THE STOCHASTIC PETRI NET ANALYZER SYSTEM DESIGN TOOL FOR BIT-MAPPED WORKSTATIONS

Michael K. Molloy and Prentiss Riddle

Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712-1188

TR-86-12 May 1986

ABSTRACT

This report provides a description of a new CAD tool for Stochastic Petri Nets. The report is intended as a program document to help a user or developer to understand the programs that make up the tool. The primary philosophy for the software is to maintain a flexible design environment while providing support tools to quickly and easily evaluate different aspects of the design. Several components of the software are innovative and are described in some detail. The user interface is graphical with highly interactive layout and editing features. The most significant feature is the inherent support for hierarchical modeling of systems.

1 Introduction

Over the last two decades, Petri Net [Petr66] models have been used in a wide area of applications. More recently, Petri Net models have been extended to include the concept of time. These models fall into two basic classes, Timed Petri Nets (TPN) [Ramc74, Rama80, Zub80] and Stochastic Petri Nets (*SPN*) [Moll81, Natk80, Moll82]. The TPN model has the concept of an underlying clock that clicks off time and can have synchronous transition firings. The Stochastic Petri Net model has the underlying notion of time as a random variable where events are asynchronous. The bridge between the two models is the discrete time Stochastic Petri Net model [Moll85a] with unit probabilities.

Several application areas have adopted one or more of these models. Research in flexible manufacturing, PERT analysis, computer architectures, system reliability, and network protocols are the most familiar examples.

There have been several modifications of these basic models, in addition to the the various semantic interpretations of firing rules. First, the Generalized Stochastic Petri Nets [Mars84] extend Stochastic Petri Nets by adding the concept of instantaneous transitions. Second, the Extended Stochastic Petri Nets [Duga84] add features to relax the exponential assumption on the transition firing rates. Third, the Generalized Timed Petri Net [Holl85] adds a selection probability to groups of transitions but maintains the discrete time concept. Fourth, Stochastic Petri Nets with 'new better than used' distributions for firing times are being used to specify regenerative simulation models [Haas85]. All of these models have had analysis software written to support their application.

All of these software packages provide similar features for general Markov analysis which were previously available, without the need to directly specify all of the possible states and state transitions. These include, analytical solutions where possible, and simulations when necessary. Since many new design problems are difficult or impossible to cast into a product form queueing network model, the need for user friendly Markov analysis tools, based upon a *SPN* representation, is clear. Unfortunately, the widespread use of such tools will require an improved user interface.

2 Stochastic Petri Nets

Recall the formal description of Petri Nets [AGE79] where the model PN has places P, transitions T, input and output arcs A and an initial marking M.

$$
\begin{aligned}
&PN \triangleq (P,T,A,M)\\
&P = \{p_1,p_2,\ldots,p_n\}\\
&T = \{t_1,t_2,\ldots,t_m\}\\
&A \subset \{P\times T\}\cup\{T\times P\}\\
&M = \{\mu_1,\mu_2,\ldots,\mu_n\}
\end{aligned}
\tag{1}
$$

The marking may be viewed as a mapping from the set of places P to the natural numbers N.

$$
M:P\rightarrow N \text{ where } M(p_i)=\mu_i \text{ for } i=1,2,\ldots,n \tag{2}
$$

Define for a Petri Net PN the set function I of input places for a transition t.

$$
I(t)\triangleq\{p \mid (p,t)\in A\} \tag{3}
$$

Define for a Petri Net PN the set function O of output places for a transition t.

$$
O(t)\triangleq\{p \mid (t,p)\in A\} \tag{4}
$$

As is common in practice, a Petri Net can be drawn using circles to represent places and bars (or boxes) to represent transitions. Tokens, to denote a marking, are represented as dots inside the circles (places). A five place, five transition Petri Net could look like the one shown in figure 1.

The continuous time stochastic Petri Net $SPN \triangleq (P,T,A,M,\lambda)$ is extended from the Petri Net $PN \triangleq (P,T,A,M)$ by adding the set of average, possibly marking dependent, transition rates $\lambda=\{\lambda_1,\lambda_2,\ldots,\lambda_m\}$ for the exponentially distributed transition firing times. These models are equivalent to continuous time homogeneous Markov chains. So, the SPN model may be considered as a concise representation, or generator, of the Markov representation of a system.

The SPN model, by the nature of its being a generator, is very useful in the verification of the state space of the system. It has all of the formal mechanisms to verify that the model correctly represents the the system, before the performance analysis is attempted. Since the entire verification and analysis phase are automated, no errors are introduced as would be if a separate model was used for the performance analysis phase.

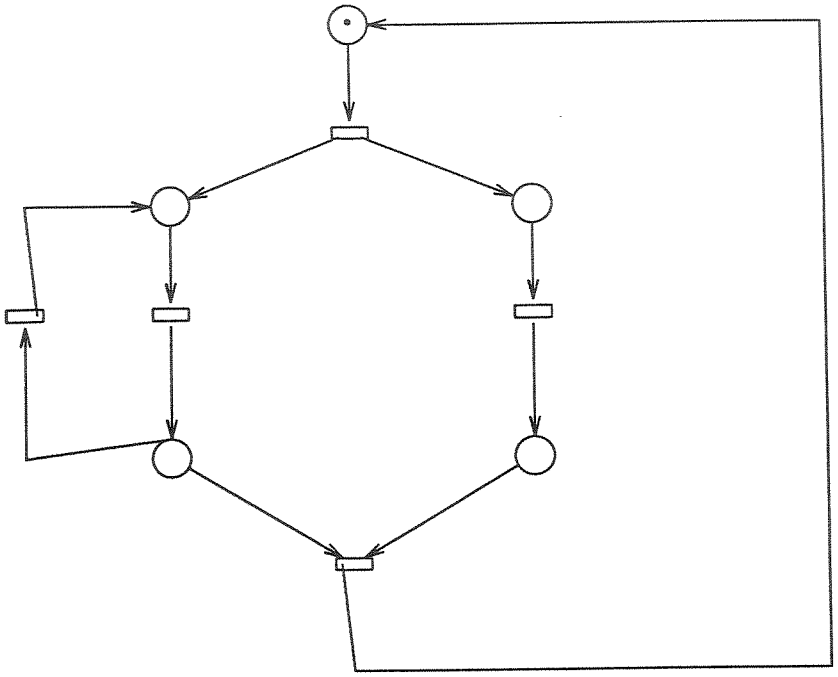


Figure 1 An Example Petri Net

3 Tool Description

This section describes the tool in some detail. The basic model of the Stochastic Petri is augmented with a new transition, called a subnet transition. This transition is used to represent an entire subnet defined (or about to be defined) in the library of nets on disk. The basic goal of the hierarchical model is to keep the size of the current net under design (representing some component at some level of abstraction) to a manageable size. We anticipate that a *SPN* model would be made up of a very high level description with lots of subnet transitions. Then, at the next level of abstraction, a successive refinement of those transitions would associate another net structure to that transition.

The following subsections describe each of the three major features of the tool that are unique and innovative. The first subsection describes the user interface, its philosophy, function, and use. The second subsection describes some of the special design features of the analysis components. The last subsection describes the current approach to hierarchical modeling along with some of the problems encountered with the approach.

3.1 The User Interface

The basic philosophy of the user interface is simple. The user interface should be flexible enough to let the designer be creative (even artistic) while removing any tedium from the activity. This is accomplished by adopting the following goals.

1. The user interface should provide very rapid, easily selectable options.

2. Anything that is repetitious or algorithmic should be done by the tool.

3. All of the rules should be gently enforced by the tool.

4. Do not try to be smarter than the designer.

5. Anything that can be done, can be undone.

6. Options should be split into logical categories with different access for each.

7. Graphical representations of options are the best for the visually oriented human.

8. Both a top down and bottom up approach must be supported.

The user interface is a tool under the SUNTOOLS ® window management software. The tool has all of the features of a normal window. It can be moved, modified in size or shape, closed into icon form, and exist in multiple copies on the screen. The mouse is the primary input device. All drawing, menu selection, and analysis activities are initiated by mouse key clicks. The keyboard is used only in those cases when arbitrary text input is needed. The basic layout of the tool, the icons, and an example drawing can be seen in figure 2. The tool has three basic subwindows which make up three of the five functional user interface components.

