REPORT DOCUMENTATION PAGE

Form Approved
OMB NO. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	August 1996	Final Report 4/1/95 - 8/30/96

4. TITLE AND SUBTITLE
A Task Networking and Visual Programming Language for Jack

5. FUNDING NUMBERS
DAAH04-95-1-0151

6. AUTHOR(S)
Dr. Norman I. Badler

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
University of Pennsylvania
Computer & Information Science Department
Center for Human Modeling & Simulation
Phila., PA 19104-6389

8. PERFORMING ORGANIZATION REPORT NUMBER
DAAH04-94-G-0221

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
U. S. Army Research Office
P.O. Box 12211
Research Triangle Park, NC 27709-2211

10. SPONSORING / MONITORING AGENCY REPORT NUMBER
34551. 1-MA

11. SUPPLEMENTARY NOTES
The views, opinions and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of the Army position, policy or decision, unless so designated by other documentation.

12 a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12 b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

This report includes how VisualJack was designed as a graphical user interface (GUI) to Lisp PaT-Nets (Parallel Transition Networks). The VisualJack conception was that a user could use the GUI to construct PaT-Nets interactively, see the relationships among the various parts of a PaT-Net program, and thus speed the task of PaT-Net definition. Essentially finite automata, Parallel Transition Networks execute in parallel in the Jack environment.x (See the Appendix for a guide to LISP PaT-Nets.) Together with the Jack LISP API, they form an intuitive interface to controlling simulation and behavior of processes and agents in Jack.

14. SUBJECT TERMS
Computer Graphics

15. NUMBER OF PAGES
18

16. PRICE CODE

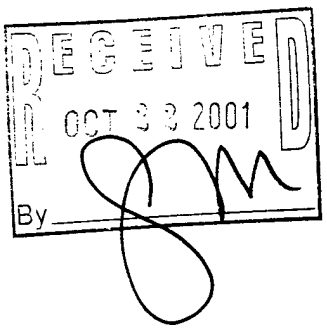
17. SECURITY CLASSIFICATION OR REPORT	18. SECURITY CLASSIFICATION ON THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Center for Human Modeling & Simulation
University of Pennsylvania
Philadelphia, PA 19104-6389
(215) 898-1488

A Task Networking and Visual Programming Language for Jack

FINAL REPORT
DAAH04-94-G-0151
August 1996

Norman I. Badler
badler@central.cis.upenn.edu



MASTER COPY: PLEASE KEEP THIS "MEMORANDUM OF TRANSMITTAL" BLANK FOR REPRODUCTION PURPOSES. WHEN REPORTS ARE GENERATED UNDER THE ARO SPONSORSHIP, FORWARD A COMPLETED COPY OF THIS FORM WITH EACH REPORT SHIPMENT TO THE ARO. THIS WILL ASSURE PROPER IDENTIFICATION. NOT TO BE USED FOR INTERIM PROGRESS REPORTS; SEE PAGE 2 FOR INTERIM PROGRESS REPORT INSTRUCTIONS.

MEMORANDUM OF TRANSMITTAL

U.S. Army Research Office
ATTN: AMSRL-RO-RI (Hall)
P.O. Box 12211
Research Triangle Park, NC 27709-2211

☐ Reprint (Orig + 2 copies)
☐ Technical Report (Orig + 2 copies)
☐ Manuscript (1 copy)
☒ Final Progress Report (Orig + 2 copies)
☐ Related Materials, Abstracts, Theses (1 copy)

CONTRACT/GRANT NUMBER: DAAH04-94-G-0151

REPORT TITLE: A Task Networking and Visual Programming Language for Jack

is forwarded for your information.

SUBMITTED FOR PUBLICATION TO (applicable only if report is manuscript):

Sincerely,
Norman I. Badler
Department of Computer & Information Science
University of Pennsylvania
Philadelphia, PA 19104-6389

A Task Networking and Visual Programming Language for Jack

Norman I. Badler, Principal Investigator

Prepared by: Brett Douville

Introduction

VisualJack was designed as a graphical user interface (GUI) to Lisp PaT-Nets (Parallel Transition Networks). The VisualJack conception was that a user could use the GUI to construct PaT-Nets interactively, see the relationships among the various parts of a PaT-Net program, and thus speed the task of PaT-Net definition.

Essentially finite automata, Parallel Transition Networks execute in parallel in the Jack environment.x (See the Appendix for a guide to LISP PaT-Nets.) Together with the Jack LISP API, they form an intuitive interface to controlling simulation and behavior of processes and agents in Jack.

The node is the basic building block of the PaT-Net . There are several types of nodes, but each has a similar structure and behavior. Each node has an associated action, and transitions between nodes determine the path through the graph. Transitions can be randomly assigned, weighted with probability, or given as a set of ordered conditions from which the first valid condition will be selected. Conditions and actions can manipulate a set of local variables. A set of monitors adds control within the PaT-Net. VisualJack allows the user to create nodes and add transitions interactively. Lisp codes to be executed in either nodes or transitions may be typed directly into the graphical objects via the GUI.

The VisualJack GUI

VisualJack has a simple window manager, with the look and feel of Microsoft Windows95's "multiple document interface." To demonstrate, select "File->New->PaT-Net" from the menubar with the left mouse button. You will note a window in the main VisualJack window which can be manipulated much like a Windows95 window. Also notice that when you move around to different selections in the "File" menu, the friendly help bar ("QuickHelp") at the bottom of the main window changes its text to give you a helpful hint about the function of that menu selection. (This will crop up at all times while you're using VisualJack -- it's a very helpful reminder of what various features do!)

Select the title bar of the window you just created (its name is "Untitled PaT-Net 1" by default -- we'll get to how to rename this later). You can drag and drop the window anywhere on the canvas just as in most window managers. As in Windows95, this window can also be maximized, minimized, and closed.

When you minimize a window using the left mouse button on either the Window Control Menu (under the button in the titlebar with the little "VJ" in it) or the button on the right hand side of the title bar with the underbar ("_") in it, it gets turned into a button which appears in the bottom of the VisualJack main window. Try minimizing the window both ways and restoring it by clicking on the button that appears in the lower left-hand corner of the VisualJack window.

You can also maximize and restore a window, using either the menubar or the button with the little picture of a window at the right hand side of the title bar. The button and menubar automatically reconfigure based on the current state of the window -- when you maximize, the maximize button becomes a restore button and vice versa, and the menu bars change also.

Finally, you can also close the window. Closing the window can be done from either the "X" button at the right hand side of the title bar, or by using the "Close" selection of the Window Control Menu or the main File menu.

Interacting with a PaT-Net Window

VisualJack features a context sensitive menu that is always available under the right mouse button (RMB). When you use the RMB inside of a VisualJack window, it gives a list of operations, which are appropriate to the window. For example, if you click the RMB in the "Untitled PaT-Net 1" canvas (which is purple), a pop-up menu appears which indicates that you can currently "Create new node" in that window.

When you select "Create new node", you will find that the QuickHelp in the bottom of the main window of VisualJack changes to read "Click the left mouse button to place a new node; hit the middle mouse button to cancel." First, try aborting the new node creation by clicking the middle mouse button (MMB) anywhere inside of the purple area.

Next, create an actual node, by repeating the above step but using the LMB instead of the MMB inside the purple region.

Now that you've created a node, you have more options on the pop-up menu. Try using the RMB again, and you'll see that you now have the following selections:

Create new node
Edit node
Create arc
Delete node

Select "Delete node" from the pop-up menu. Note that the cursor will change to a skull and crossbones, which you should place over the node you've just created and choose the LMB (again, selecting the MMB will cancel this action).

