  usleep(400000);             /* -- Wait long enough for window to be set up */
  draw_geo_info( 0 );         /* -- Draw net in window  */
  XFlush(theDisplay);

  /* -- Set up connections  */
  for ( i = 0; i < NO_HOSTS + 1; i++ )
    {
      H[ i ].socket = serv_setup_intr( H[ 0 ].HOSTNAME, DISPLAYPORT,
                                       "XDisplay waiting for next connection");
      setblock( H[ i ].socket, FALSE );
    }

  init_elapsed_time();
  signal(SIGALRM, alarm_handler);  /* -- Set signal for interrupt timer  */
  set_intr(1,0);
  signal (SIGIO, input_interrupt_handler);
  printf( "XDisplay listening ...\n" );
```

```
  while( TRUE )                    /* -- Do until quit  */
    {
      switch ( count1++ )
        {
        case 0 : printf( "\b-" ); break;
        case 1 : printf( "\b/" ); break;
        case 3 : printf( "\b|" ); break;
        case 4 : printf( "\b\\" ); break;
        case 5 : count1 = 0; break;
        }
      fflush( stdout );
      pause();                     /* -- Wait for io  */

      read_command_socket();       /* -- Check for all events and update window*/
      read_player_sockets();
      XEventHandler();

      set_intr(1,0);               /* -- Set timer interrupt for 1 sec. and  */
      display_elapsed_time();      /*    display elapsed time  */
    }
}
```

A.6.4 *Xdraw.c* — Drawing Routines for net

```
#include "size_limits.h"
#include <stdio.h>
#include "defs.h"
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <fcntl.h>
#include <errno.h>
#include <signal.h>
#include <math.h>
#include <X11/Xlib.h>
#include "Xdraw.h"

extern Display *theDisplay;
extern XEvent theEvent;
extern Window theWindow, openWindow();
extern GC theGC;
extern struct obj PLACES[];
extern struct obj TRANS[];
extern struct hosts H[];
extern struct gmv GM[];
extern int GLOBAL_PLACES;
extern int NO_HOSTS;
extern int NO_PLACES;
extern int NO_TRANS;
extern XSetWindowAttributes        theWindowAttributes;

/*-------------------------------------------------------------------------
 * st_erase()
 *        (set erase mode). Sets the GC to clear mode.
 *-------------------------------------------------------------------------*/
st_erase()
{
  XSetForeground(theDisplay, theGC, theWindowAttributes.background_pixel);
```

```
}

/*-----------------------------------------------------------------------
 * st_normal()
 *        (set normal mode), undoes the effect of st_erase, if used earlier.
 *        using st_normal multiple times should not cause any harm.
 *-----------------------------------------------------------------------*/
st_normal()
{
  XSetForeground(theDisplay, theGC, theWindowAttributes.border_pixel);
}

clear_net()        /* Clears the Petri-Net Window */
{
       XClearWindow(theDisplay, theWindow);
}

/* ----------------------------------------------------------------------
**
**        write_tag(tag, x, y)
**
**        Writes the given tag at the given position.
*/

void write_tag(tag, x, y)
char *tag;
int x, y;
{
  x += Dia/4;
  y += Dia/4;
  XDrawString(theDisplay, theWindow, theGC,
              x, y,
              tag, strlen(tag));
#ifdef DEBUG2
  printf("%s\n",tag);
#endif
}

/* ----------------------------------------------------------------------
**
**        draw_place(host, obj_no)
**
**        Looks up the x and y coordinates of the given object and draws it.
*/

void draw_place(host, obj_no)
int host, obj_no;
{
  int        x, y, xt, yt;

  if ( host >= 0 )
    {
      x = mag(H[host].P[obj_no].x_pos, 0);
      y = mag(H[host].P[obj_no].y_pos, 0);
      xt = mag(H[host].P[obj_no].xt_pos, 0);
      yt = mag(H[host].P[obj_no].yt_pos, 0);
      write_tag(H[host].P[obj_no].name, xt, yt);
    }
  else
    {
      xt = mag(GM[obj_no].xt_pos, 0);
```

```
      yt = mag(GM[obj_no].yt_pos, 0);
      x = mag(GM[obj_no].x_pos, 0);
      y = mag(GM[obj_no].y_pos, 0);
      write_tag(GM[obj_no].name, xt, yt);
    }
  XDrawArc( theDisplay, theWindow, theGC,
              x - Dia/2 , y - Dia/2,
              Dia, Dia,
              0, 360 *64);
}

/* ------------------------------------------------------------------------
**
**        draw_trans(host, obj_no)
**
**        Looks up the x and y coordinates of the given object and draws it.
*/

void draw_trans(h, no)
int h, no;
{
  int        x, y;

  x = mag(H[h].T[no].x_pos, 0);
  y = mag(H[h].T[no].y_pos, 0);
  switch( H[h].T[no].rot )
    {
    case 0 :      /* -- Horizonthal  */
      XDrawLine(theDisplay, theWindow, theGC,
                x - Dia/2, y,
                x + Dia/2, y);
      break;
    case 1 :      /* -- Vertical  */
      XDrawLine(theDisplay, theWindow, theGC,
                x, y + Dia/2,
                x, y - Dia/2);
      break;
    case 2 :    /* -- 45 degree tilted  */
      XDrawLine(theDisplay, theWindow, theGC,
                x - Dia2/2, y - Dia2/2,
                x + Dia2/2, y + Dia2/2);
      break;
    default : ;
    }
  x = mag(H[h].T[no].xt_pos, 0);
  y = mag(H[h].T[no].yt_pos, 0);
  write_tag(H[h].T[no].name, x, y);
}