1. The menu subwindow (it can be on the left or right hand side) is a collection of icons representing drawing and editing features.

    a. Create a Place

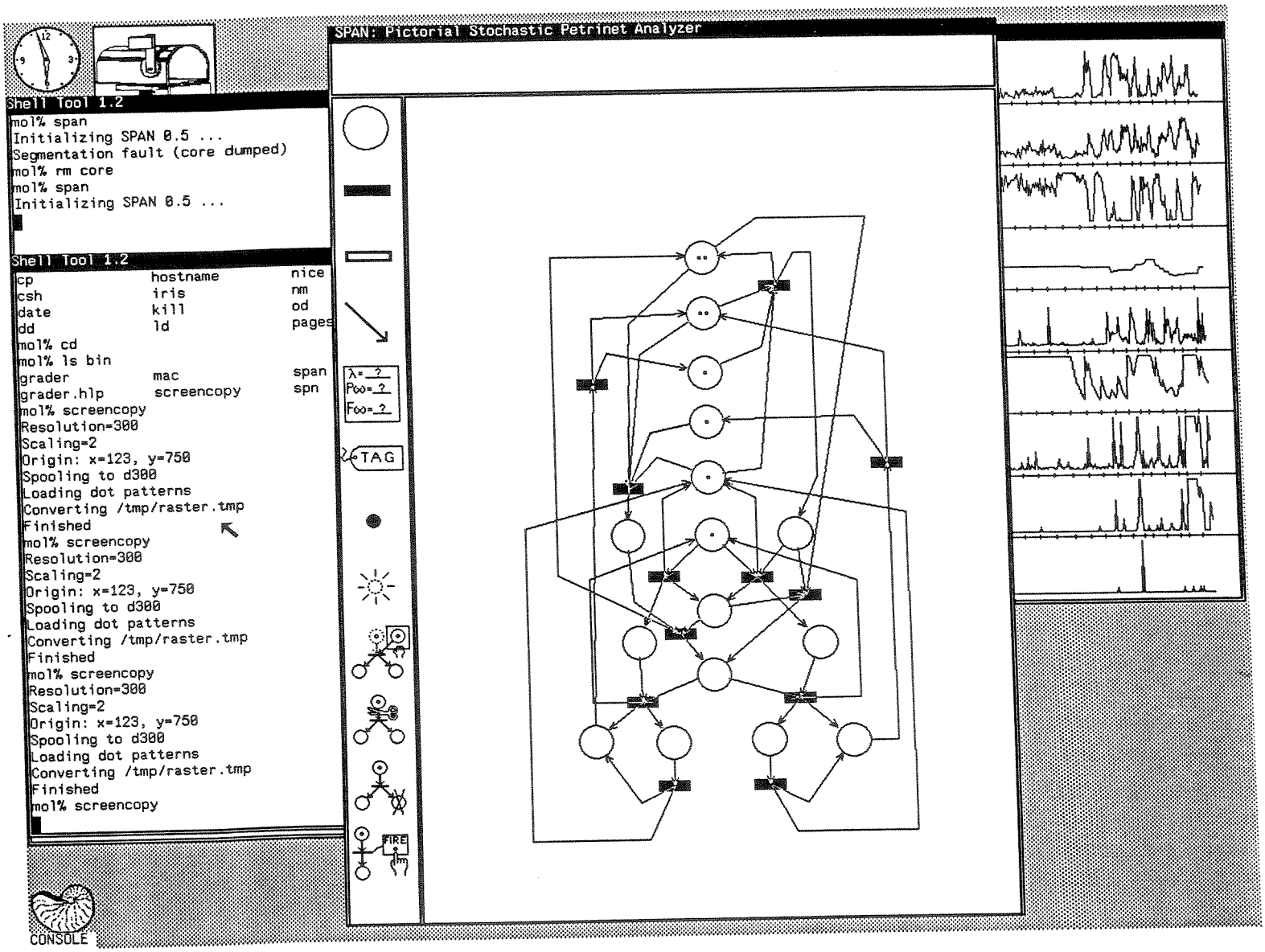
    b. Create a Transition

    c. Create a Subnet Transition (a transition representing a subnet)

    d. Create an Arc

    e. Annotate a Transition (specify firing rate, etc.)

® SUNTOOLS is a registered trademark of Sun Microsystems Inc.


Shell Tool 1.2
mol% span
Initializing SPAN 0.5 ...
Segmentation fault (core dumped)
mol% rm core
mol% span
Initializing SPAN 0.5 ...
Shell Tool 1.2
cp hostname nice
csh iris nm
date kill od
dd ld pages
mol% cd
mol% ls bin
grader mac span
grader.hlp screencopy spn
mol% screencopy
Resolution=300
Scaling=2
Origin: x=123, y=750
Spooling to d300
Loading dot patterns
Converting /tmp/raster.tmp
Finished
mol% screencopy
Resolution=300
Scaling=2
Origin: x=123, y=750
Spooling to d300
Loading dot patterns
Converting /tmp/raster.tmp
Finished
mol% screencopy
Resolution=300
Scaling=2
Origin: x=123, y=750
Spooling to d300
Loading dot patterns
Converting /tmp/raster.tmp
Finished
mol% screencopy
SPAN: Pictorial Stochastic Petrinet Analyzer
λ=_?
P∞=_?
F∞=_?
TAG
FIRE
CONSOLE

f. Tag a Node (Place, Transition, or Subnet) with an ASCII string

g. Add a Token

h. Delete a Token (add an 'anti-token')

i. Move a portion of the Net

j. Delete an Arc

k. Delete a Node

l. Manually Fire a Transition or Subnet

2. The message subwindow on the top of the tool window which is used for error messages, comments, and text input to selections.

3. The canvas subwindow where all the actual drawing is done. The canvas subwindow represents a window into the drawing space. It may not show all of a particular *SPN*, but can be moved around or scaled to do so.

The basic idea of the tool is similar to the popular MacPaint for the MacIntosh computer. The menu icons are used to select an action. The selection is performed by pointing at the icon, and clicking the left mouse button. Once the selection is made, the cursor can be moved back to the drawing canvas, where it changes to represent the action selected. In the case of creating places, transitions, and subnet transitions, the cursor looks like the object to be created. Once a location is determined (pointed to by the cursor) the left mouse button is clicked to create an instantiation of the object. In a similar fashion, tokens can be added or removed from places, ASCII tags can be modified, and transition firing rates can be changed. In all cases, the user selects an option from the menu and activates the option by pointing and clicking the left mouse button. An English description of the option is displayed in the message subwindow and the icon is displayed in reverse video.

Since many drawing options are not semantically correct for Stochastic Petri Nets (such as connecting transitions to transitions), the tool enforces the rules. The user gets an error message in the message subwindow when an illegal action is taken, and the action is not be allowed to occur. There are several such rules.

1. An object can not be placed on top of another object.

2. Transitions and subnet transitions can only be connected to places.

3. Deleting a place, transition, or subnet transition deletes all the arcs associated with that node.

4. All locations are lined up with an invisible grid so that being close to lining up becomes exactly lined up.

5. Tokens and Anti-tokens can only be put into places.

6. Anti-tokens have no effect when there are no tokens present.

7. Only transitions and subnet transitions can be annotated with firing rates.

There are two additional functional components of the tool. These can be seen in figure 3. First, a popup menu (textual), activated by the right mouse button, provides options for the manipulation of the drawing environment. These include several commands (again activated by pointing).