Once you've selected the node to be deleted, you'll see a dialog box which asks whether or not you really want to delete the node, telling you that all edges leading in and out of the node will also be deleted. Select "cancel" and then repeat the whole process and select "OK".

You'll see that the node gets removed from the display. Furthermore, the pop-up menu will have changed again, so that now, the only action you're able to take is "Create new node" again.

Manipulating Nodes and Actions

Create another new node by using the pop-up menu and placing the node wherever you'd like. First, let's look at the action of that node by selecting the button that says "Action1". There's a lot of information about an action that you can fill in -- although most of this information isn't necessary for the actual running of a PaT-Net, it will be helpful for future versions of VisualJack to do things like debugging, etc.

Note that the title of the dialog window is the same as the title in the entry under "Title". If you change this name, you'll also change the name of the dialog.

You can also change the name of the action, annotate it (to help explain what it's supposed to do, for example) and change the lisp name of the action. Default names are given for all of these, except the annotation. By clicking the "Edit" button, you can edit the lisp code of the function, using a separate editor that pops up when the button is clicked. Selecting the "Cancel" button revokes the changes so that the original settings are restored, and selecting "OK" makes the changes.

You can also edit a node by selecting the "Edit node" entry on the pop-up menu. When you do so, you will see several bits of information about the node, much like the action editor. Furthermore, you can set the action of the node to be an existing action in the PaT-Net (such as ":no-op", which exists in every PaT-Net by default) by clicking the left mouse button on the entry window containing the action name. This pops up a list box from which one of several existing actions can be selected. Simply select one of these to change the action (or keep the action) of the node. You can also create a new action for the node by selecting the "New" button. This will allow you to set the name of the new action.

Finally, you can choose to change the type of a node, for example, from a "Normal" node (from which arcs are taken based on the values of conditions) to a "Probabilistic" node (from which arcs are taken based on the probabilities of taking various arcs).

Of course, any changes you make can be cancelled with the "Cancel" button, and they are made when you select the "OK" button.

Creating Arcs

Once you've created a node, you have the option of creating arcs (you need only create a single node, since arcs can have the same beginning and ending). To create an arc, simply select "Create arc". QuickHelp will now tell you to select the start node of the arc using the left mouse button and having done so, will then prompt for the end node for the arc. To create an arc beginning and ending at the same node, simply select the same node twice.

Try creating nodes and arcs for a while -- in particular, note that arcs are laid out so that they don't overlap with one another when more than one arc goes between the start and end nodes.

Arcs are laid down with a simple two-line diagram because of speed tradeoffs. Tcl scripts are unfortunately very slow in drawing many objects (they do not take advantage of specialized hardware such as is found on the SGI) so a more rounded curve is simply impractical when many arcs occupy a scene. In future versions of VisualJack, other options could be:

(1) Implementing in a language with lower level control of the hardware such as C (or C++)

(2) Changing level of detail on the arcs when interactive drawing rates are required.

In any case, having created several arcs, you can now manipulate the conditions and arc information using the "Modify arc" command on the pop-up menu.

APPENDIX: LISP PaT-Net Guide

Appendix: PaT-Net User's Guide

PaT-Net User's Guide

Brett J. Douville

August 4, 1995

1 Introduction

Essentially finite automata, **P**arallel **T**ransition **Net**works (*PaT-Nets*) execute in parallel in the *Jack* environment. Together with the *Jack* LISP api, they form an intuitive interface to controlling simulation and behavior of processes and agents in *Jack*.

The *node* is the basic building block of the PaT-Net. There are several types of nodes (see Section 7), but each has a similar structure and behavior. Each node has an associated *action*, and *transitions* between nodes determine the path through the graph. Transitions can be randomly assigned, weighted with probability, or given as a set of ordered conditions from which the first valid condition will be selected. Conditions and actions can manipulate a set of local variables. A set of *monitors* adds control within the PaT-Net.

This framework provides a powerful yet intuitive machinery for simulation control. This document describes in detail all important features of the PaT-Net formalism; the addenda provide several contextual details.

2 PaT-Net Declaration

Before any actions, conditions, nodes, or anything else about a PaT-Net can be declared, the PaT-Net itself has to be declared. This is just a simple line which alerts the LISP interpreter to the existence of a PaT-Net having a particular name and a particular set of *local variables* (as described in Section 3 below).

2.1 Declaring the PaT-Net

The LISP syntax for declaring a PaT-Net is:

`(deffsm <pname> '(<lvar1> <lvar2> ... <lvarn>))`

where

- `<pname>` is the name of the PaT-Net.

- <lvar$_i$> is the name of the ith variable local to the PaT-Net. These will be the only variables local to the PaT-Net.

Appendix B describes a declaration macro, `defnet`, in detail. `Defnet` is useful only for small PaT-Nets consisting of between ten and twenty nodes. Essentially, the macro bundles action, condition, node, and monitor assignments into a simpler form that doesn't require the PaT-Net name to be included in every assignment. For more details, see Appendix B.

2.2 Example

The list of local variables can be as simple or as lengthy as desired. Here's a simple example:

```
(deffsm grab_tool_net '(human hand tool))
```

A more complicated example would include a longer list of variables, such as:

```
(deffsm thumb_net '(humanptr handstr jptr0 jptr1 jptr2 seg0
  seg1 seg2 fin0 fin1 fin2 fin3 goal_x goal_z object seg_list
  start end time motionflag opp))
```

2.3 A Tip for Declaring PaT-Nets

When writing PaT-Nets, it's helpful to describe precisely the purpose of each local variable. That way, when a user attempts to use your PaT-Net, there is a clear specification (if not an overly formal one) of what exactly your PaT-Net manipulates.

For example, the simple declaration above could be modified as follows:

```
(deffsm thumb_net
  '(humanptr      ;; Pointer to human whose thumb is moved
    handstr       ;; Which hand ("right" or "left")
    jptr0         ;; Joint pointers
    jptr1
    jptr2

    seg0          ;; Strings for the names of the segments
    seg1          ;; in the thumb
    seg2

    fin0          ;; Lists of strings, three strings each
    fin1          ;; containing the names of the segments in
    fin2          ;; the fingers -- sans human-name
    fin3

    goal_x        ;; Goal angles for the thumb0 joint
    goal_z        ;; Will default to hand geometry limits
```

```
    object          ;; Grasped object
    seg_list        ;; Possibly colliding segments
                    ;; (less object name)
    start           ;; Start time
    end             ;; End time
    time            ;; Explicit representation of time
    motionflag      ;; Use motion system for time?

    opp             ;; Defaults to "palmar".
                    ;; May be one of "palmar", "adducted", "pad".
                    ;; Refers to opposition space info.
))
```

This specification clarifies the above example, and this style tends to be generally beneficial in more complicated examples.

3 Local Variables

Each PaT-Net may have several local variables, variables which are local to each *instantiation* (see Section 10, below). These local variables must be declared when the PaT-Net is declared (see Section 2 above).

Accessing Local Variables

Local variables of a PaT-Net may be accessed and manipulated solely through the `val` function. The syntax for the `val` function is as follows:

```
(val <label> <?newval>)
```

where **<label>** is the name of the local variable being accessed, and **<?newval>** is an optional parameter which, if present, becomes the new value of the local variable. The function `val` returns the value of the local variable after the function call; if the function changes the value of the variable, it returns the new value.