#define dMark        5
#define dArcAng        30

/* ------------------------------------------------------------------------
**
**        drawtoken(x, y)
**
**        Draws one token at the specified x,y address ( terminal coordinates )
*/

drawtoken(x, y)
int x, y;
```

```
{
  XFillArc(theDisplay, theWindow, theGC,
                x, y,
                dMark, dMark,
                0, 360*64);
}

/* ------------------------------------------------------------------------
**
**        draw_token(host, obj_no, tokens)
**
**        Draws the tokens if < 5 and writes # for 5 or more.
*/

void draw_tokens(host, obj_no, tokens, new)
int host, obj_no, tokens, new;
{
  int        x, y, i;
  float x_pos, y_pos;
  char tmp[10];

  if ( host >= 0 )
    {
      x = mag(H[host].P[obj_no].x_pos, -2);
      y = mag(H[host].P[obj_no].y_pos, -2);
      if ( !new )
        tokens = H[host].P[obj_no].tokens;
    }
  else
    {
      x = mag(GM[obj_no].x_pos, -2);
      y = mag(GM[obj_no].y_pos, -2);
      if ( !new )
        tokens = GM[obj_no].tokens;
    }

  switch (tokens)
    {
    case 0 : break;
    case 1 :
      drawtoken(x, y); break;
    case 2 :
      drawtoken(x - Dia/4, y);
      drawtoken(x + Dia/4, y); break;
    case 3 :
      drawtoken(x, y - Dia/4);
      drawtoken(x - Dia/4, y + Dia/4);
      drawtoken(x + Dia/4, y + Dia/4); break;
    case 4 :
      drawtoken(x - Dia/4, y - Dia/4);
      drawtoken(x + Dia/4, y - Dia/4);
      drawtoken(x - Dia/4, y + Dia/4);
      drawtoken(x + Dia/4, y + Dia/4); break;
    default :
      sprintf(tmp, "%d", tokens);
      XDrawString( theDisplay, theWindow, theGC, x, y+6, tmp, strlen(tmp));
    }
}

/* ------------------------------------------------------------------------
```

```
**
**      erase_tokens(host, obj_no)
**
**        Removes the tokens by changing foreground to background and
**        drawing the token.
*/

erase_tokens(host, obj_no)
int host, obj_no;
{
        st_erase();
        draw_tokens(host, obj_no, 0, 0);   /* -- last 0 specifies draw old
                                              tokens  */
        st_normal();
}


/* ------------------------------------------------------------------------
**
**        arrowhead(x1,y1, x, y, theta)
**
**        Draw an arrowhead.
*/

void arrowhead( x1, y1, x, y, theta )
int x1, y1, x, y;
float theta;
{
  int x2, y2, x3, y3;

  /* -- Correct 'x' alignment */
  if ( x > 0 )
    {
      x2 = x1 - ARROWL*cos(theta + M_PI/10);
      x3 = x1 - ARROWL*cos(theta - M_PI/10);
    }
  else
    {
      x2 = x1 + ARROWL*cos(theta + M_PI/10);
      x3 = x1 + ARROWL*cos(theta - M_PI/10);
    }
  /* -- Correct 'y' alignment */
  if ( y > 0 )
    {
      if ( x < 0 )
        {
          y2 = y1 + ARROWL*sin(theta + M_PI/10);
          y3 = y1 + ARROWL*sin(theta - M_PI/10);
        }
      else
        {
          y2 = y1 - ARROWL*sin(theta + M_PI/10);
          y3 = y1 - ARROWL*sin(theta - M_PI/10);
        }
    }
   else  /* -- y > 0  */
    {
      if ( x > 0 )
        {
          y2 = y1 - ARROWL*sin(theta + M_PI/10);
          y3 = y1 - ARROWL*sin(theta - M_PI/10);
```

```
      }
    else
      {
        y2 = y1 + ARROWL*sin(theta + M_PI/10);
        y3 = y1 + ARROWL*sin(theta - M_PI/10);
      }
  }
 XDrawLine(theDisplay, theWindow, theGC,
           x1, y1,
           x2, y2);
 XDrawLine(theDisplay, theWindow, theGC,
           x1, y1,
           x3, y3);
}

/* ------------------------------------------------------------------------
**
**        draw_simplearc(from, host, obj_no, mult, mode)
**
**      Draws an arch between the place specified by from by looking up PLACES
**      to transition (host, obj_no) with multiplicity mult.  If the mode is
**      IN, the arrowhead is at the transition, else it is at the
**      transition.
*/

void draw_simplearc(from, host, obj_no, mult, mode)
int from, host, obj_no, mult, mode;
{
  int i, h, no;
  char tmp[MAXLEN];
  int x1, y1, x2, y2, x, y;
  float a, theta;

  strcpy(tmp, PLACES[from].name);   /* -- get name of place from */
  h = gethost(tmp);
  no = getnum(h, tmp);

  if ( h >= 0 )       /* -- Place belongs to a token player  */
    {
      x1 = mag(H[h].P[no].x_pos, 0);
      y1 = mag(H[h].P[no].y_pos, 0);
    }
  else                /* -- Place belongs to the global marking vector  */
    {
      x1 = mag(GM[no].x_pos, 0);
      y1 = mag(GM[no].y_pos, 0);
    }

  x2 = mag(H[host].T[obj_no].x_pos, 0);
  y2 = mag(H[host].T[obj_no].y_pos, 0);

  x = x1 - x2;        /* -- Calculate angle of attack of arc  */
  y = y1 - y2;
  if (!x)
    theta = M_PI/2;
  else
    theta = atan((float)y/x);

  if ( x > 0 )        /* -- Correct 'x' alignment of entry point to place  */
    x1 -= (Dia/2*cos(theta));
  else
```