1. Undo - undo the last drawing action (icon menu only)

2. Refresh - clear and redisplay the net.

3. Zoom In - magnify the image in the canvas subwindow with the cursor location as the new center.

4. Recenter - move the entire drawing space to center on the cursor location.

5. Zoom Out - reduce the image in the canvas subwindow with the cursor location as the new center.

6. Clean Up - align a coarse drawing onto the imaginary grid points. (This is for files generated by other programs.)

7. Quit - abort the current session. A confirmation is required if any modifications have been made.

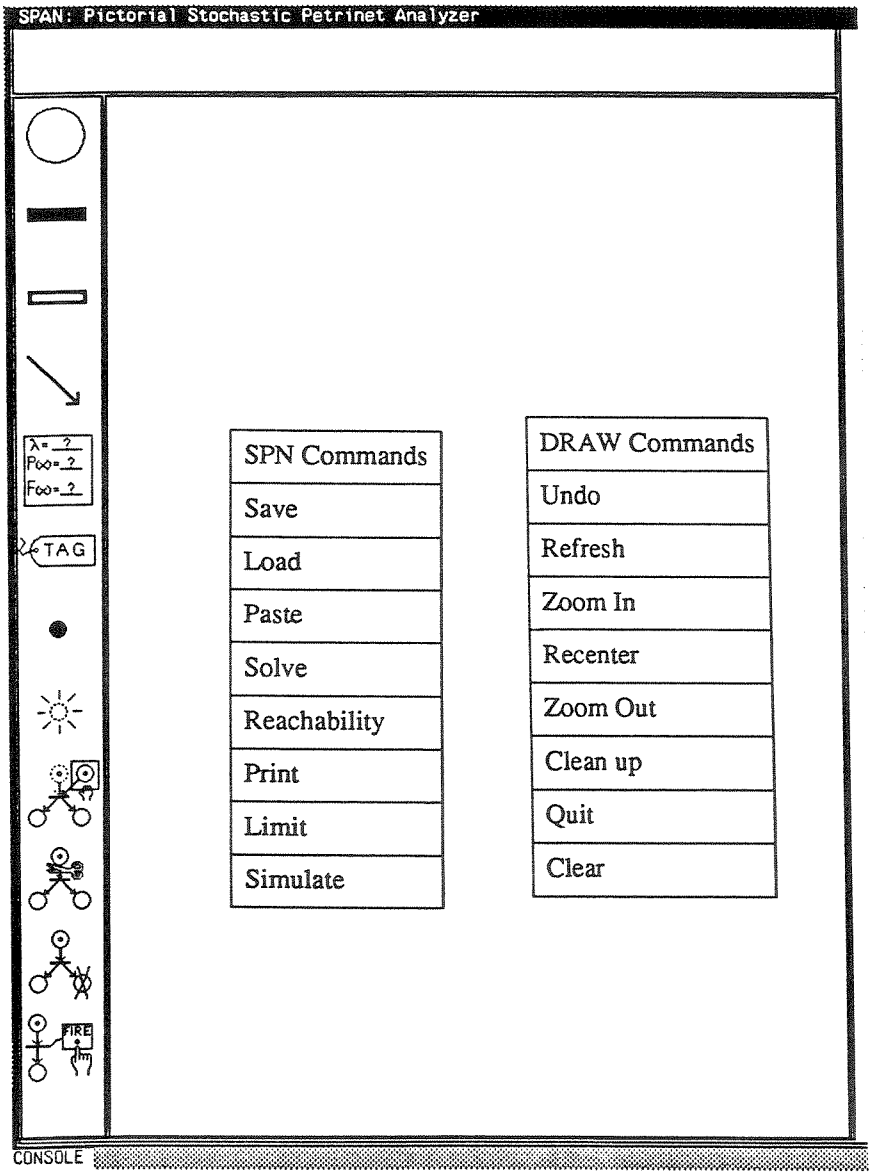
8. Clear - clear the screen and destroy the current net. A confirmation is required if any modifications have been made.

Second, a popup menu (textual), activated by the middle mouse button, provides analysis options for studying the network currently designed.

1. Save - save the current design in a disk file. The file format is PIC compatible for inclusion in DITROFF documents.

2. Load - load the current design from a disk file.

3. Paste - read in another net description and paste it on the current design.


SPAN: Pictorial Stochastic Petrinet Analyzer
λ= ?
P∞= ?
F∞= ?
TAG
FIRE
SPN Commands
Save
Load
Paste
Solve
Reachability
Print
Limit
Simulate
DRAW Commands
Undo
Refresh
Zoom In
Recenter
Zoom Out
Clean up
Quit
Clear
CONSOLE

4. Solve - generate the reachability set, construct the Markov matrix, solve the matrix for the steady state solution, and display the average number of tokens in each place.

5. Reachability - generate the reachability set, and send a printable copy of the reachability tree to the standard output.

6. Print - generate the reachability set, construct the Markov matrix, solve the matrix for the steady state solution, and display the average number of tokens in each place. Send a copy of the token probability densities for each place to the standard output.

7. Limit - solve for bounds on the throughput of the *SPN* using the bottleneck analysis technique [Moll85b].

8. Simulate - run a Monte Carlo simulation to animate the display of the *SPN* using a mouse controlled throttle for the clock rate.

3.2 Special Data Structures

Because maintaining information about the graphical representation needs to be flexible for ease of editing, it is not appropriate for analysis. Since analysis requires speed and compact size, an entirely different structure was used for nets when the analysis phase is initiated. Each of the basic structures are described below. The result of carefully selecting each of these data structures is the superior performance of the tool. Most of the design time went into selecting and coding these structures. All of the structures are allocated dynamically to increase the range of the tool. The allocation of the reachability set and Markovian structures is only done when an analysis is requested, and deallocated when it is complete. This philosophy has made it possible for this system to draw, edit and analyze *SPN* with a large number of nodes and at least 5000 states in the reachability set, in an interactive environment.

3.2.1 Graphical Data Structures

The data structures for drawing are object oriented. Many of the data structures are built upon some simplier data structures useful for drawing, such as coordinates and positions.

```
struct coordinate       { int                    x, y;
                          struct coordinate      *next;
                        };

struct position         { int          x, y };
```

A *SPN* is simply a collection of pointers to lists of objects. The lists of objects include, places,

transitions, subnet transitions, arcs, and compound objects.

```
struct net_object        { struct position          nwcorner;
                           struct position          secorner;
                           int                      boxht;
                           int                      boxwid;
                           int                      circlerad;
                           struct place_object      *places;
                           struct trans_object      *trans;
                           struct subnet_object     *subnets;
                           struct arc_object        *arcs;
                           struct text_object       *texts;
                           struct compound_object   *compounds;
                           struct compound_object   *next;
                         };
```

Each element in the lists are data structures of a type for that list. All structures for places have information about the location of the place, its ASCII label, the number of tokens, etc.

```
struct place_object      { char                     *tag;
                           short                    tokens;
                           struct position          center;
                           struct place_object      *next;
                         };
```

All structures for transitions have information about the location of the transition, its ASCII label, its firing rate, etc.

```
struct trans_object      { char                     *tag;
                           short                    orient;
                           float                    fire_rate;
/*                         char                     test();
                           char                     func();     */
                           struct position          center;
                           struct trans_object      *next;
                         };
```

All structures for subnet transitions have information about the location of the subnet transition, its ASCII label, its firing rate, the filename where the complete subnet description is (if any), etc.

```
struct subnet_object    { char                          *tag;
                          short                          orient;
                          struct pnet                   *subnet;
                          char                          *file_name;
                          float                         fire_rate;
/*                        char                           test();
                          char                           func();    */
                          struct position                center;
                          struct subnet_object          *next;
                        };
```