4 PaT-Net Initialization

Each PaT-Net executes an *initialization phase* when instantiated by a message (see Section 10 below). When the PaT-Net is first instantiated, the arbitrary LISP code in the body of the initialization is evaluated; this code is not evaluated at any other time and only one initialization phase may be defined for a particular PaT-Net. (Actually, more than one initialization phase may be defined, but only the last one bound would be executed when the PaT-Net is instantiated.)

4.1 Defining the Initialization Phase

The LISP syntax for the initialization routine of a PaT-Net is as follows:

```
(definit <pname> <iargs> <ibody>)
```

where

- pname is the name of the PaT-Net for which the initialization is defined.
- args is a list of arguments to the PaT-Net. This is a LISP argument string, and may contain, for example, optional and keyword arguments.
- body is the body of the PaT-Net initialization.

Initialization occurs immediately after the PaT-Net is instantiated see Section 10 below).

4.2 An Example

In the following example, a PaT-Net with local variables (humanptr handstr toolptr) is instantiated with a string describing a human in the *Jack* environment, a string describing a tool in the *Jack* environment, and a string telling with which hand to reach for the tool.

```
(definit reach_net (humanstr toolstr &key (handstr ''right''))
  ;; Assign the local variable humanptr to the pointer to the human figure.
  (val humanptr (figure-find humanstr))

  ;; Assign the handstr to the argument handstr, which must be either
  ;; ''right'' or ''left''.
  (val handstr handstr)

  ;; Assign the local variable toolptr to the pointer to the tool
  ;; described by the string toolstr.
  (val toolptr (figure-find toolstr))

  ;; Print a message to the calling window.
  (format t ''Initialization complete ~%''))
```

5 PaT-Net Conditions

A *condition* is an arbitrary LISP expression that evaluates to *nil* or non-*nil*. Typically, conditions are tested to determine what transition to take from a node to a subsequent node (see Section 12 below).

Rather than being directly connected to any particular transition, conditions are bound to a PaT-Net, and can be called as a message. They can be combined arbitrarily in this way to make new conditions; similarly, if they have side effects, they can be used as parts of actions (see Section 9).

5.1 Defining Conditions

The LISP syntax for defining a condition is:

```
(defcond <pname> :<cname> <cbody>)
```

where

- `<pname>` is the name of the PaT-Net in which the condition is defined.
- `<cname>` is the name of the condition.
- `<cbody>` is the condition body. The body of the condition is evaluated when the condition is called.

A condition evaluates to the return value of the last evaluated expression in the condition body.

Predefined Conditions

The `:default` condition is predefined in every PaT-Net. Evaluating `:default` always returns non-*nil*.

5.2 Examples

As LISP expressions, conditions can be arbitrarily complex and take full advantage of the expressive power of LISP. Some of the following examples use local variables, which are described in Section 3. All of the PaT-Nets have the same name: "`simnet`".

- This example checks the condition of some global variables.

```
(defcond simnet :c_threshold
  ;; Checks whether either global variable <x1> or global variable <x2>
  ;; has exceeded the global variable <tolerance>.
  (or (> x1 tolerance)
      (> x2 tolerance)))
```

- An example of a condition which tests to see whether a certain time has elapsed in *Jack*'s motion system. (The function `(motion-time)` is a *Jack* LISP api call which returns a floating point value of the current simulation time.)

```
(defcond simnet :c_test_time
  ;; Returns false if the motion system hasn't yet reached time <time>.
  (< (motion-time) (val time)))
```

- This condition is again a tolerance measure, but it instead checks a robot's current fatigue against its expected fatigue tolerance and its current heat against its threshold heat. (The functions `fatigue`, `fatigue_threshold`, `sys_temp` and `heat_tolerance` would have to be written – they are given as examples; their names suggest their results.)

```
(defcond simnet :c_breakdown
  ;; Tests whether robot breakdown is imminent by looking at fatigue
  ;; (measured by energy levels) and current system temperature.
  (or
    (> (fatigue (val robot)) (fatigue_threshold (val robot)))
    (> (sys_temp (val robot)) (heat_tolerance (val robot)))))
```

5.3 Tips for Writing Conditions

- Name your conditions mnemonically, so that you have some idea of what it means at a glance. This will also make it easier for another programmer to understand your code (especially node transitions) and it might make it easier to generate diagrams of your PaT-Nets.

- You might also want to preface your conditions with the letter 'c' if you plan to use them for any side effects they might cause. This will make it clear that the method is a condition, but that it might be used out of context. Also, it makes transitions a little more clear (sort of like putting a 'p' before pointers when writing *C* code).

- If you find yourself repeating a lot of code in your conditions, *i.e.* anding several conditions together as another condition, try to take advantage of PaT-Net communication (Section 9, below). The following shows two sets of conditions, one with repeated code, and one with conditions called by PaT-Net communication:

```
(defcond simnet :c_fatigue
  ;; Tests to see whether the robot has drained its batteries.
  (> (fatigue (val robot)) (fatigue_threshold (val robot))))

(defcond simnet :c_overheat
  ;; Tests to see whether the robot is in danger of overheating.
  (> (sys_temp (val robot)) (heat_tolerance (val robot))))

(defcond simnet :c_breakdown
  ;; Tests whether robot breakdown is imminent by looking at fatigue
  ;; (measured by energy levels) and current system temperature.
  (or
    (> (fatigue (val robot)) (fatigue_threshold (val robot)))
    (> (sys_temp (val robot)) (heat_tolerance (val robot)))))
```

Compare the rewriting of code in `:c_breakdown` with the following, which is much clearer:

```
(defcond simnet :c_breakdown
  ;; Tests for imminent robot breakdown by looking at fatigue
  ;; (measured by energy levels) and current system temperature.
  (or
    (send self :c_fatigue)
    (send self :c_overheat)))
```

This also has the virtue of reminding you what other conditions you are testing in the PaT-Net.

6 PaT-Net Actions

An action is an arbitrary LISP expression. Though typically an action modifies something in the *Jack* environment, it can compute an arbitrary function, for example. However, since we are concerned primarily with the use of PaT-Nets for simulation purposes, we will focus on actions which have meaning in the *Jack* simulation environment.

As with conditions (Section 5), actions are not associated directly with any particular node; rather, actions are defined for a PaT-Net and are called by nodes when entered. They can also be called independently from individual nodes, using PaT-Net communication; an example appears in Section 6.2.

6.1 Defining Actions

The LISP syntax for defining a condition is:

`(defaction <pname> :<aname> <abody>)`

where

- `<pname>` is the name of the PaT-Net in which the action is defined.
- `<aname>` is the name of the action.
- `<abody>` is the action body. The body of the action is evaluated when the action is executed.

Similarly to a condition, an action evaluates to the return value of the last evaluated expression in the action body.

Predefined Actions

There is a single predefined action in every PaT-Net, the `:no-op` action. `:no-op` means "no operation" – this is a null action which simply returns non-*nil*.

6.2 Examples

Because they are LISP expressions, conditions can be arbitrarily complex and can take full advantage of the expressive power of LISP. However, we will usually think of actions as having some impact on a simulation, whether directly (such as adjusting a joint angle) or indirectly (such as instantiating another PaT-Net which will have some effect on the simulation). Here are a few simple actions (some of which use functions that are undefined). We'll continue using our robot controller example.