```
    x1 += (Dia/2*cos(theta));

  /* -- Correct 'y' alignment of entry point to place  */
  if ( y > 0 )
    {
      if ( x < 0 )
y1 += (Dia/2*sin(theta));
      else
y1 -= (Dia/2*sin(theta));
    }
  else  /* -- y > 0  */
    {
      if ( x > 0 )
y1 -= (Dia/2*sin(theta));
      else
y1 += (Dia/2*sin(theta));
    }

  switch (mode)
    {
    case IN :
      arrowhead(x2, y2, -x, -y, theta);
      XDrawLine(theDisplay, theWindow, theGC,
                x1, y1,
                x2, y2); break;
    case OUT :
      arrowhead(x1, y1, x, y, theta);
      XDrawLine(theDisplay, theWindow, theGC,
                x1, y1,
                x2, y2); break;
    case INH :
      XDrawArc(theDisplay, theWindow, theGC,
               x2, y2,
               dMark, dMark,
               0, 360*64);
      XDrawLine(theDisplay, theWindow, theGC,
                x1, y1,
                x2, y2);
    }
} /* -- End of        draw_simplearc(from, host, obj_no, mult, mode)  */

/* ------------------------------------------------------------------------
**
**        start_arc(host, obj_no, x_pos, y_pos, mode)
**
**        Draws the start of an arc from transition (host, obj_no) to (x,y).
*/

void start_arc(host, obj_no, x_pos, y_pos, mode)
int host, obj_no;
float x_pos, y_pos;
int mode;
{
  int i, h, no;
  char tmp[MAXLEN];
  int x1, y1, x2, y2, x, y;
  float a, theta;

  x1 = mag(x_pos, 0);
  y1 = mag(y_pos, 0);
  x2 = mag(H[host].T[obj_no].x_pos, 0);
```

```
  y2 = mag(H[host].T[obj_no].y_pos, 0);
  x = x2 - x1;
  y = y2 - y1;
  if (!x)
    theta = M_PI/2;
  else
    theta = atan((float)y/x);

  switch (mode)
    {
    case IN :
      arrowhead(x2, y2, x, y, theta);
      XDrawLine(theDisplay, theWindow, theGC,
                x1, y1,
                x2, y2); break;
    case OUT :
      XDrawLine(theDisplay, theWindow, theGC,
                x1, y1,
                x2, y2); break;
    case INH :
      XDrawArc(theDisplay, theWindow, theGC,
               x2, y2,
               dMark, dMark,
               0, 360*64);
      XDrawLine(theDisplay, theWindow, theGC,
                x1, y1,
                x2, y2);
    }
} /* -- end of start_arc(host, obj_no, x_pos, y_pos, mode) */

/* ------------------------------------------------------------------------
**
**        end_arc(from, x_pos, y_pos, mode)
**
**        Draws the end of an arc from (x,y) to the place specified by "from".
*/

void end_arc(from, x_pos, y_pos, mode)
int from;
float x_pos, y_pos;
int mode;
{
  int i, h, no;
  char tmp[MAXLEN];
  int x1, y1, x2, y2, x, y;
  float a, theta;

  strcpy(tmp, PLACES[from].name);    /* -- get name of place from */
  h = gethost(tmp);                  /* -- Look up host number and */
  no = getnum(h, tmp);               /*    place # at host  */

  if ( h >= 0 )       /* -- Place belongs to a token player  */
    {
      x1 = mag(H[h].P[no].x_pos, 0);
      y1 = mag(H[h].P[no].y_pos, 0);
    }
  else                /* -- Place belongs to the global marking vector  */
    {
      x1 = mag(GM[no].x_pos, 0);
      y1 = mag(GM[no].y_pos, 0);
    }
```

```
    x2 = mag(x_pos, 0);
    y2 = mag(y_pos, 0);
    x = x1 - x2;        /* -- Calculate angle of attack of arc  */
    y = y1 - y2;
    if (!x)
      theta = M_PI/2;
    else
      theta = atan((float)y/x);

    if ( x > 0 )        /* -- Correct 'x' alignment of entry point to place  */
      x1 -= (Dia/2*cos(theta));
    else
      x1 += (Dia/2*cos(theta));

    /* -- Correct 'y' alignment of entry point to place  */
    if ( y > 0 )
      {
        if ( x < 0 )
  y1 += (Dia/2*sin(theta));
        else
  y1 -= (Dia/2*sin(theta));
      }
    else  /* -- y > 0  */
      {
        if ( x > 0 )
  y1 -= (Dia/2*sin(theta));
        else
  y1 += (Dia/2*sin(theta));
      }

    if ( mode == IN || mode == INN)
      {
        XDrawLine(theDisplay, theWindow, theGC,
                  x1, y1,
                  x2, y2);
      }
    else
      {
        arrowhead(x1, y1, x, y, theta);
        XDrawLine(theDisplay, theWindow, theGC,
                  x1, y1,
                  x2, y2);
      }