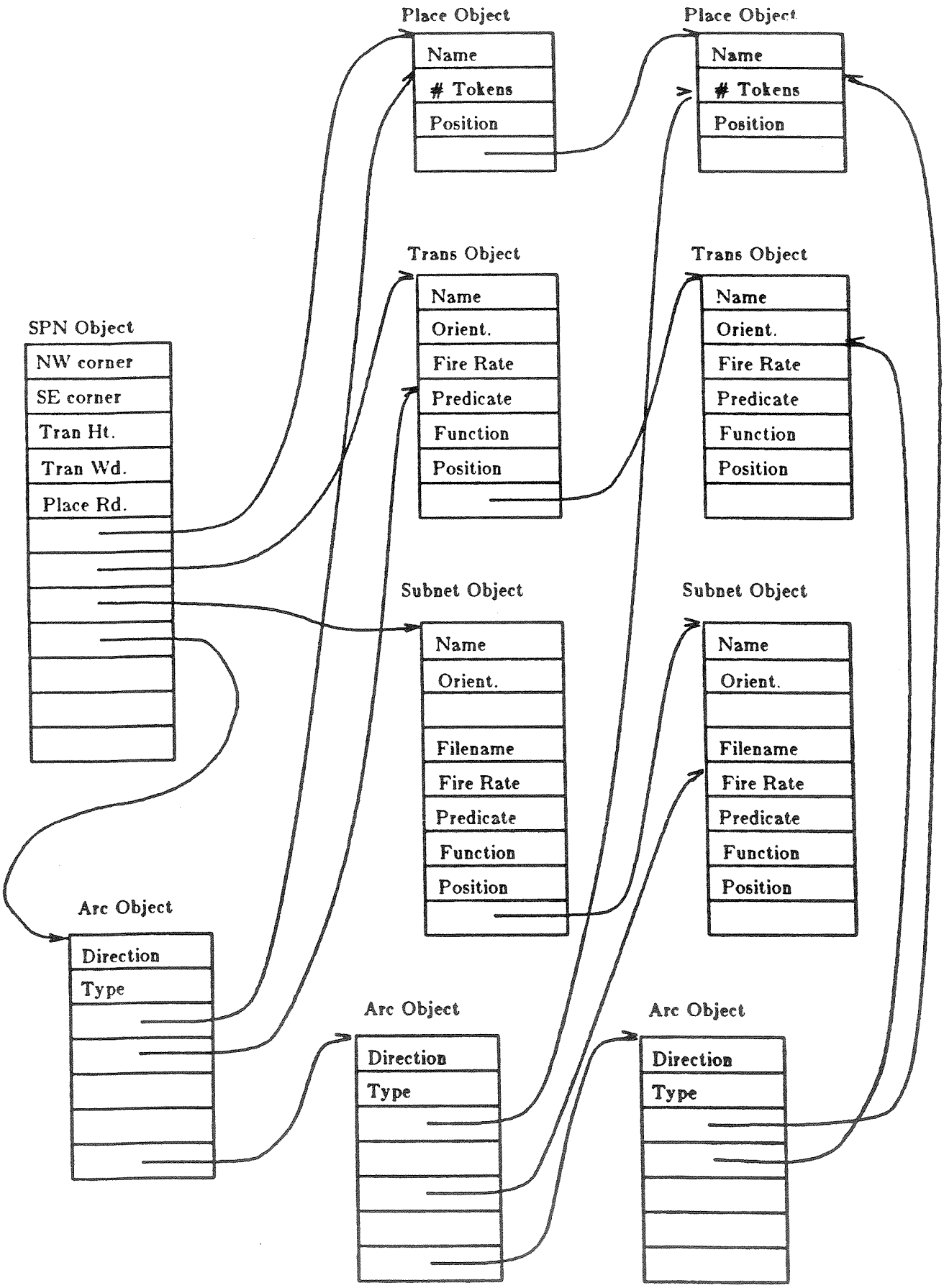
All structures for arcs have information about the sequence of points (arcs can be segmented) which make up the arc, the direction of the arc (to or from a place) and pointers to the specific place, transition, or subnet transition the arc connects. Note that there must be at least one NULL pointer, and you can never allow a transition to be connected to a subnet and vise-versa.

```
struct arc_object       { char                          direction;
                          char                          type;
                          struct place_object           *place;
                          struct trans_object           *trans;
                          struct subnet_object          *subnet;
                          struct coordinate             *point;
                          struct arc_object             *next;
                        };
```

In addition to all of these, there is a backup net structure to save deletions for the undo of a deletion.

In addition to these basic structures, some other structures are used for bookkeeping during the editing process. When a section of the Petri Net is selected on the screen for moving, a compound object is constructed to keep track of what parts of the net are being moved. In addition, copying sections of a net would use such data structures.

```
struct compound_object  { struct position             nwcorner;
                          struct position             secorner;
                          struct place_list           *places;
                          struct trans_list           *trans;
                          struct subnet_list          *subnets;
                          struct arc_list             *arcs;
                          struct text_list            *texts;
                          struct compound_object      *compounds;
                          struct compound_object      *next;
```


Place Object
Name
Tokens
Position
Place Object
Name
Tokens
Position
Trans Object
Name
Orient.
Fire Rate
Predicate
Function
Position
Trans Object
Name
Orient.
Fire Rate
Predicate
Function
Position
SPN Object
NW corner
SE corner
Tran Ht.
Tran Wd.
Place Rd.
Subnet Object
Name
Orient.
Filename
Fire Rate
Predicate
Function
Position
Subnet Object
Name
Orient.
Filename
Fire Rate
Predicate
Function
Position
Arc Object
Direction
Type
Arc Object
Direction
Type
Arc Object
Direction
Type

```
};

struct place_list        { struct place_object     *place;
                           struct place_list       *next;
                         };

struct trans_list        { struct trans_object     *trans;
                           struct trans_list       *next;
                         };

struct subnet_list       { struct subnet_object    *subnet;
                           struct subnet_list      *next;
                         };

struct arc_list          { struct arc_object       *arc;
                           struct arc_list         *next;
                         };

struct text_list         { struct text_object      *text;
                           struct text_list        *next;
                         };
```

4 Reachability Data Structures

This section is intended to explain the data structures used for the reachability analysis in the Stochastic Petri Net CAD tool SPAN. Additional documentation is, of course, to be found in the code itself.

Because of the different tasks involved in the editing of a Petri Net and performaing analysis on it, and the desire to optimize the reachability analysis for speed, it was decided to represent the Petri Nets in SPAN with two distinct sets of data structures (defined in the file "object.h") and the reachability analysis data structures (described in this section and defined in the file "reach.h"). The first step in performing the analysis on a Petri Net, is to convert the net from its form in the editing data structures, to the analysis data structures. That conversion is performed by the code in "conv.c".

4.1 Petri Net Representation

The fundamental type used to represent a Petri Net is a structure called pnet.

```
struct pnet { struct transition    *transtab;
              unsigned              ntrans;
              unsigned              alloctrans;
              struct place         *placetab;
              unsigned              nplaces;
              unsigned              allocplaces;
              char                 *initmark;
              char                  omega;
            };
```

In addition to defining the initial marking 'initmark', and the maximum number of tokens per place 'omega', the pnet contains points and other values necessary to define two tables, the place table and the transition table. (For the time being, subnet transitions are treated no differently from ordinary transitions during the reachability analysis.) Each of the two tables is a contiguous vector of structures representing transitions or places, addressable by adding an integer offset to the point in the pnet structure. Thus, if the variable "net" is the current pnet "*(net.placetab + j)" would address the jth element of the place table and "*(net.transtab + k)" would address the kth element of the transition table. The order of the places and the transitions in the tables is arbitrary; they are added to it during the execution of "conv()" starting at the low-memory end of each table. The storage for the tables grows dynamically as needed. Initially, "conv()" calls "emalloc()" in order to allocate sufficient storage for a certain number of elements in each table. When the table is full, it calls "erealloc()" to add room for more elements to the table (how many more elements to make room for is set by the constant ALLOCINC). The pnet fields "ntrans" and "nplaces" are the number of transitions and places which have benn added so far to their respective tables, and the variables "alloctrans" and "allocplaces" are the number of transitions and places for which room has been allocated so far.

The structures comprising the elements in the place and transition tables are structures of type place and structures of type transition.

```
struct place { struct arc   *out;
               char         *name;
               int           screenx;
               int           screeny;
             };

struct transition { struct arc          *in;
```

```
    struct arc          *out;
    char                *name;
    float               rate;
    struct pnet         *subnet;
};
```

The name field in each structure points to its ascii label. The place structure contains the place's screen co-ordinates in "screenx" amd "screeny", which are used later for the display of solution results on the screen. The transition structure contains the transition's firing rate in the variable "rate" and a pointer to its associated subnet in the variable "subnet". If the "subnet" variable is NULL, then the transition is a simple one. (This field is currently unused.) In addition, both structures contain pointers to lists of arcs connecting them to other nodes in the net. A transition has a list of both input and outpu arcs (in the fields "in" and "out"), but a place has only a list of output arcs (in "out"). Since a transition affects both the input places and the output places when it fires, pointers to both must be maintained. However, a place can only affect transitions which are at the end of output arcs, so only a list of ouput arcs is maintained.

The arcs connecting places and transitions are represented by linked lists of elements of type "arc".

```
struct arc { int                index;
             int                narcs;
             struct arc         *next;
           };
```

The "index" field specifies a place or transition by serving as an integer offset to be added to the "placetab" or "transtab" pointer in pnet. (Which table it is depends upon the context. So if you are scanning a list off of an transition type, it is an offset into the placetab, and vice-versa.) The "narcs" field is used in cases of multiple arcs connecting the same two nodes and pointing in the same direction. Only one item in an arc list can contain a given "index" value, and "narcs" contains the weighting factor for it. The "next" field points to the next element in the list of arcs. The order of the elements in an arc list is arbitrary. See the routine "addarc()" to see how the arc list is built.