- Here's an action that turns the robot to the left a certain arc. (Several of these functions are bound in the *Jack* LISP api. Assume that `*frame-rate*` is a global constant describing the *Jack* simulation rate.)

```
(defaction simnet :turn_left
  ;; Turns a robot to the left a certain number of degrees dependent on the
  ;; frame rate and the turning velocity (angular velocity in degrees/sec)
  ;; of the wheels.
  (rawjcl (format nil ''move_figure(%s,%a);''
          (val robot)
          (matrix-tojcl
           (matrix-mult
            (figure-transform (figure-find (val robot)))
            (matrix-rot (* *frame-rate* (val turn-velocity)))))))))
```

- In this example, an avoidance is bound to a wall in the environment. This indirectly affects the simulation because the behavioral simulation system (*BSS*), which provides some low-level control of human locomotion through attraction and avoidance, will change the path of the human to avoid the wall.

```
(defaction simnet :bind_wall_avoid
  ;; Binds an avoidance to a wall, whose pointer is currently in the local
  ;; variable 'wall'.  The function (bind-avoidance) takes two arguments, a
  ;; human and a figure, and the human avoids the figure (through BSS).
  (bind-avoidance (figure-find (val human))
                  (val wall)))
```

- In this example, the action spawns another PaT-Net. Again, this doesn't directly affect the simulation, but instead creates another PaT-Net which may itself directly (or indirectly) affect the simulation.

```
(defaction simnet :start_simnet2
  ;; Starts a different PaT-Net with the argument of the current robot.
  (send simnet2 :new (val robot)))
```

6.3 Tips for Writing Actions

- Name your actions mnemonically, so that you have some idea of what it means at a glance. This will also make it easier for other programmers to understand your code at a higher level just by looking at the nodes and transitions; it might make it easier to diagram your PaT-Nets as well.
- Prefacing actions with the letter 'a' is helpful if you intend to use them as parts of conditions.
- Actions can be called similarly to conditions through messages; for more detail, see Section 9.

7 PaT-Net Nodes

Nodes form the heart of PaT-Nets, as they are the structure on which actions and conditions are hung. In the following, the *follow node* refers to the node to which a transition is made. There are several types of PaT-Net nodes, each of which will be described in detail below:

Simple Node The type of node one normally associates with finite state machines, this has a number of transitions whose conditions are tested before moving to another node.

Probabilistic Nodes This node has a number of transitions associated with it, but instead of having particular conditions associated with each node, probabilities summing to a value less than or equal to `1.0` are associated with transitions. When selecting a transition, the node will select the follow node probabilistically.

Weighted Nodes Weighted nodes are like probabilistic nodes, except that the probability values are scaled to sum to `1.0`.

Random Nodes Random nodes are simply probabilistic nodes where every node has equal probability; these sum to `1.0`. The follow node is selected randomly from among a list of nodes.

7.1 Defining Simple Nodes

The LISP syntax for defining a simple node is:

```
(defnode <pname> <nname> :<aname>
(:<c1> <n1>) ;; Transition on c1 to n1
(:<c2> <n2>)...)  ;; Transition on c2 to n2 etc.
```

where

- `<pname>` is the name of the PaT-Net in which the condition is defined.
- `<nname>` is the name of the node which you are defining. (You must name nodes so that transitions between nodes can be bound.)

- <aname> is the name of the action to execute upon entering the node (see Section 12 below).
- <c_n> is the name of the condition associated with a particular transition. These conditions will be evaluated in order until one of them returns non-*nil*, at which point there will be a transition to the follow node.
- <n_n> is the name of the follow node if the condition evaluates to non-*nil*.

7.2 Defining Probabilistic Nodes

The LISP syntax for defining a probabilistic node is:

```
(def-probnode <pname> <nname> :<aname>
(<p1> <n1>) ;; Probability of transition to n1
(<p2> <n2>) ...)  ;; Probability of transition to n2 etc.
```

where

- <pname> is the name of the PaT-Net in which the node appears.
- <nname> is the name of the node.
- <aname> is the name of the action to execute upon entering the node.
- <p_n> is the probability associated with a particular transition. The follow node will be determined probabilistically, and with probability <p_n> will transition a particular node.
- <n_n>, which is the name of the follow node with probability <p_n>.

A probabilistic node is only meaningful if the probabilities sum to less than or equal to 1.0. If the sum of the probabilities $P \leq 1.0$, then no transition will be taken (see Section 12).

7.3 Defining Weighted Nodes

A weighted node is a special case of a probabilistic node, where probabilites are assigned according to the weights on the nodes and scaled to sum to 1.0. The LISP syntax for defining a weighted node is:

```
(def-weightnode <pname> <nname> :<aname>
(<w1> <n1>) ;; Transition to n1 with weight w1
(<w2> <n2>)...)  ;; Transition to n2 with weight w2 etc.
```

where

- <pname> is the name of the PaT-Net in which the node is defined.

- <nname> is the name of the node.
- <aname> is the name of the action to execute upon entering the node.
- <w_n> is the weighted probability to assign to a particular transition.
- <n_n> is the name of the node to take with weighted probability <w_n>.

A transition from a weighted node will always be taken (it is impossible to remain in the state, though an explicit translation to the state is possible).

7.4 Defining Random Nodes

A random node is a special case of a weighted probabilistic node in which all states have equal weight. The LISP syntax for defining a random node is:

```
(def-randnode <pname> <nname> :<aname>
n1 n2 ... nn) ;; Possible transitions
```

where

- <pname> is the name of the PaT-Net in which the node is defined.
- <nname> is the name of the node.
- <aname> is the name of the action associated with the node.
- <n_n> are the names of potential follow nodes among which a random selection is made. The probability of selecting any particular follow node is $\frac{1}{k}$, where k is the number of nodes.

Predefined Nodes

There is one predefined node in every PaT-Net, the `exit` node. Making a transition to the exit node removes the PaT-Net from the list of active PaT-Nets. (The list of active PaT-Nets is described in Section 12.)

7.5 Examples

Here are examples of each type of node:

- This node in a robot controller determines whether to back up, continue straight, or turn on the basis of some primitive sensation. The robot continues in its current direction if there is no obstacle directly in front of it, backs up if it's boxed in, and goes to a turning node otherwise.

```
(defnode simnet locomote :move_forward
  (:forward_clear  locomote)
  (:boxed_in       back_up)
  (:default        turn))
```

- This node in the same robot controller decides among turns probabilistically. It will go left with probability .2, right with probability .2, and straight with probability .4. With probability .2 the robot will stay in this node (which is different from returning to this node, see Section 12) and do nothing. In the subsequent simulation step, a second probabilistic evaluation will occur.

```
(def-probnode simnet possibly_turn :move_forward
  (0.2  turn_left)
  (0.2  turn_right)
  (0.4  possibly_turn))
```

- Here is a weighted probability version of the previous node. Note that the PaT-Net can't remain in this node in this version. The values associated with the transitions will scale to `0.25`, `0.25`, and `0.5` respectively.

```
(def-weightnode simnet possibly_turn2 :move_forward
  (0.2 turn_left)
  (0.2 turn_right)
  (0.4 possibly_turn))
```

- Finally, here's a version of the same node using a random node. Again, there is no possibility of remaining in the same node without making a transition.

```
(def-randnode simnet possibly_turn3 :move_forward
  turn_left turn_right possibly_turn)
```

7.6 The Initial Node

The first node defined in a PaT-Net is the *initial node.* This will be the node which is entered immediately after the initialization phase of the PaT-Net (see Section 4).

With the exception of the initial node, the order in which nodes are defined is unimportant.