  }  /* -- end of end_arc(from, x_pos, y_pos, mode)  */

  /* ------------------------------------------------------------------------
  **
  **        draw_arc(x1, y1, x2, y2)
  **
  **        Draws a line beeween the given points.
  */

  void draw_arc(x1, y1, x2, y2)
  float x1, y1, x2, y2;
  {
    int xa, ya, xb, yb;
    xa = mag(x1, 0);
    ya = mag(y1, 0);
    xb = mag(x2, 0);
```

```
  yb = mag(y2, 0);

  XDrawLine(theDisplay, theWindow, theGC,
            xa, ya,
            xb, yb);
}
```

A.6.5 *stats.c* — Statistics Routines

```
#include "size_limits.h"
#include <stdio.h>
#include "defs.h"
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <fcntl.h>
#include <errno.h>
#include <signal.h>
#include <math.h>
#include <X11/Xlib.h>
#include "Xdraw.h"

/*#define DEBUG1 1
#define DEBUG2 2
#define DEBUG3 3*/

extern Display *theDisplay;
extern XEvent theEvent;
extern Window theWindow, openWindow();
extern GC theGC;
extern struct obj PLACES[ MAXTRANS ];   /* -- name and host assignment of all
                                           places*/
extern struct obj TRANS[ MAXTRANS ];    /* -- name & host of all trans.  */
extern struct hosts H[ MAXHOST ];       /* -- All net infor stored here  */
extern struct gmv GM[ MAXTRANS ];       /* -- Info abt. global marking vector*/
extern int GLOBAL_PLACES;
extern int NO_HOSTS;
extern int NO_PLACES;
extern int NO_TRANS;
extern BOOLEAN display_stats;
extern BOOLEAN avg_rate;

void clear_stats();

/* ------------------------------------------------------------------------
**
**        reset_stats
**
**        Sets all transition firing counters to 0.
*/

void reset_stats()
{
  int h, t;

  clear_stats();
  for ( h = 0; h < NO_HOSTS; h++ )
    for ( t = 0; t < H[h].NO_TRANSITIONS; t++ )
```

```
      H[h].T[t].times_fired = 0;
  init_elapsed_time();
}

/* ------------------------------------------------------------------------
**
**        update_stats( buf )
**
**        Updates the statistic of all transitions reported in buf.
*/

void update_stats( buf )
short buf[];
{
  int i, h, t, x, y;
  char tmp[20];
  float rate;

  for ( i = 1; i < 1 + H[ buf[ 0 ] ].NO_TRANSITIONS; i++ )
    if ( buf[ i ] >= 0 )
      {
        h = buf[ 0 ];
        t = i - 1;
        if ( display_stats )
          {
            if ( avg_rate )     /* -- Display average rate  */
              {
                st_erase();
                sprintf( tmp, "%f", H[h].T[t].rate );
                x = mag(H[h].T[t].xr_pos, 0);
                y = mag(H[h].T[t].yr_pos, 0);
                if ( check_layer ( h ) )
                  write_tag( tmp, x, y);
                st_normal();
                H[ h ].T[ t ].old_times_fired = H[ h ].T[ t ].times_fired;
                H[ h ].T[ t ].times_fired = buf[ t + 1 ];
                rate = (float)H[h].T[t].times_fired/elapsed_time_sec();
                H[h].T[t].rate = rate;
                sprintf( tmp, "%f", rate );
                x = mag(H[h].T[t].xr_pos, 0);
                y = mag(H[h].T[t].yr_pos, 0);
                if ( check_layer ( h ) )
                  write_tag( tmp, x, y);
              }
            else                /* -- Display # of times fired */
              {
                st_erase();
                sprintf( tmp, "%d", H[h].T[t].times_fired );
                x = mag(H[h].T[t].xr_pos, 0);
                y = mag(H[h].T[t].yr_pos, 0);
                if ( check_layer ( h ) )
                  write_tag( tmp, x, y);
                st_normal();
                H[ h ].T[ t ].old_times_fired = H[ h ].T[ t ].times_fired;
                H[ h ].T[ t ].times_fired = buf[ t + 1];
                sprintf( tmp, "%d", H[h].T[t].times_fired );
                x = mag(H[h].T[t].xr_pos, 0);
                y = mag(H[h].T[t].yr_pos, 0);
                if ( check_layer ( h ) )
                  write_tag( tmp, x, y);
              }
```

```
        }
      else
        {
          H[ h ].T[ t ].old_times_fired = H[ h ].T[ t ].times_fired;
          H[ h ].T[ t ].times_fired  = buf[ t + i];
        }
    }
}

/* ------------------------------------------------------------------------
**
**        clear_stats()
**
**        Clears all stats from display.
*/

void clear_stats()
{
  int h, t, x, y;
  char tmp[20];

  for ( h = 0; h < NO_HOSTS; h++ )
    for ( t = 0; t < H[h].NO_TRANSITIONS; t++ )
      if ( avg_rate )     /* -- Display average rate  */
        {
          st_erase();
          sprintf( tmp, "%f", H[h].T[t].rate );
          x = mag(H[h].T[t].xr_pos, 0);
          y = mag(H[h].T[t].yr_pos, 0);
          write_tag( tmp, x, y);
          st_normal();
        }
      else                /* -- Display # of times fired */
        {
          st_erase();
          sprintf( tmp, "%d", H[h].T[t].times_fired );
          x = mag(H[h].T[t].xr_pos, 0);
          y = mag(H[h].T[t].yr_pos, 0);
          write_tag( tmp, x, y);
          st_normal();
        }
}