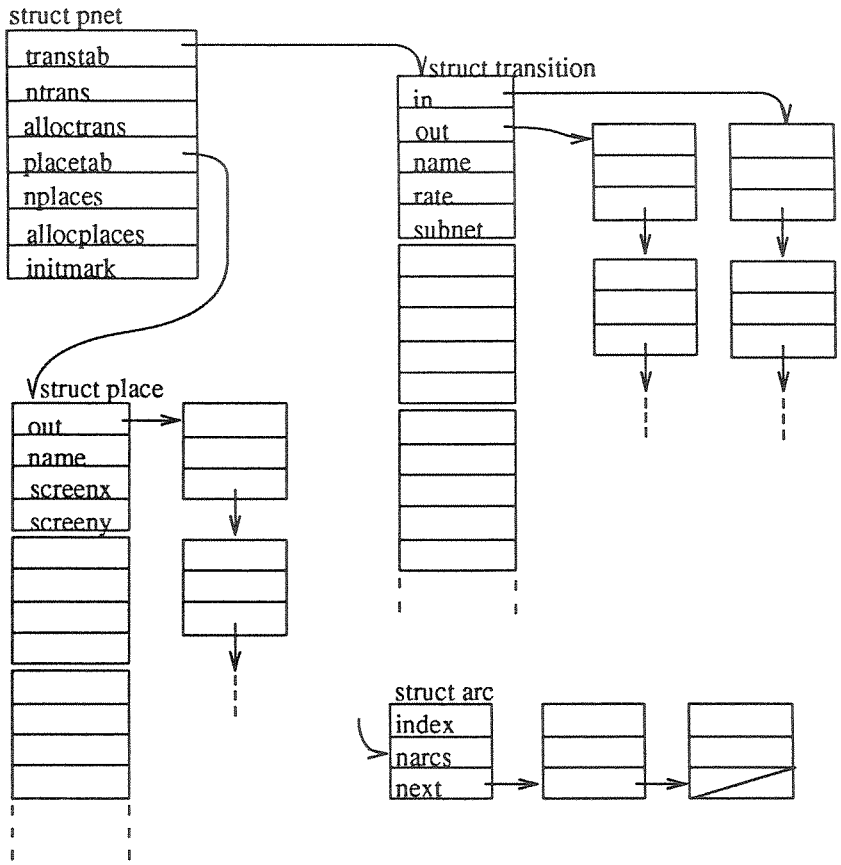


Figure 5 Petri Net Reprentation

4.2 Reachabiltiy Graph Structures

The most important data structure in the reachability graph is that structure which is used to represent the markings themselves. A marking is tored as a string of bytes, allocated using "malloc()" as a vector of type "char" and thus referred to by a pointer of type "char *". Each byte contains the number of tokens in an associated place in the Petri Net. The order corresponds to the order of the entries in the place table. Once the network has been contructed, the number of places remains fixed so the size of all marking in a given net analysis is the same (unlike real character strings, amrking are not NULL terminated). The individual token counts are packed into bytes and unpacked from bytes using the following macros.

```
#define MPACK(i)          (char)(i-128)
#define MUNPACK(c)          (int)(c+128)
```

The algorithm used to generate the reachability graph is straightforward. Given a marking, a list of enabled transitions is generated. Each of these transitions is fired, and each marking that results from those firings is checked against a list of markings that have already been seen. If it is new, it is added to queue of markings to be checked further. Every time a transition is fired, an arc is recorded connecting the parent marking with the child marking. This process is repeated for each marking on the queue. When the queue is empty, the graph is complete.

Some of the data structures used for this are equally straightforward. The list of enabled transitions for the marking under consideration and the queue of markings retained for further consideration, are simple linked lists.

```
struct shortlist { short              s;
                   struct shortlist  *next;
                 };

struct queue     { char               *marking;
                   struct queue       *next;
                 };
```

The "s" field in the shortlist structure is an index into the transition table of the current pnet. The "marking" field of the queue structure points to the same copy of a marking as is stored in the reachability graph.

The reachability graph itself is slighty more complicated. The primary structures used are the reachgraph structure and the childlist structure.

```
struct reachgraph { struct childlist   *children;
                    int                 index;
                    char               *marking;
                    char                visited;
                  };

struct childlist { struct reachgraph   *child;
                   short               trans;
                   struct childlist    *next;
                 };
```

Each unique marking in the graph is pointed to by the "marking" field of one reachgraph structure. Each transition that is enabled by that marking is represented by an element in a linked list structure of childlist. Each childlist has a child pointer to the reachgraph node associated with the marking resulting from the firing of that transition. In this way the two structures represent a muliway graph of markings connected by arcs to thir immediate descendants.

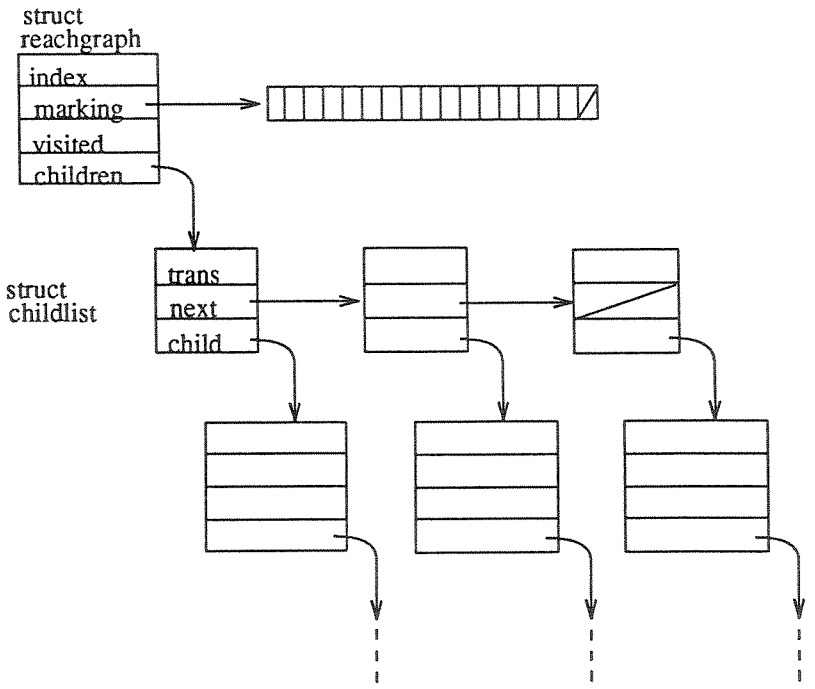


Figure 6 Reachability Graph Representation

Some of the fields in the structures still need some explanation. The "trans" field of the childlist structure is an index into the transition table of the current pnet. The "visited" field of the reachagraph structure is used when recursively traversing the graph: as each node in the graph is visited, the "visited" field is set to some flag value. Encountering the same flag value gain indicates a loop in the graph and that the node's descendants need not be visited again (see the routine "printgraph()" for an example). The "index" field is used for a companion data structure, the sorttree.

An additional pair of data structures are used for fast lookup of markings and thir associated elements in the reachability graph. During execution, a binary sorted tree of markings is built using the sorttree structure.

```
struct sorttree { char                  *markp;
                  struct reahgraph      *graphp;
                  struct sorttree       *left;
                  struct sorttree       *right;
                };
```

The pointer "markp" in the binary sorted tree, is a pointer to the associated node in the reachability graph "graphp". The pointers to the left and right subtrees ("left" and "right") contain elements whose markings are lexicagraphically less or greater respectively. The ordering of markings is checked by the function "markcmp()". No attempt is made to ensure that the tree remains balanced. (Which may be a source of some performance quirks associated with large problems.)

Once the reachability graph has been generated, the storage overhead associated with the marking tree is no longer needed and the routine "tree_to_vec()" converts the tree to a simpler structure that accomplishes the same thing. The final structure is a vector of pointers into the reachability graph, sorted by marking. This is complimented by the "index" field of the reachgraph structure, which is an integer index into this sorted marking vector. Thus, there is a two-way lookup between the reachability graph and the marking vector when the reachability analysis is complete.

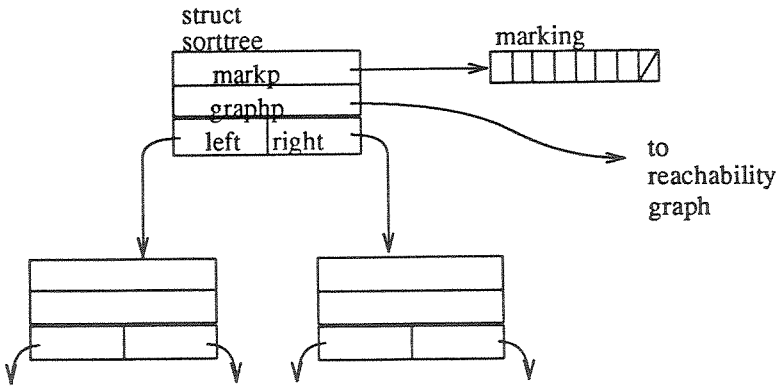


Figure 7 Binary Sorted Marking Tree

4.3 Markov Data Structure

The end result of the analysis is the generation of the equivalent Markov chain. Since this tool is currently restricted to continuous time Stochastic Petri Nets, the resulting matrix tends to be very sparse. Therefore, the data structure used for the matrix is the uncompressed pointer storage structure defined for the Yale Sparse Matrix software package [Eise76].