7.7 Tips for Writing Nodes

- As with conditions and actions, it is often helpful to name your nodes mnemonically. Names like `node1` and `state12`, while obvious choices, are not very illustrative of information about the state.

- Take care when listing your transitions in a simple node. Remember that they will be tested *in order* rather than non-deterministically.

8 Monitors

A *monitor* is a device which exists outside of the node structure of a PaT-Net. As its name suggests, it checks whether a specific condition is non-*nil* of the PaT-Net at every simulation step, and if it is, executes an action. Only one monitor executes per clock tick; the first whose condition is true.

Monitors are useful for looking at special cases, updating local variables (such as an explicit time variable or a counter), and other actions which might be useful to execute every frame. They can be helpful in debugging, too, by executing an action which prints the values of all local variables (or tracking one variable in particular) after every frame.

8.1 Defining Monitors

The LISP syntax for defining a monitor is:

```
(defmonitor <pname> <mname> :<cname> :<aname>)
```

where

- `<pname>` is the name of the PaT-Net in which the monitor resides.
- `<mname>` is the name of the monitor.
- `<cname>` is the name of a condition associated with the PaT-Net. This condition will be evaluated every simulation frame; if non-*nil*, the action below will be executed.
- `<aname>` is the name of an action associated with the PaT-Net. This action will only be executed in frames when the condition denoted by `<cname>` is true.

8.2 Example

This monitor increments a variable representing simulation time on every simulation step. Because of the nature of monitors, the action associated with the monitor is provided; the condition is predefined.

```
(defaction simnet :increment_time
  (val time (+ (val time) *frame-rate*)))

(defmonitor simnet inc_time :default :increment_time)
```

8.3 Tips for Writing Monitors

- Use monitors sparingly; if you find yourself requiring a lot of monitors, you may want to reconsider the design of your PaT-Net, either through refining your actions or by taking advantage of PaT-Net parallelism and having several PaT-Nets execute at once to simulate your process.

- Monitors can also benefit from using mnemonic names. Consider the readability of the example above against the following:

```
(defmonitor simnet mon :default :mon_action)
```

- Remember that at most one monitor will be executed per simulation step (see Section 12 below); keep this in mind when designing monitors.

9 PaT-Net Communication

PaT-Nets communicate by *messages*. A message to a PaT-Net class can instantiate a new class (see Section 10, below), signal a wait (see Section 11, below), and execute actions and conditions by name (for an example, see Section 5.3 above).

9.1 Sending Messages

Several different messages can be sent to PaT-Nets; however, these all follow a common LISP syntax, given below.

```
(send <name> :<message> <args>)
```

where

- `<name>` is the name of the object being sent a message (either a PaT-Net or a PaT-Net class in the case of instantiation, see Section 10 below).
- `<message>` is the message being sent. Messages are always preceded by a colon.
- `<args>` are any arguments required by the message. Arguments are most commonly seen when instantiating a new PaT-Net (any arguments to the initialization of the PaT-Net must be sent), or when setting *waits*, which are discussed in detail in Section 11 below.

9.2 Message Evaluation

Message evaluation returns different values depending on the message. The following is a list messages and their return values:

- Evaluating a `:new` message to a PaT-Net class results in a new PaT-Net instance. This result can be used subsequently for the sending of other messages, for example:

```
(setf simnet1 (send simnet :new ''jack'' ''hammer'' ''left''))

(send simnet1 :c_threshold)
```

- A special message receiver (`self`) always refers to the object from which the message is sent. For example, in Section 5.2 above, we defined a condition in which two messages were passed to create a disjunction of two existing conditions. This example is repeated below using messages:

```
(defcond simnet :c_breakdown
  ;; Tests whether robot breakdown is imminent by looking at fatigue
  ;; (measured by energy levels) and current system temperature.
  (or
    (send self :c_fatigue)
    (send self :c_overheat)))
```

- Evaluating a `:<condition>` message to a PaT-Net results in the evaluation of the condition *within* that PaT-Net. For example, in the above, the `(send simnet :c_threshold)` command would return the same result as evaluating `:c_threshold` within the PaT-Net containing the condition.

- Section 11 provides a full discussion of *waits*.

10 PaT-Net Instances

A specification of a PaT-Net differs from an *instantiation* (which I will call an *instance*) of a PaT-Net; the specification of a PaT-Net provides a template which derives individual instances. This important distinction separates the PaT-Net class from the actual PaT-Net which is instantiated via a message.

Sending a `:new` message (along with any arguments required to initialize the PaT-Net) instantiates a PaT-Net of the type being sent the message. The evaluation of such a message returns the PaT-Net instance thereby created. For example, evaluating

```
(send simnet :new ''jack'' ''hammer'')
```

will instantiate the PaT-Net whose initialization phase is described in Section 4 above.

11 Waits

An action, condition, or initialization can post a set of *waits*; the PaT-Net will block until the wait is no longer active. The following sections, and the description of execution (Section 12) should also be helpful. There are several types of waits, each described in detail below:

Wait for Another PaT-Net This allows a PaT-Net to wait until another PaT-Net exits before continuing.

Wait for a Specific Time This allows a PaT-Net to wait for a specific time in *Jack*'s motion system before proceeding.

Wait for a Specific Condition This allows a PaT-Net to wait until a specific condition becomes true before continuing.

Wait for a Motion to Complete The PaT-Net will wait for a motion in *Jack*'s motion system to finish before continuing.

Wait for a Semaphore This allows a PaT-Net to wait based on a operating system-style semaphore with a particular priority. A full description of our implementation of semaphores is given in Appendix C.

Each of these waits will be described in detail below.

11.1 Waiting for Another PaT-Net

Currently, the PaT-Net formalism only supports waits for another PaT-Net to exit (that is, pass through its *exit* node). This functionality will be extended so that the wait can be used to block the PaT-Net until another PaT-Net goes through an arbitrary node. The LISP syntax for this extension will be:

```
(send <patnet> :wait-fsm <patnet-instance> ?<:state statename>)
```

where

- `<patnet>` is the PaT-Net which is going to wait. This argument is most commonly filled with `self`, however, a wait can be signalled for any PaT-Net in this manner. I refer to this as the *waiting* PaT-Net.
- `<patnet-instance>` is the PaT-Net for which to wait, which I refer to as the *wait-for* PaT-Net.
- `?<:state statename>` refers to the optional keyword argument giving a state through which the *wait-for* PaT-Net must pass before the *waiting* PaT-Net can continue. For an example, see Section 11.6 below.

11.2 Waiting for a Specific Time

Often when a PaT-Net is running in lock step with the *Jack* motion system, waiting for a specific time in the motion system can be helpful. (For example, if you're simulating a teapot on a stove, then you might want steam to come out of it after three minutes on high heat.) The LISP syntax for defining a wait of this type is:

```
(send <patnet> :wait-time <time>)
```

where

- `<patnet>` is the name of the *waiting* PaT-Net.
- `<time>` is the time in the motion system until which to wait.

11.3 Waiting for a Specific Condition

PaT-Nets can be made to wait for an arbitrary condition to be true within the PaT-Net; such conditions can therefore refer to local variables using the `val` macro (see Section 3 above). The conditions must be expressed in terms of functions which take no arguments, usually a zero-argument `lambda` function. The LISP syntax for describing such a wait is:

`(send <patnet> :wait-condition <condition-fun>)`

where

- `<patnet>` is a PaT-Net instance or `self`.
- `<condition-fun>` is a LISP function closure which takes no arguments.