/* ------------------------------------------------------------------------
**
**        update_all_stats()
**
**        Updates the stats for all transitions.
*/

void update_all_stats()
{
  int h, t, x, y;
  char tmp[20];

  for ( h = 0; h < NO_HOSTS; h++ )
    for ( t = 0; t < H[h].NO_TRANSITIONS; t++ )
      if ( avg_rate )     /* -- Display average rate  */
        {
          sprintf( tmp, "%f", H[h].T[t].rate );
          x = mag(H[h].T[t].xr_pos, 0);
```

```
        y = mag(H[h].T[t].yr_pos, 0);
        if ( check_layer ( h ) )
          write_tag( tmp, x, y);
      }
    else                /* -- Display # of times fired */
      {
        sprintf( tmp, "%d", H[h].T[t].times_fired );
        x = mag(H[h].T[t].xr_pos, 0);
        y = mag(H[h].T[t].yr_pos, 0);
        if ( check_layer ( h ) )
          write_tag( tmp, x, y);
      }
}

/* ------------------------------------------------------------------------
**
**        display_elapsed_time()
*/
static long int time = 0;

void display_elapsed_time()
{
  char tmp[20];

  tmp[0] = '\0';
  st_erase();
  sprintf( tmp, "                %d s", time );
  write_tag( tmp, 100, 10 );
  st_normal();

  time = (long) elapsed_time_sec();
  sprintf( tmp, "Elapsed Time : %d s", time );
  write_tag( tmp, 100, 10 );
}
```

A.6.6 *Xroutines.c* — Miscellaneous Routines for X Windows

```
#include <X11/Xlib.h>
#include <X11/Xutil.h>
#include <stdio.h>

Display *theDisplay;
int theScreen;
int theDepth;
unsigned long theBlackPixel;
unsigned long theWhitePixel;
#include "theIcon"
XSetWindowAttributes        theWindowAttributes;
#define BORDER_WIDTH 2

initX()
{
        theDisplay = XOpenDisplay(NULL);

        if (theDisplay == NULL) {
             fprintf(stderr,
                "Error: Cannot establish a connection to the X Server %s\n",
                        XDisplayName(NULL));
             exit(1);        }
```

```
        theScreen               = DefaultScreen(theDisplay);
        theDepth                = DefaultDepth(theDisplay, theScreen);
        theBlackPixel       = BlackPixel(theDisplay, theScreen);
        theWhitePixel       = WhitePixel(theDisplay, theScreen);
}

quitX()
{
        XCloseDisplay(theDisplay);
}

Window openWindow(x, y, width, height, flag, theNewGC)
int x, y;
int width, height;
int flag;
GC *theNewGC;
{
        XSizeHints                              theSizeHints;
        unsigned long                       theWindowMask;
        Window                                  theNewWindow;
        Pixmap                                  theIconPixmap;
        XWMHints                            theWMHints;
        theWindowAttributes.border_pixel            = theBlackPixel;
        theWindowAttributes.background_pixel        = theWhitePixel;
        theWindowAttributes.override_redirect       = True;

        theWindowMask = CWBackPixel | CWBorderPixel;/* | CWOverrideRedirect;*/

        theNewWindow = XCreateWindow(theDisplay,/* Open a window on the dis. */
                RootWindow( theDisplay, theScreen),
                x, y, width, height,
                BORDER_WIDTH, theDepth, InputOutput,
                CopyFromParent, theWindowMask, &theWindowAttributes);

        theIconPixmap = XCreateBitmapFromData(theDisplay, /* 3) Set up icon */
                theNewWindow, theIcon_bits, theIcon_width, theIcon_height);

        /* 4) Send hints to Window Manager */
        theWMHints.icon_pixmap              = theIconPixmap;
        theWMHints.initial_state        = NormalState;
        theWMHints.flags                        = IconPixmapHint | StateHint;
        XSetWMHints(theDisplay, theNewWindow, &theWMHints);

        /* 5) Size and location for our windows */
        theSizeHints.flags          = PPosition | PSize;
        theSizeHints.x                  = x;
        theSizeHints.y                  = y;
        theSizeHints.width          = width;
        theSizeHints.height         = height;
        XSetNormalHints(theDisplay, theNewWindow, &theSizeHints);

        /* 6) create the graphics context for the window */
        if (createGC(theNewWindow, theNewGC) == 0) {
                XDestroyWindow(theDisplay, theNewWindow);
                return((Window) 0);
        }

        /* 7) Ask X to make the window visible */
        XMapWindow(theDisplay, theNewWindow);
        /* 8) Flush out all the queued up X requests */
        XFlush(theDisplay);
```

```
        /* 9) Return the window ID */
        return(theNewWindow);
}

createGC(theNewWindow, theNewGC)
Window theNewWindow;
GC *theNewGC;
{
        XGCValues theGCValues;

        *theNewGC = XCreateGC(theDisplay, theNewWindow,
                (unsigned long) 0, &theGCValues);
        if (*theNewGC == 0) {
                return(0);
        }
        else {
                XSetForeground(theDisplay, *theNewGC, theBlackPixel);
                XSetBackground(theDisplay, *theNewGC, theWhitePixel);
                return(1);
        }
}
```

A.6.7 *eventx.c* — X Event Handler