5 Program Structure

The basic program structure follows the hierarchy of the tool. There is a main program that gets the size of the tool (default for initial, and suntool driven for reshaping), then calls three routines to handle the set up of three subwindows. The routines which implement the actions requested by a user are all interrupt driven from mouse actions. In the case of keyboard input, it must be enabled first by some mouse action before keyboard input will be accepted.

There are some global routines used throughout the program. These routines, emalloc(), eremalloc(), null_proc(), etc. are all in the file "global.c". Variables needed by all the other routines are also in that file. Constants are defined in an include file "const.h" which is then included in most routines. Complex data structures are also defined in include files for easy access by routines without compromising consistency control.

5.1 Tool Specific Routines

There any three groups of routines which are specific to the SUNTOOLS ® environment. This routines handle the various functions when the cursor is in the particular subwindow. The system environment will activate the tool selected function whenever the cursor is within the borders of SPAN and will activate the sw_selected function whenever the cursor is over some particular subwindow. Note that the cursor is always there, even if it is not visible. The only way to have an invisible cursor is have all of the bits off.

5.1.1 Canvas Subwindow

The drawing subwindow (the canvas) is handled by routines in the file "canvas.c". Many of the routines are similar for each of the subwindows. The first routine is 'init_canvas()' which calls the SUNWINDOWS software to create the subwindow, initialize the mouse interrupt handlers, and create a pixel rectangle on the screen. Several of the initializations put references to routines defined locally to handle special interrupts.

The routine 'canvas_selected()' is called whenever the system has determined that the cursor is over the rectangle defined for this subwindow. This routine will then check the input event queue for any errors, and will call the normal interrupt handler if there are none.

The normal interrupt handler for the canvas subwindow is the routine "canvas_reader()" which gets the input event data structure as input. If the event is a negative event, the routine ignores the input. If the event is a keyboard input event, then that will be processed if the receiving_msg flag is set, otherwise the input is ignored. If the event is a mouse key input event, then the particular action pointed to by the pointer 'reader_left' is called for left button clicks, the spn menu is popped up by the routine 'set_spn_menu()' if the middle button is clicked, and finally the drawing menu is popped up by the routine 'set_drawing_menu()' if the right button is clicked.

Since other actions, such as elastic box drawing, must respond to any movement of the mouse, not just key clicks, another interrupt handler can be used to replace the normal one. That routine, 'canvas_tracker()', is used to handle interrupts for each movement of the mouse. That routine uses the x,y coordinates of the mouse position rather than the input event data structure

for the normal interrupt handler. The routine 'reset_canvas_reader()' is used to reinitialize the normal interrupt handling.

The other basic routines to handle damage to the window due to an overlapping window, 'canvas_sighandler()', 'redisplay_canvas()' and 'clear_canvas()' are also found in the file "canvas.c".

5.1.2 Icon Menu Subwindow

As in the case of the canvas subwindow, the basic routines to initialize, respond to damages, and size changes are similar. The new idea in the icon menu subwindow is the drawing of the icons found in the file "menu.h" and locating the icon pointed to by the cursor when the left mouse button is clicked. When the icon menu subwindow is selected (i.e. the system calls the routine 'menu_selected()', the flags 'action_on' and 'receiving_msg' are checked before proceeding. If either of these conditions are true, the user must have moved the cursor before the action or inputing of a response was complete, so they icon menu selection is ignored. If these conditions are false, then the code for the input event is checked. Only left mouse button clicks are accepted as a valid input, and when received, the position of the mouse is read to determine which icon has been selected. This is accomplished by the routine 'menu_action()'. Menu actions can be very complex. They can require the changing of interrupt pointers, the outputting of messages, etc. Each action is described by a section of code in a case statement. When an action is taken ('set_command()' is executed), then the icon associated with that action is changed to reverse-video to provide visual feedback to the user that the key click was recognized, and will remain that way so the user knows what mode the tool is operating in.

In the case that the changes to the tool position/size have had a major effect on the size of the subwindows, the menu may be laid out in two rather than the default number of columns. This is accomplished by the routine 'menu_layout()'.

5.1.3 Message Subwindow

Finally, the message subwindow is the simplest of the subwindows used in the tool. The usual 'init_msg()' and 'msg_selected' routines are included in the file "msg.c". The main interrupt handling routine is called 'msg_consume()' and it continues to copy keyboard input into a buffer until a carriage return is received. The routine then turns the 'receiving_msg' flag off and passes the pointer to the buffer onto the use routine. The message receiving is set up by 'init_msgreceiving()' which includes arguments to print in the message subwindow as an explanation or as a prompt. Several other routines, such as 'clear_message()', 'blink_msg' and 'put_msg' are available to change the contents or get someone's attention without requiring keyboard input.

5.1.4 Popup Menus

There are two popup menus which are handles by the routines in the file "popup.c". There are two basic routines to handle the two popup menus. The first routine 'set_spn_menu()' is for the spn command popup menu activated by the middle mouse button. The second routine 'set_drawing_menu()' is for the drawing specific commands that are not in the icon menu subwindow.

Each of these routines calculates the menu item selected by noting the position of the cursor when the left mouse button is clicked. Then, through a case statement, the appropriate tasks are performed.

The remaining routines in this file are to interface the action specified by the popup menu selection with other routines found else in the program structure. The three routines 'write_file()', 'add_net()', and 'read_new_net()', are simple routines which get the character string for file input/output. The actual file input/output is performed by routines found the file "fileio.c".

5.2 Drawing Specific Routines

In order to draw objects on the screen, the routines which manipulate individual pixels and pixel rectangles are used. This was done to speed up the execution of the tool. In the future, even the place and transition objects will be predrawn in a user buffer and then copied directly to the screen using "pix_rect" routines. There are several routines which are used to do the basic drawing of boxes, circles, and lines. These routines work at the pixel level and are not scaled. Other routines which scale the coordinates for the particular zoom level will call the basic routines. The basic routines 'draw_rectbox()', 'draw_vector()', and 'elastic_box()' are found in the file "box.c". The basic routines for drawing circles (places) 'circle()' and tokens 'dot()' are found in the file "circle.c".

5.2.1 Creating Objects

All of the routines for creating and drawing objects are located in the file "draw.c". These routines are quite straightforward. The only special notes are that a conversion from the coordinates picked up from the mouse position must be converted to the drawing coordinates before using them. In addition, special variables 'cur_x', 'cur_y', 'fix_x', and 'fix_y' are used to keep track of the last position of drawing actions. The drawing of an object on top of an object already on the screen erases the object. Because and "undo" command is implemented, the last place, transition, subnet, or arc that was created is left in the pointer variables 'cur_place', 'cur_trans', 'cur_subnet', and 'cur_arc'.

The actual creation of an object is accomplished by selecting an iconic menu item (which sets up a pointer to the appropriate routine) and then clicking the left mouse button at the position where the object is to be created. The places are created by the routine 'new_place()', transitions are created by the routine 'new_trans()', subnets are created by the routine 'new_subnet()', and arcs are created in two phases by the routines 'new_arc()' and 'add_segment()'. Arcs are special since the arc tracks the cursor until a left mouse click pins it down. If the arc is pinned down in a place away from a legal node the routines continue to track and add segments until a legal node (place for transitions and subnets, or a subnet/transition for a place).

The routines to actually draw the object are equally easy to recognize. To draw a place, the routine 'draw_place()' is executed. Similarly, for transitions, 'draw_trans()', for subnets 'draw_subnet()' and for arcs, 'draw_arc()' are the routines executed. By definition, arcs end in an arrowhead, so the order of the linked list is important. The actual elements in the list are the line segments in the arc. The pointers to the starting and ending nodes are for reference only and not used to actually draw an arc. They are used to calculate the truncation position for the end of the arc during its creation. This means that the last point of an arc can look funny after a node is moved. It will still be attached, but the segment may not point to the center of the node any more. An additional the routine 'draw_seg()' is used to draw or erase the segmented line laid down by the creation actions.

Of special note is an underlying assumption about the location of all objects. In order to create nicely aligned drawings, it was too tedious for the user to line up everything. Therefore, all objects must fall on an invisible grid the width of a transition. This means that the creation of an object need only be very close, not exactly on the same x or y coordinate to line up with another object. The routine to change the coordinates is called 'align()' and is found in the file "align.c". The routine 'align_arc()' uses that function to straighten out a newly created arc.