11.4 Waiting for a Motion to Complete

A PaT-Net can wait for a particular motion in *Jack*'s motion system to complete before continuing. To do so, the PaT-Net must be provided with a pointer to the motion in the animation system, which can be found using the *Jack* LISP api call `(motion-find <str>)`, where `<str>` is the name of the motion in *Jack* (a string). The LISP syntax for such a wait is:

`(send <patnet> :wait-motion <motion-pointer>)`

where

- `<patnet>` is a PaT-Net instance or `self`.
- `<motion-pointer>` is a pointer to a motion in the *Jack* motion system.

For more information about *Jack*'s motion system, see the *Jack* User's Guide and the *Jack* LISP api.

11.5 Waiting for a Semaphore

The LISP version of semaphores is analogous to the operating system concept of a semaphore, and are described in more detail in Appendix C. Waiting on a semaphore means being put on the semaphore's *wait queue* in order of priority. The LISP code for waiting on a semaphore is as follows:

`(send <patnet> :wait-semaphore <semaphore> ?<:priority priority>)`

where

- `<patnet>` is a PaT-Net instance or `self`.
- `<semaphore>` is an instance of a semaphore.
- `?<:priority priority>` is an optional assignment of the *priority* the PaT-Net has on the semaphore's wait queue. This defaults to `1.0`.

11.6 Examples

Included below are examples for several types of waits.

- In the following example, the action `spawn_simnet2` of `simnet` instantiates a new PaT-Net of type `simnet2`. Any instance of `simnet` executing this action will wait until the new instantiation of `simnet2` exits before continuing.

```
(defaction simnet :spawn_simnet2
  ;; Spawn another PaT-Net and wait for it to exit.
  (send self :wait-fsm (send simnet2 :new (val human) (val tool))))
```

- The following example illustrates the use of a wait for a specific time in the motion system:

```
(defaction simnet :wait_until_time
  ;; The variable time has been determined by other means, and
  ;; now the PaT-Net must wait until that time to continue.
  (send self :wait-time (val time)))
```

- In the following example, assume the function `attached-fig` returns non-*nil* if its first argument is attached to its second argument. The PaT-Net waits until this condition returns non-*nil*.

```
(defaction simnet :wait_for_attach
  ;; simnet waits until the tool has been attached to the human.
  (send self :wait-condition
        #'(lambda () (attached-fig (val tool) (val human)))))
```

- The following provides an example of waiting for a specific motion (in *Jack*'s motion system) to complete:

```
(defaction simnet :wait_on_arm
  ;; Simnet waits until the arm motion completes.
  (send self :wait-motion (motion-find (val motion_name))))
```

- The extended example in Appendix A gives a non-trivial example of a semaphore wait.

12 Execution View of a PaT-Net

PaT-Nets execute in the context of the *Jack* 3D graphics modeling and simulation package developed at the University of Pennsylvania. Briefly, transitions between nodes and action execution take place at the rate (commonly referred to as the *frame-rate*) at which the simulation updates. In real-time simulation, the frame-rate is $\frac{1}{30}$ seconds, though in practice this rate is much slower, perhaps between .5 and 5 seconds depending on the complexity of the simulation. For now, we will treat the frame-rate as a fixed unit of time independent of any particular simulation and refer to its units as clock *ticks*.

When instantiated, a PaT-Net is placed on the *active* list, a list of active PaT-Nets in the current simulation environment. This list is maintained for the duration of a *Jack* session; it may be cleared using the `(reset-fsm)` command described in Appendix D.

This section will describe in detail the execution of PaT-Nets both individually and in parallel; for additional detail about the PaT-Net structure, see Appendix D.

12.1 Executing a Single PaT-Net

When instantiated, a PaT-Net executes its initialization phase, instantiating local variables and executing the arbitrary LISP code associated with the initialization phase. Furthermore, it is added to the *active* list, and will enter its start node on the following simulation tick.

On each subsequent tick, the following steps take place:

- Each of the PaT-Net's monitors are checked. Currently, only the first monitor whose associated condition returns true has its action executed, however in *Jack* 6.0 the PaT-Net formalism will be extended to include multiple monitor execution.
- If the action associated with the current node has not been executed, then it is executed.
- The PaT-Net's waits are polled. If all wait conditions return non-*nil*, then execution continues; otherwise, the PaT-Net passes execution and its waits will be checked again on the following frame. Waits which return non-*nil* are removed from the list of waits.
- The PaT-Net queries the node for a transition. In the case of a probabilistic node, this requires the generation of a random value which is checked against the weights of each of its arcs. In the case of a simple node, each of the arcs' conditions are checked in order until one returns true, and the associated transition is taken.
- If no transition is taken, then no transition is made from the node. (The PaT-Net will not execute another action until it makes a transition.)
- If any waits are posted, they are added to an active wait list, which will be checked on every following tick.

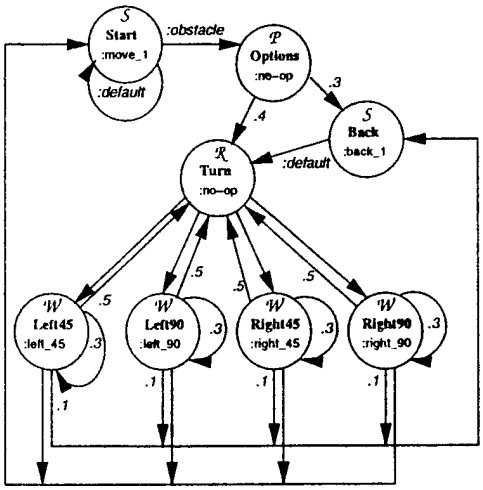


Figure 1: A Simple PaT-Net. Script letters indicate the type of node: S for simple, P for probabilistic, R for random, and W for weighted. Node names appear in **bold** and actions to be executed appear directly below node names. Conditions and weights are attached to transitions (arrows) and are designated by *italics*.

12.2 Example Execution of a Simple PaT-Net

Consider the PaT-Net displayed in Figure 1. This PaT-Net represents a very simple robot controller for navigating a maze. A sample environment for the robot is given in Figure 2; one could imagine testing the weights of given probabilities to see what differing results occured in the traversal of the maze. In any case, the following example will show the motion of the robot through the maze according to the PaT-Net of Figure 1.

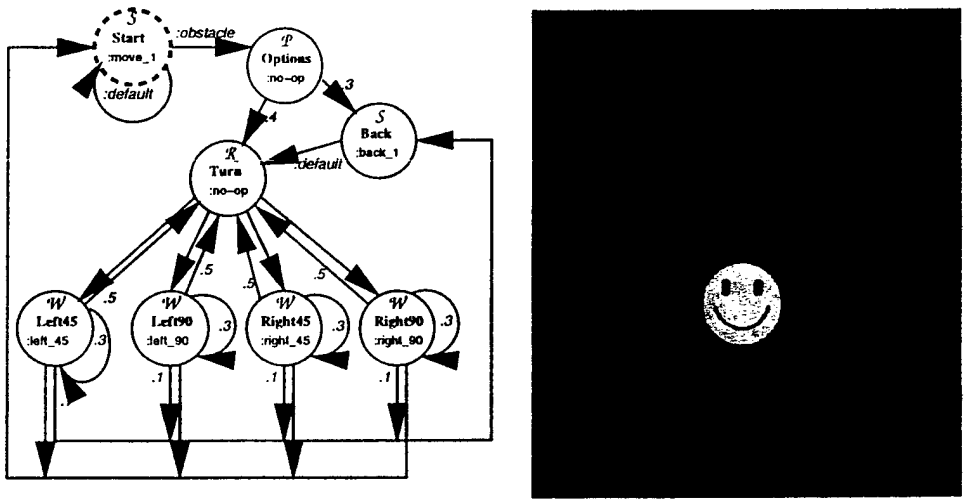


Figure 2: The PaT-Net and environment after the initialization phase. The dash-bordered node is the current node in the left diagram; the figure at the right shows the environment.