```
/*
**       eventx.c
**
*/

#include "Xdraw.h"
#include <X11/Xlib.h>
#include <X11/Xutil.h>

extern Display *theDisplay;
extern Window theWindow;
extern int layer;

#define EV_MASK (ButtonPressMask | \
                 KeyPressMask    | \
                 ExposureMask    | \
                 StructureNotifyMask)

XEvent theEvent;
refreshWindow( theExposedWindow )
Window theExposedWindow;
{
  draw_geo_info( 0 );
  XFlush(theDisplay);
}

XEventHandler()
{
  XComposeStatus theComposeStatus;
  KeySym theKeySym;
  int theKeyBufferMaxLen = 64;
  char theKeyBuffer[ 65 ];
  int new_layer;

  while( XCheckWindowEvent( theDisplay, theWindow, EV_MASK, &theEvent ) )
    {
```

```
      switch( theEvent.type )
        {
        case Expose :
#ifdef DEBUG1
          printf("Exposed\n");
#endif
          refreshWindow( theEvent.xany.window );
          break;
        case MapNotify :
#ifdef DEBUG1
          printf("Mapped\n");
#endif
          refreshWindow( theEvent.xany.window );
          break;
        case ButtonPress :
#ifdef DEBUG1
          printf("button\n");
#endif
          break;
        case KeyPress :
          XLookupString( &theEvent, theKeyBuffer, theKeyBufferMaxLen,
                         &theKeySym, &theComposeStatus );

          /* -- Entire net selected for displaying  */
          if ( ( theKeySym == 'a' ) || ( theKeySym == 'A' ) )
            layer = 0x7FF;

          /* -- Global places only  */
          if ( ( theKeySym == 'g' ) || ( theKeySym == 'G' ) )
            layer ^= 0x400;

          /* -- Specific layer selected  */
          if ( ( theKeySym >= '0' ) && ( theKeySym <= '9' ) )
            {
              new_layer = theKeySym - 48;
              /* -- If layer 'on', turn off, else turn on  */
              layer = (1<<new_layer)^layer;
            }
#ifdef DEBUG1
          printf("<%s> pressed\n", theKeyBuffer);
          printf("%x\n",layer);
#endif
          clear_net();
          refreshWindow( theEvent.xany.window );
          break;
        case ConfigureNotify :
#ifdef DEBUG1
          printf("Configuration\n");
#endif
          refreshWindow( theEvent.xany.window );
          break;
        }
    }
}

initXEvents( theWindow )
Window theWindow;
{
  XSelectInput( theDisplay, theWindow, EV_MASK );
}
```

APPENDIX B
EXAMPLE SOURCE CODE

B.1 Controller Code

B.1.1 *tname2.h* — Transition Name Definitions

```
#define     T32_AT_2        0
#define     T31_AT_2        1
#define     T30_AT_2        2
#define     T15_1       3
#define     T15_2       4
#define     T15_3       5
#define     T16_1       6
#define     T16_2       7
#define     T16_3       8
#define     T25Done_IN_AT_2         9
#define     T25Start_OUT_AT_2         10
#define     T24Done_IN_AT_2         11
#define     T24Start_OUT_AT_2         12
#define     T23Done_IN_AT_2         13
#define     T23Start_OUT_AT_2         14
#define     T21_3N      15
#define     T6_3Y_AT_2          16
#define     T21_2N      17
#define     T6_2Y_AT_2          18
#define     T21_1N      19
#define     T6_1Y_AT_2          20
```

B.1.2 *interface2.i* — Driver Subroutines

```
/* ------------------------------------------------------------------------
**
**      interface2.i
**
**      This file contains all procedures for implementing device drivers
**      (except socket initialization which _must_ be done in socket_init())
**      in player.c
**
*/

#include "tname2.h"             /* -- This include file defines the transition
                                      numbers for token player 0 of the
                                      thesis.net controller  */

#include "SIM_2/tname0.h"       /* -- This include file contains the transition
                                      numbers of the transitions of token
                                      player (thesis_sim2.net  */

extern BOOLEAN event_flag;
extern int io_socket;
extern long io_buffer[2];

long input_status = 0;          /* -- Stores the input status  (make it an array
                                      if there are several machines)  */
```

```
double rnd_01d();

/* ------------------------------------------------------------------------
**
**      precond_test( transition_number )
**
**      Checks the proconditions for the transition number.
**
**      Note that this scheme does _NOT_ work for timed transitions as this
**      checking is destructive and a timed transition has this checked twice
**      for, thus the first check erases the flag and the transition may never
**      fire.
**
**      Note use of tr_no; it corresponds to the bit checked.  The program
**      transmitting this information uses the tname<i>.h include file
**      generated by building this token player to determine which bit
**      to set.  I.e. the device driver of this program sets the preconditions
**      true by using the transition number stored in tname<i>.h.
**
*/

int
precond_test(tr_no)
int tr_no;
{

  if ( input_status & ( 0x1 << tr_no ) )        /* -- Check flag  */
    {
      input_status = input_status & ~( 0x1 << tr_no );  /* -- Clear flag  */
      return( 1 );
    }

  return( 0 );
}

/* ------------------------------------------------------------------------
**
**      postproc_test()
**
**      Writes to the device driver (in the test case another token player)
**      corresponding to the transition to be set.
**
**        Note the naming convention used for the transitions.  In the case where
**        two token players are connected, each transition name corresponds to
**      another at the other token player, except that if one is IN, the
**      other in OUT.
*/

int
postproc_test(tr_no)
int tr_no;
{
  long output_status;
  long outbuf[ 2 ];


   switch ( tr_no )
     {
     case T23Start_OUT_AT_2 :    /* -- from transition ...   */
       output_status = ( 0x1 << T23Start_IN);  /* -- to transition ...    */
```