Since several special editing actions can be done to a net design, a couple of special functions are required. First, the drawing of a compound object (a list of real objects) is accomplished by the routine 'draw_compound()'. Second, the routines to check for proximity to other objects are found in the file "near.c". A complete net structure is returned from the routine 'near_obj()' and 'near_node_obj()' with NULL pointers in every list except the one with the object found. Those routines are defined in terms of separate routines for each type of object 'near_place_obj()', 'near_trans_obj()', 'near_subnet_obj()', and 'near_arc_obj().

5.2.2 Editing Objects

All of the routines for editing an existing net graphical object are found in the file "edit.c". The routines include the functions for delete and undo actions.

In order to easily edit objects and still undo the edit, an additional data structure is involved. A backup net structure 'bkup_netobj' is used to store deleted objects until the deletion is committed (i.e. the user has selected some other action beyond the delete action). Once an action is committed, the backup net object is thrown away. This is accomplished by executing the routine 'free_bkup_netobj()'.

The routines which perform the deletion action are simply called 'delete_arc()', 'delete_node()' (for places, transitions, and subnets). Those routines will find the appropriate object and call the routines which manipulate the data structures 'remove_arc()', 'remove_place()', 'remove_trans()', and 'remove_subnet()'.

Since any action which can be done, can be undone, every one of the action routines has a corresponding undo routine. So the routines 'undo_new_place()', 'undo_new_trans()', 'undo_new_subnet()', and 'undo_new_arc()' perform their obvious functions. Since an undo of an undo is considered legal, the routines which undo the creation actions also put the removed object into the backup net object, 'bkup_netobj'. The delete routines also have their corresponding undo functions. So the routines 'undo_delete_arc()' and 'undo_delete_node()' are also available to undo a deletion.

In addition to the simple delete objects and undo commands an additional feature has been added to read in a file with a net description to augment the net being designed. This is accomplished to in two phases. The first reads in the net (showing the projected position as a bounding box), allows the user to move that around until a mouse click drops the net at that position. The routine 'paste()' finds the offset of the new net and calls 'merge_nets()' to actually put them together after which it frees the now empty header for 'tmp_netobj'. The routine 'merge_nets()' puts the objects found in 'tmp_netobj' into the data structure 'netobj' which is the current net under design.

There are many other aspects to a design, besides the actual objects which may be changed. Associated with each object is an ASCII label called a tag. This may also be changed. The routines to edit the tags are found in the file "tag.c". The routine 'show_tag()' will simply display the current tags for all the objects on the screen. The location of the tag is fixed, and is located a little to the right of each object. The routine 'edit_tag()' will set up the message subwindow to receive a new ASCII tag to replace the current one. All objects received a default tag of numbered P's, T's, or S's depending on the type of the object. The routine 'get_new_tag()' will actually read in and check the new tag for conformity to the rules for PIC. All labels must begin with a capital, non-numeric character, and no two labels may be the same.

Since this is again an editing function, an undo must also be possible. The routine 'undo_edit_tag()' will undo any change to the tag.

Another aspect of the design which may be modified is the firing rate for the transitions and subnets. In a future SPAN system, more than just a firing rate may be associate with the events represented by the transitions or subnets, but for now the routine 'annotate()' simply changes the firing rate for a transition or subnet. This edit can also be undone by the routine 'undo_annotate()'.

Since markings are also a major part of the design, there are several routines to manipulate the markings. These routines are found in the file "mark.c". Increasing the number of tokens in a place is accomplished by executing the routine 'incr_token()'. Similarly, decreasing the number of tokens in a place is accomplished by executing the routine 'decr_token()'. Since these are also editing features, they can be undone by executing 'undo_incr_token()' and 'undo_decr_token()'. As an option to check a net design manually, the manual fire action has been included. This action is implemented by the routine 'fire_trans()' and can be undone by 'undo_fire_trans()'. The actual firing is done by the routine 'exec_trans_fire()'. Since transitions can not be fired unless they are enabled a routine to check to see if a transition is enabled is provided. The routine 'trans_enabled()' will return a one if the transition is enabled and zero otherwise. Similarly, for subnets, the routines 'exec_subnet_fire()' and 'subnet_enabled()' are used.

5.2.3 Scaling Objects

Since net designs may become quite large, and arcs into and out of places or transitions can be hard to distinguish, a facility for zooming in (enlarging) and zooming out (shrinking) is provided. The coordinate system is drawn at normal scale when the zoom level is zero. The zooming magnifies or reduces the image by a power of two. So there are a maximum of two levels above and below the normal display level. At the maximum reduction, the normal (there is no real limit) drawing region would fit on the entire screen.

Since the screen input values from mouse position are fixed pixel locations, the position must be translated into the coordinates for the drawing at the appropriate zoom level. The routine 'project()' will convert the given coordinates to the appropriate coordinates at the drawing plane, given the current zoom level. The routine 'zoom()' will convert the given drawing coordinates into the proper screen coordinates.

The actions of zooming in and zooming out which are initiated by the selection of a menu item from the popup menus, are performed by the routines 'zoom_in()' and zoom_out()'. Both of these routines take the current cursor location as the new center of the display window (canvas area) when the zoom is completed. Another routine 'recenter()' simple takes the cursor

input as the new center of the display window without changing the zoom level. This provides a easy way to pan through a large net design.

5.3 Analysis Specific Routines

The analysis specific routines include all of the actions initiated by the spn popup menu. The actions include the saving and recalling of net descriptions, the reachability analysis of the spn, the Markovian analysis of the equivalent Markov chain, an animation of the firing behavior of the spn, and various printout routines to provide some hardcopy of an analysis.

5.3.1 Disk Read/Write

Since the file format is an ASCII level format for the TROFF preprocessor PIC, several routines are needed to simply parse the input and create the data structures, or vice-versa. The routine 'check_bounds()' is used to see if newly parsed objects are outside the currently defined bounding box for the net. This can be used as a check for funny files, or to note when to change the bounding box in a dynamic way. The routine 'load_file()' is the routine that actually parses the PIC input file. Since such a routine is very complex, several functions are separately defined. The routine 'parsearc()' is used to look through an arc definition for the starting and ending points, the intermediate points, and the starting and ending objects. The routine 'resolve_arc()' is used to find the pointers to the objects which are the starting and ending objects of the arc. Since many of the inputs have values associated with then, such as coordinates, the routine 'scanalnum()' will scan a line for alphanumeric strings.

The routine 'save_file()' is the routine that actually writes out the contents of the data structures which make up the spn, into a file in PIC format. This task is far simplier since the program has control over the data structures and can output a fixed format for every type of object. The routine 'printbullets()' will construct and output of PIC constructs to draw tokens (bullets) in a nicely arranged pattern.

5.3.2 Reachability Analysis

The reachability analysis routines are at the heart of the SPAN system. The main routine for this analysis is the routine 'reach()' which returns a pointer to a 'reachgraph' structure. Several routines are used in the process of building the reachability set. The routine 'enabled_list()' will determine the enabled transitions for a marking and return them is a linked list. The routine 'fire()' will apply a transition to the marking to produce a new marking. The routine 'insertmarking()' will find or insert a marking in the binary sorted tree. If the marking is already in the tree, the marking created by the routine 'fire()' is discarded (freed up). The rou-

tine 'lookupmarking()' will use the binary sorted tree to find a marking. The routine 'extendgraph()' is used to add an entry into the reachability tree with a pointer to the marking being already inserted into the list of markings. The routine 'markcmp()' will simply compare two markings for equivalence, or dominance. The routines 'qpush()' and 'qpop()' will push or pop a marking onto the currently active queue. The routine 'leaf_check()' will check the tree for leaf nodes. The routine 'tree_to_vec()' will convert the original reachability tree to the more condensed form of a vector.

6 Conclusions

The CAD tool, SPAN, for the analysis of Stochastic Petri Models has proven to be very useful. The design goals of a highly interactive, user friendly, and high performance design aid have been attained. The system can handle large nets by breaking them down in a hierarchical manner and can solve medium size nets in a relatively short time (given the fact that the SUN 2/120 ® is poor at floating point). A summary of the timings for the standard example net with increasing reachability set size is given below.