After the initialization phase of this PaT-Net enters its start node (Figure 2) and begins moving forward. In the absence of obstacles, this PaT-Net will locomote the robot forward

forever; in this simulation, a maze, this structure is appropriate.

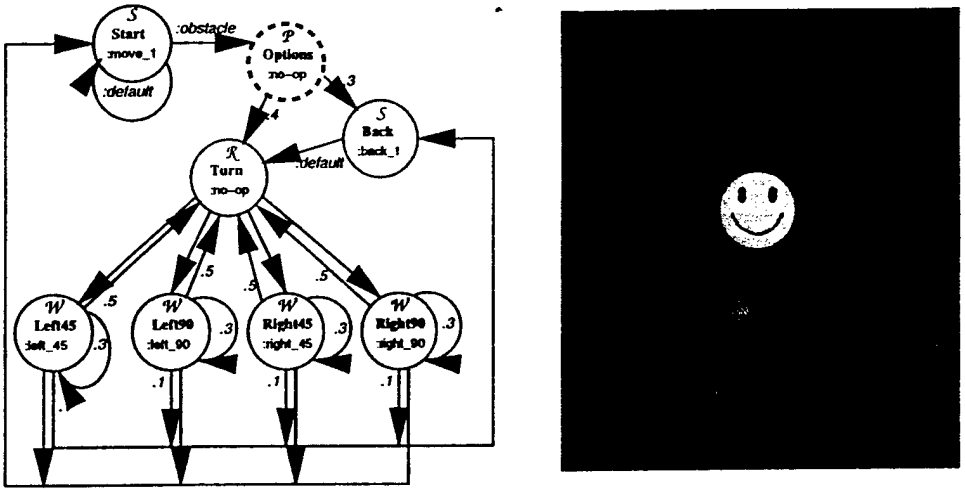


Figure 3: When it nears a wall, the robot will stop and entertain a number of options. The PaT-Net will be in the state depicted in the left figure; the current state is indicated by the dashed-border. A sample environment is shown on the right.

The robot locomotes until it encounters its first wall, when it must make a transition to the *options* node (Figure 3); the environment at this point is also displayed. This node is probabilistic: for purposes of discussion, we'll suppose that the follow node is the *turn* node.

After making a transition to the *turn* node (see Figure 4), where no action is taken, a transition is selected randomly from the possible transitions out of the node.

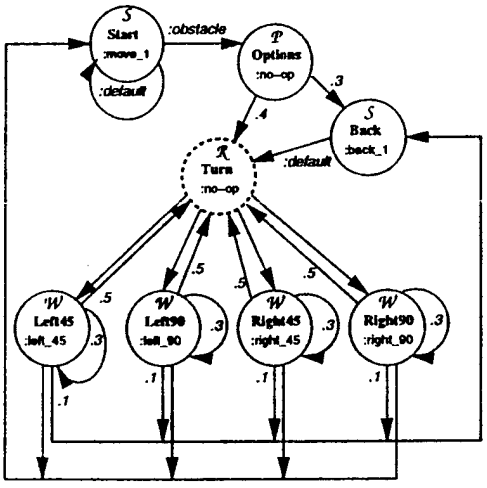


Figure 4: The PaT-Net after a transition to the *turn* node. The dash-bordered node is the current node.

Suppose the transition to the *left90* node is taken. After making this transition, the PaT-Net is in the state as shown in Figure 5. After executing the associated action of the node, the environment should look something like that shown in Figure 5.

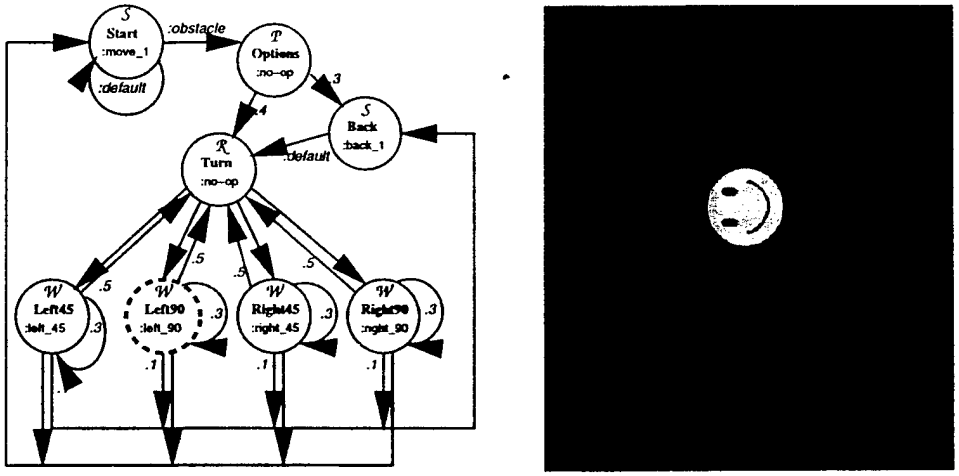


Figure 5: The PaT-Net and environment after a random transition to the *left90* node. The dash-bordered node is the current node; the figure on the right displays the environment after the execution of the action associated with the node.

12.3 Parallel Execution

As mentioned above, PaT-Nets are added to an *active list* as they are instantiated. This list of active PaT-Nets is maintained throughout a simulation; only a single active list may exist in a single *Jack* session.

PaT-Nets exhibit parallelism in a very restricted sense: each PaT-Net executes one action and transitions to another node in every simulation step (assuming that the PaT-Net isn't waiting). Thus, if three PaT-Nets are instantiated at time 0, then at the first tick, each PaT-Net will execute the action in its initial node. The actions of each of these PaT-Nets will occur *serially*, in the order in which they were added to the active list (i.e. the order in which they were instantiated).

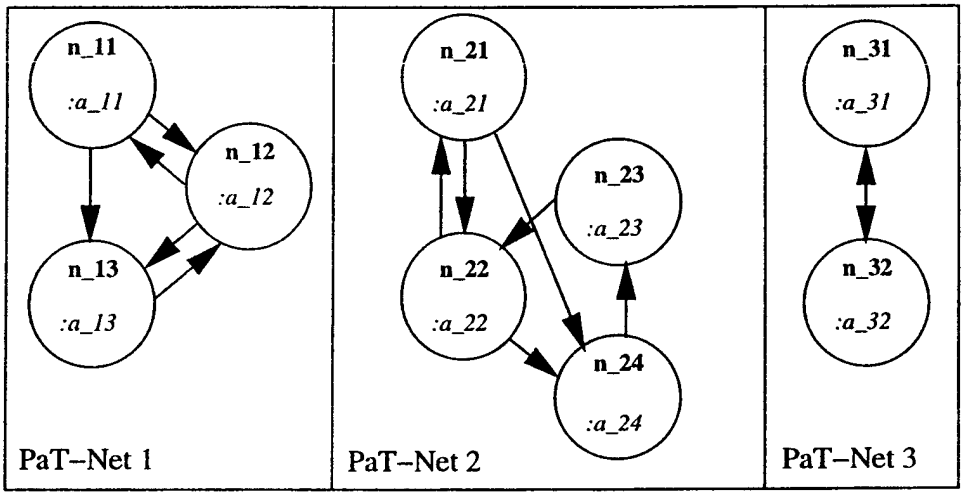


Figure 6: Three simple PaT-Nets.