```
      break;
    case T24Start_OUT_AT_2 :
      output_status = ( 0x1 << T24Start_IN);
      break;
    case T25Start_OUT_AT_2 :
      output_status = ( 0x1 << T25Start_IN);
      break;
    }
  outbuf[ 0 ] = 0;
  outbuf[ 1 ] = output_status;

  printf("\nIO trmt'd: %x %x \n", outbuf[0], outbuf[1] );

  /* -- Send status to token player or device driver */
  if ( write ( io_socket, outbuf, sizeof( outbuf ) ) == -1 )
        syserr( "write io_socket" );

}


/* ------------------------------------------------------------------------
**
**      decode_io()
**
**      Decodes the I/O received on the io_socket.
**
**      io_buffer[0] : device # id.
**      io_buffer[1] : status word.
**
*/

void
decode_io()
{
  input_status |= io_buffer[1];
  event_flag = TRUE;
  printf("\nIO rec'd: %x %x ", io_buffer[0], io_buffer[1] );
}


/* ------------------------------------------------------------------------
**
**      conflict_resolution(tr_no)
**
**      Returns true/false with probability Transition[tr_no].firing_rate
**      for an immediate transition.  This is a crude method of simulating
**      conflict between immediate transitions that have assigned
**      probabilities of firing.
**
**      Note that the porbablilites must be scaled depending on marking and
**      how many transitions are in conflict to get the correct results
**      (like it is done in SPNP).
**
*/

int
conflict_resolution( tr_no )
int tr_no;
{
  double rnd, tmp;
```

```
  rnd = rnd_01d();
  tmp = (double) Transition[ tr_no ].firing_rate;

  if ( rnd < tmp )
    return( 1 );
  else
    return( 0 );
}
```

B.1.3 *tr_links2* — Links for Subroutine Calls

```
/* ------------------------------------------------------------------------
**
**      Initialization file for Host 2 controller
**
**      Note that both the flag must be set and the address must be assigned
**      in order for the process to be executed.
*/

Transition[ T6_1Y_AT_2 ].preconditions = 1;
Transition[ T6_1Y_AT_2 ].preprocess = conflict_resolution;

Transition[ T21_1N ].preconditions = 1;
Transition[ T21_1N ].preprocess = conflict_resolution;

Transition[ T6_2Y_AT_2 ].preconditions = 1;
Transition[ T6_2Y_AT_2 ].preprocess = conflict_resolution;

Transition[ T21_2N ].preconditions = 1;
Transition[ T21_2N ].preprocess = conflict_resolution;

Transition[ T6_3Y_AT_2 ].preconditions = 1;
Transition[ T6_3Y_AT_2 ].preprocess = conflict_resolution;

Transition[ T21_3N ].preconditions = 1;
Transition[ T21_3N ].preprocess = conflict_resolution;


Transition[ T23Done_IN_AT_2 ].preconditions = 1;
Transition[ T23Done_IN_AT_2 ].preprocess = precond_test;

Transition[ T24Done_IN_AT_2 ].preconditions = 1;
Transition[ T24Done_IN_AT_2 ].preprocess = precond_test;

Transition[ T25Done_IN_AT_2 ].preconditions = 1;
Transition[ T25Done_IN_AT_2 ].preprocess = precond_test;

Transition[ T23Start_OUT_AT_2 ].postprocessing = 1;
Transition[ T23Start_OUT_AT_2 ].postprocess = postproc_test;

Transition[ T24Start_OUT_AT_2 ].postprocessing = 1;
Transition[ T24Start_OUT_AT_2 ].postprocess = postproc_test;

Transition[ T25Start_OUT_AT_2 ].postprocessing = 1;
Transition[ T25Start_OUT_AT_2 ].postprocess = postproc_test;
```

B.2 Simulator # 2 Code

B.2.1 *portnums.h* — Socket Port Numbers for Simulator

```
#define OFFSET 75          /* -- Offset to prevent use of same address between
                              simulator an controller  */
#define PLAYERPORT 1500+OFFSET    /* -- Port number for token ring sockets  */
#define COMMANDPORT 2500+OFFSET   /* -- Port number for command sockets  */
#define DISPLAYPORT 3500+OFFSET   /* -- Port number for display sockets  */
#define IOPORT 4500+OFFSET        /* -- Port number for i/o with device driver.
                                       Note - no offset since it connects with
                                       the server socket of the controller */
```

B.2.2 *tname0.h* and *tr_links0.i* — Name and Link Definitions

```
#define        T24Done_OUT        0
#define        T23Done_OUT        1
#define        T25Done_OUT        2
#define        T25Start_IN        3
#define        T24Start_IN        4
#define        T23Start_IN        5
```

B.2.3 *interface0.i* — Driver Routines

```
/* ------------------------------------------------------------------------
**
**      interface.c
**
**      This file contains all procedures for implementing device drivers
**      (except socket initialization which _must_ be done in socket_init())
**      in player.c
**
*/

#include "../tname2.h"          /* -- This include file defines the transition
                                      numbers for token player 2 of the
                                      thesis.net controller  */

#include "tname0.h"             /* -- This include file contains the transition
                                      numbers of the transitions of this token
                                      player (thesis_sim2.net  */

......