Tokens (#)	States (#)	Reachability (sec)	Markovian (sec)
1	5	<1	<1
2	14	<1	<1
3	30	<1	1
4	55	<1	4
5	91	1	6
6	140	3	19
7	204	6	45
8	285	10	110
9	385	18 (14)	283
10	506	27 (21)	---
11	2212	270 (130)	---
12	5322	1300 (862)	---

The sudden increase in processing time for the cases above 500 states is not completely understood. At this time, we believe that at 500 states, paging becomes a significant factor on our SUN 2/120 ® workstation since we have only 2meg of memory. This belief is supported by the timings on a 3meg SUN 2/120 ® which are shown in parentheses. The lack of data for the time on the Markovian analysis for large systems is due to the fact that those routines are currently in FORTRAN and can not do dynamic memory allocation. Therefore, if SPAN does not allocate enough space for the Yale package, the program aborts the Markovian analysis.

® SUN is a registered trademark of Sun Microsystmes Inc.

This is currently under conversion to avoid the problem.

It is clear from even the small amount of use the package has seen, that several changes are needed. The human interface for reachability analysis must be changed to support some type of query-response dialogue rather than a long listing. In a similar vein, the Markovian results need to be represented differently. It would be much better to point at an arc, and get the throughput for that arc. Similarly, pointing at a transition would trigger the display of the utility of that transition. Finally, pointing at a place could give you the mean time until a token returns to that place.

Some of the drawing features need to updated. It would be nice to select and duplicate sections of a *SPN* in a fashion similar to the way we move sections of a net. The addition of a mouse activated keypad would further minimize the number of keyboard actions required.

7 Acknowledgements

This software could not have been written without the help of two outstanding individuals. First, a graduate student, Supoj Sutanthavibul, spearheaded all of our efforts by creating several tools on the SUN with such great style, that several sections of the SPAN code still bear his name. Second, Prentiss Riddle actually wrote half of the code with the author and provided exactly the right balance for the completion of this project in only 3 man-months. The software is available, royality free, from the author. (See the attached license agreement for details.)

8 References

Duga84 Dugan, J.B.; Trivedi, K.S.; Geist, R.M.; Nicola, V.F. "Extended Stochastic Petri Nets: Applications and Analysis" *Performance 84,* pp. 507-519, Paris France, Dec. 1984

Eise76 Eisenstat, S.C.; Schultz, M.H.; Sherman, A.H. "Considerations in the Design of Software for Sparse Gaussian Elimination" *Sparse Matrix Computations* Ed. J.F. Bunch, D.J. Rose, pp. 263-273, Academic Press 1976 also in Proceedings of Symposium on Spares Matrix Computations at Argonne National Laboratory, Sept 1975

Haas85 Haas, P.; Shedler, G. "Regenerative Simulation of Stochastic Petri Nets" *Proceedings of the Workshop on Timed Petri Nets,* Torino, Italy, July 1985

Holl85 Holliday, M.A.; Vernon, M.K. "A Generalized Timed Petri Net Model for Performance Analysis" *Proceedings of the Workshop on Timed Petri Nets,* Torino, Italy, July 1985

Mars84 Marsan, M.; Balbo, G.; Conte, G. "A Class of Generalized Stochastic Petri Nets" *ACM Transactions on Computer Systems* Vol 2, pp. 93-122, May 1984

Moll81 Molloy, M.K. "On the Integration of Throughput and Delay Measures in Distributed Processing Models" Report CSD-810921, University of California, Los Angeles, 1981

Moll82 Molloy, M.K. "Performance Analysis Using Stochastic Petri Nets" *IEEE Transactions on Computers* Vol. C-31, No. 9, Sept. 1982, pp. 913-917

Moll85a Molloy, M.K. "Discrete Time Stochastic Petri Nets" *IEEE Transactions on Software Engineering* Vol. SE-11, No. 4, April 1985, pp 417-423

Moll85b Molloy, M.K. "Fast Bounds for Stochastic Petri Nets" *Proceedings of The Workshop on Timed Petri Nets,* Torino Italy, July 1985

Natk80 Natkin, S.O. "Les Reseaux de Petri Stochastiques et leur Application a L'Evaluation des Systemes Informatiques" Doctoral Thesis, Conseratoire National des Arts et Metiers, 1980

Petr66 Petri, C.A. "Communication with Automata" PhD Thesis, Translated by C.F. Green, Information System Theory Project, Applied Data Research Inc., Princeton N.J., 1966

Pete77 Peterson, J.L. "Petri Nets" *Computing Surveys* ACM Sept 1977 Vol 9 No 3 pp 223-252

Ramc74 Ramchandani, C. "Analysis of Asynchronous Concurrent systems by Timed Petri Nets" PhD Thesis, MIT 1974 Project Mac report #MAC-TR-120

Rama80 Ramamoorthy, C.V.; Ho, G.S. "Performance Evaluation of Asynchronous Concurrent Systems using Petri Nets" *IEEE Transactions on Software Engineering* Vol SE-6 No. 5 Sept 1980 pp. 440-449

Zube80 Zuberek, W.M. "Timed Petri Nets and Preliminary Performance Evaluation" *Proceedings of the 7th annual Symposium on Computer Architecture* 1980, pp. 88-96

SOFTWARE LICENSE

Purpose

This Agreement between _____, hereafter referred to as the LICENSEE, and Michael K. Molloy, hereafter referred to as the LICENSOR, is to provide a mutually agreed upon mechanism to coordinate the distribution and use of the licensed software.

Covered Software

The software licensed under this Agreement is the Stochastic Petri Net Analyzer, hereafter referred to as SPAN. The software includes the source code, documentation, and object code for a Sun Microsystem's SUN 2/120 workstation. The following software modules make up the said software.

Makefile	canvas.c	drag.c	markov.c	reach.c
README	choices.h	draw.c	menu.c	reach.h
align.c	circle.c	edit.c	menu.h	span
alloc.c	const.h	fileio.c	msg.c	span.doc
annotate.c	conv.c	global.c	near.c	tag.c
arrow.c	cursor.h	main.c	object.h	undo.c
box.c	debug.c	mark.c	popup.c	zoom.c

Two modules are supplied in object form only. The routines ndrv.o and odrv.o are compiled FORTRAN modules. These routines are part of the Yale Sparse Matrix solver and are not the authored by the LICENSOR. They are included as public domain software and for the convenience of the use of the SPAN software by the LICENSEE. Any reference or questions about these routines should be directed to the authors A. W. Sherman, S. C. Eisenstat, and M. H. Schultz.

Resale

The resale of this software is expressly prohibited. No copy, in part or whole, or modification of this software may be sold. By accepting this license, the LICENSEE agrees to make available, under the same terms as this Agreement, any modifications of the software.

Distribution

The software is distributed royality free and any charges are for reproduction, postage and handling only. Redistribution of the licensed software by the LICENSEE is prohibited, but any third party may request and receive a separate license. This restriction is intended to provide control on the location of copies of the software for the purpose of distributing bug reports and updates whenever possible.

Transfer of License

The LICENSEE may not sublicense, assign or transfer the license or the program except as expressly provided in this Agreement. Any attempt otherwise to sublicense, assign or transfer any of the rights, duties or obligations hereunder, is void.

Reproduction and Modification

The LICENSEE is granted permission to copy the software as many times as is necessary for the LICENSEE's site. Copies for backup purposes are recommended. The LICENSEE is responsible for enforcing the terms of this Agreement for all copies of the software made by the LICENSEE. Modification of the software is not only allowed, but encouraged. However, copies of any modifications that significantly extend the capabilities or reliability of the software should be sent to the LICENSOR for redistribution to other sites.

Publication

The licensed software was created for the purpose of performing research on the concepts embedded within the software. If the software is used for further research and/or applications which lead to publications, the LICENSEE agrees to reference the author as the primary source for the software. In addition, any subsequent modifications of this software which are distributed under this Agreement, will contain references to the authors of the modifications.

Warranty

The is no warranty, expressed or implied, on the functioning of the licensed software. Since the software is licensed at no charge, the LICENSOR can not be held financial responsible for damages of any kind. The software has been tested and evaluated for completeness and correctness to the best of the author's ability. However, errors may still exist in the code and any damage or loss incurred by the LICENSEE in the use of the licensed software is the sole responsibility of the LICENSEE.

Termination

This license Agreement is in effect until terminated.

Failure of the LICENSEE to abide by all of the terms specified in this Agreement, will terminate the license. The LICENSEE may terminate this Agreement at anytime by submitting a request, in writing, to the LICENSOR. The Agreement may also be terminated by mutual agreement between the LICENSEE and the LICENSOR.

Upon Termination of this license, all copies held by the LICENSEE must be destroyed. Furthermore, the LICENSEE must certify to the LICENSOR, in writing, within thirty days of the termination, that the copies have been destroyed.

LICENSEE	LICENSOR
ADDRESS	ADDRESS
Signature	Signature
Name	Name
Title	Title