Figure 6 shows three simple PaT-Nets to be instantiated in a simulation. PaT-Nets 1

and 2 will be instantiated on clock tick 0, and PaT-Net 3 will be instantiated on clock tick 2.

At time 0, PaT-Nets 1 and 2 are instantiated in that order. PaT-Net 1 executes its initialization phase and is added to the active list; then PaT-Net 2 executes its initialization phase and is added to the end of the active list.

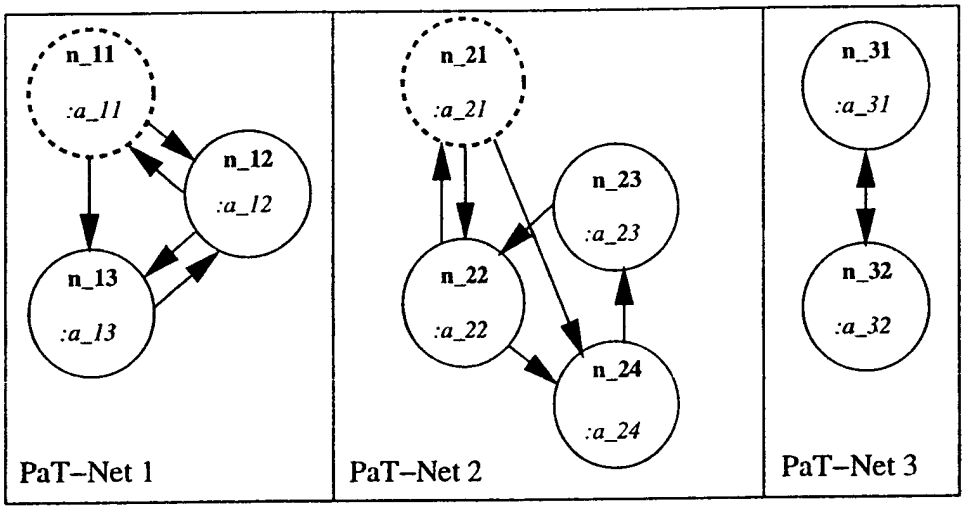


Figure 7: Three simple PaT-Nets immediately after the start of a simulation. Dashed borders reflect the current node of the PaT-Net.

At time 1, PaT-Nets 1 and 2 will both enter their initial states, as depicted in Figure 7. However, they will do this in the order in which they appear in the active list. PaT-Net 1 will execute action *:a_11* first; once a transition has been selected (for example, to node *n_12*), the PaT-Net will pass execution to PaT-Net 2. PaT-Net 2 will execute the action *:a_21* and make a transition (for example, to node *n_24*) and pass execution. Since there are no more active PaT-Nets on the active list, the simulation will continue.

At time 2, PaT-Nets 1 and 2 both continue execution by executing the actions associated with their current nodes. PaT-Net 1 will first execute action *:a_12* and select a transition (to node *n_13*); then PaT-Net 2 will execute action *:a_24* and select a transition (to node *n_23*). Remember that PaT-Net 3 is to be instantiated on this step; we will assume that this happens last (perhaps because it was instantiated in the action *:a_24* or by user input). PaT-Net 3 executes its initialization phase and execution continues.

At time 3, all three PaT-Nets continue execution. PaT-Net 1 executes action *:a_13*, then selects a transition. Then PaT-Net 2 repeats this process, executing action *:a_23* and selecting a transition. Finally. PaT-Net 3 executes action *:a_31* and selects a transition. Execution continues throughout the simulation.

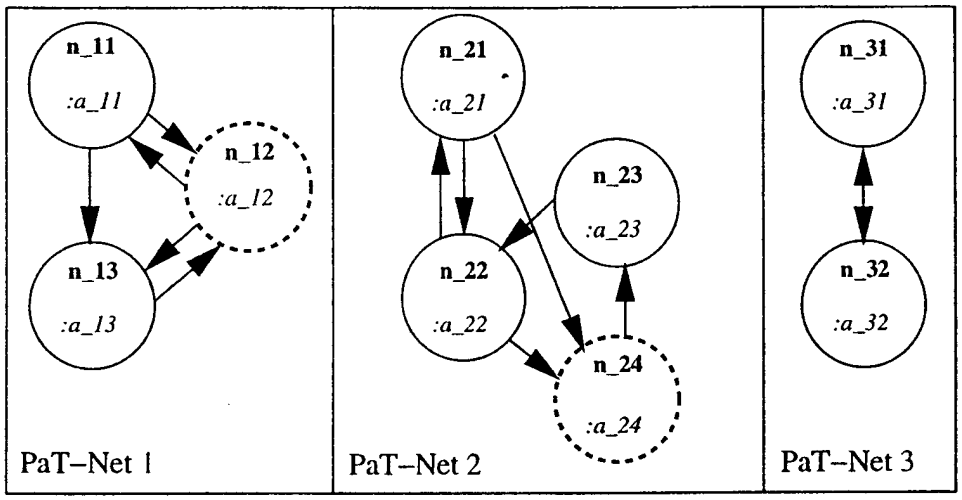


Figure 8: The three simple PaT-Nets at clock tick 2. Dashed borders reflect the current node of the PaT-Net.

A An Example PaT-Net

B The DEFNET Macro

C Semaphores

D Other Random Nonsense

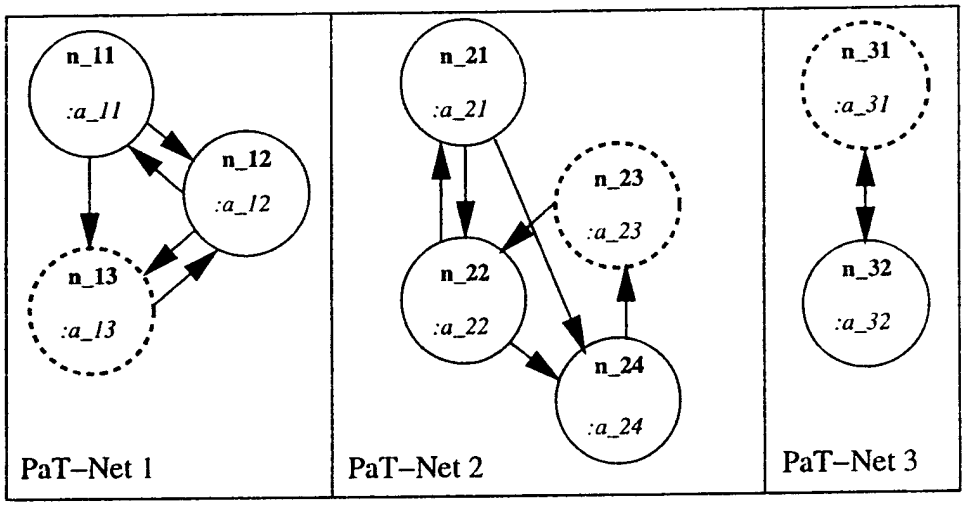


Figure 9: The three simple PaT-Nets at clock tick 3. Dashed borders reflect the current node of the PaT-Net; note that PaT-Net 3 has just become active.