NOTE : STUFF DELETED HERE IS IDENTICAL TO ABOVE LISTING OF interface2.i

.....

  switch ( tr_no )
    {
    case T23Done_OUT :   /* -- from transition ...  */
      output_status = ( 0x1 << T23Done_IN_AT_2); /* -- to transition ...   */
      break;
    case T24Done_OUT :
      output_status = ( 0x1 << T24Done_IN_AT_2 );
      break;
    case T25Done_OUT :
      output_status = ( 0x1 << T25Done_IN_AT_2 );
      break;
```

```
    }
  outbuf[ 0 ] = 0;
  outbuf[ 1 ] = output_status;

  /* -- Send status to token player or device driver */
  if ( write ( io_socket, outbuf, sizeof( outbuf ) ) == -1 )
        syserr( "write io_socket" );

  printf("\nIO trmt'd: %x %x ", outbuf[0], outbuf[1] );

}

.........

DELETED
```

B.2.4 *player.c* — Listing of init_sockets()

```
NOTE - ONLY DIFFERENCE BETWEEN REGULAR PLAYER.C FILE AND THIS IS THE FOLLOWING
       ROUTINE:
/* ------------------------------------------------------------------------
**
**      init_socket()
**
**      Initializes the sockets for global marking vector token ring.
**
**      token_in is socket for the server of SERVERNAME that runs on this host
**      and reads the token message from SERVERNAME.
**
**      token_out is the socket for the client to CLIENTNAME and writes to
**      this.
**
*/

init_sockets()
{
  char temp;

  /* -- Connect to XDisplay  */
  out_X_win = client_setup( HOSTONAME, DISPLAYPORT );

#if NO_HOSTS > 1
#ifdef HOST0   /* -- This is token player # 0   */
  /* -- Setup SERVER connection for token player # 1   */
  token_in = serv_setup_intr( HOSTNAME, PLAYERPORT,
                              "Server waiting for pn1 ..." );
  setblock( token_in, FALSE );

  /* -- Wait until the last token player has opened its server for token
        player 0   */
  printf(" Hit Enter when last token player server is ready : \n");
  scanf( "%s", &temp );
  token_out = client_setup( SERVERNAME, PLAYERPORT );
#else    /* -- This is not token player 0  */
  token_out = client_setup( SERVERNAME, PLAYERPORT);
  token_in = serv_setup_intr( HOSTNAME, PLAYERPORT,
                              "Server waiting for pn(i+1)" );
  setblock( token_in, FALSE );
#endif
```

```
#endif
  command_in = serv_setup_intr( HOSTNAME, COMMANDPORT,
                                "Server waiting for XCommand ..." );
  setblock( command_in, FALSE );

  /* -- If the token player is a controller, this is a client socket  */
#if HOSTNO == 0
#ifdef EXTERNAL_IO
  io_socket = client_setup( HOSTNAME, IOPORT,
                            "Server waiting for device driver program ..." );
  setblock(io_socket, FALSE );                    /*    receive commands from
                                                        device driver */
#endif
#endif
  signal( SIGIO, input_interrupt_handler );  /* -- Set up interrupt to be
                                                   issued when i/o occurs */

}       /* -- socketinit()   */
```

B.2.5 *event_handler.c* — Listing of get_event()

```
NOTE : THE FOLLOWING ROUTINE IS THE ONLY DIFFERENCE BETWEEN THE FILE FOR
       THE CONTROLLER AND THE FILE FOR THE SIMULATOR

/* ------------------------------------------------------------------------
**
**          get_event()
**
**          Reads all the input sockets and pipes.  If there is any info,
**          decodes this.
**
**          Reads:
**
**          - command_in socket from XCommand window
**
**          - endpipe pipe from timed_trans_handler
**
**          - token_in socket from the previous token player
**
**          - io_socket socket from device driver program.
**
*/

void
get_event()
{
  long token_buffer[ GMV_BUFSIZE ];/* -- Buffer for received marking vector */
  long command_buffer[ 2 ];

  int nread, i;

#ifdef EXTERNAL_IO
  /* -- Check if there is anything from the device driver program to read  */
  if ( nread = read( io_socket, io_buffer, sizeof( io_buffer ) )
       > 0 )
    {
      decode_io();
    }
#endif
```

```
  /* -- Check if there is a command available to read  */
  if ( nread = read( command_in, command_buffer, sizeof( command_buffer ) )
       > 0 )
    {
      decode_comd( command_buffer[ 0 ] );
    }

#if NO_HOSTS > 1
  /* -- Check if it is the global marking vector that is available  */
  if ( nread = read( token_in, token_buffer, sizeof( token_buffer ) ) > 0 )
    {
      get_gmv( token_buffer );    /* -- gmv available */
#ifdef DEBUG
      printf("gmv read %d bytes : ",nread);
      for (i = 0; i < sizeof(token_buffer)/4; i++)
        printf(" %d",token_buffer[i]);
      printf("\n");
#endif
    }
#endif
}
```