The Trellis Project
Process Modeling for CSCW[*]

P. David Stotts

Department of Computer Science
University of North Carolina
Chapel Hill, NC 27599-3175

Richard Furuta

Hypermedia Research Laboratory
Department of Computer Science
Texas A&M University
College Station, TX 77843-3112

Abstract

We describe our work over the past several years on the Trellis project. The original Trellis model exploited the graph/automaton duality of classical Petri nets to formally describe interactive information systems. We show applications in a hypertext system called αTrellis, our first implementation based on this early model. We then discuss a newer Trellis model based on colored place/transitions nets. The new model is intended to allow expression of collaboration protocols in the context of interactive information systems. We are in the process of implementing this collaborative model in an X-windows-based hypermedia system called χTrellis.

1 Trellis and αTrellis

The Trellis project [SF89a, SF89b, FS89] has investigated for the past several years the structure and semantics of human computer interaction, in the context of hypertext (hypermedia) systems, program browsers, visual programming notations, and process models. In the ensuing discussion, we will refer to an information structure in Trellis as a *hyperprogram*. Due to the unique features combined in the Trellis model, a hyperprogram integrates user-manipulatable information (the hypertext) with user-directed execution behavior (the process). We say that a hyperprogram *integrates task with information*.

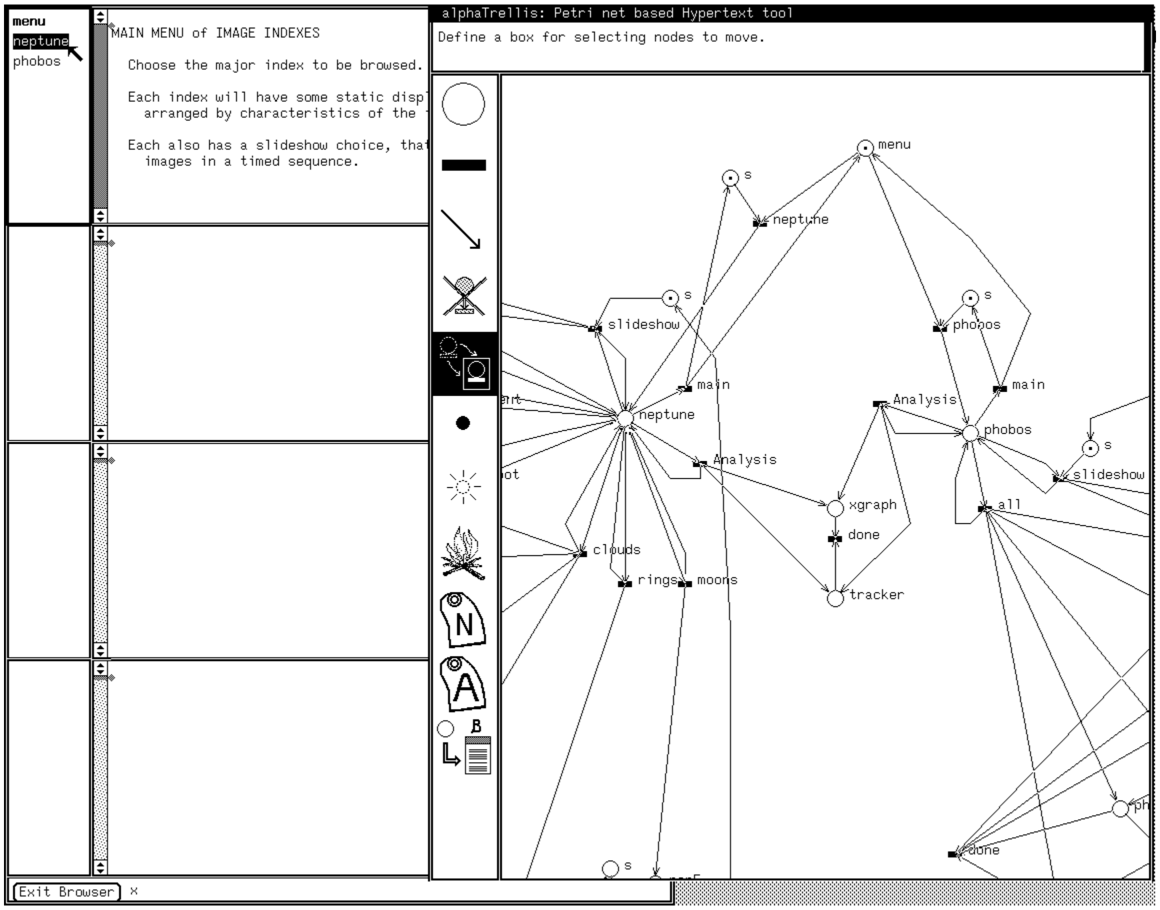
In the following sections we will summarize the Trellis model and semantics, and then describe the distributed architecture of our Trellis implementations. To illustrate the capabilities of the model, we give specific examples of how Trellis hyperprograms are used as information repositories (image browsing indexes), as parallel program browsers, and as general process models. We conclude with a description of our methods of analyzing the task that is encoded into a Trellis hyperprogram.

1.1 Place/transition nets and browsing semantics

The Trellis model treats a hyperprogram as an annotated, timed, place/transition net (PT net) that is used both as a graph (for static information) and as parallel automaton (for dynamic behavior). For complete understanding of the example in following sections, we very briefly explain here the syntax and semantics of PT nets.

An example PT net is shown in the graphics window on the right side of figure 1 (the software system itself is discussed in more detail later). PT nets are graphically represented as bipartite graphs in which the circular nodes are called *places* and the bar nodes are called *transitions*. A dot in a place is called a *token*, and it represents activity, or the realization of some logical condition associated with the place.

[*]This work is based upon work supported by the National Science Foundation under grant numbers IRI–9007746 and IRI–9015439, as well as by the Software Engineering Research Center of the University of Florida and Purdue University.



alphaTrellis: Petri net based Hypertext tool

Define a box for selecting nodes to move.

MAIN MENU of IMAGE INDEXES

Choose the major index to be browsed.

Each index will have some static disp
arranged by characteristics of the

Each also has a slideshow choice, tha
images in a timed sequence.

Figure 1: Screen from αTrellis showing an image browsing index.

A place containing one or more tokens is said to be *marked*. When each place incident on a transition is marked that transition is *enabled*. An enabled transition may *fire* by removing one token from each of its input places and putting one token into each of its output places. The full token distribution among places is the *state* of the net and is termed a *net marking*. A state change in the net marking that results by firing an enabled transition.

The places of the net are annotated with fragments of information (text, graphics, video, audio, executable code, other hyperprograms); these annotations are termed the *content* elements of the hyperprogram. A hierarchy is created by allowing the content element of a place to be itself an independent hyperstructure.

To use a Trellis model as hypertext, for example, we provide visual interfaces to give a user a tangible interpretation of the PT net and its annotations. When a token enters a place during execution, the content element for the place is presented for viewing (or for other user consumption). Any enabled transitions leading out of the place are shown next to the displayed content element as selectable *buttons*, or hot spots in the interface. Selecting a button (with a mouse, usually) will cause the net to fire the associated transition, moving the tokens around and changing the visible content elements.

We mentioned that a hyperprogram integrates task with information. If one takes the graph view of a PT net, then the content elements and the arcs among them collectively comprise a linked information base; if one considers the parallel automaton view of a PT net, then its execution behavior defines a task composed of concurrent control threads running throughout the information base. The dual nature of a PT net is the integration.

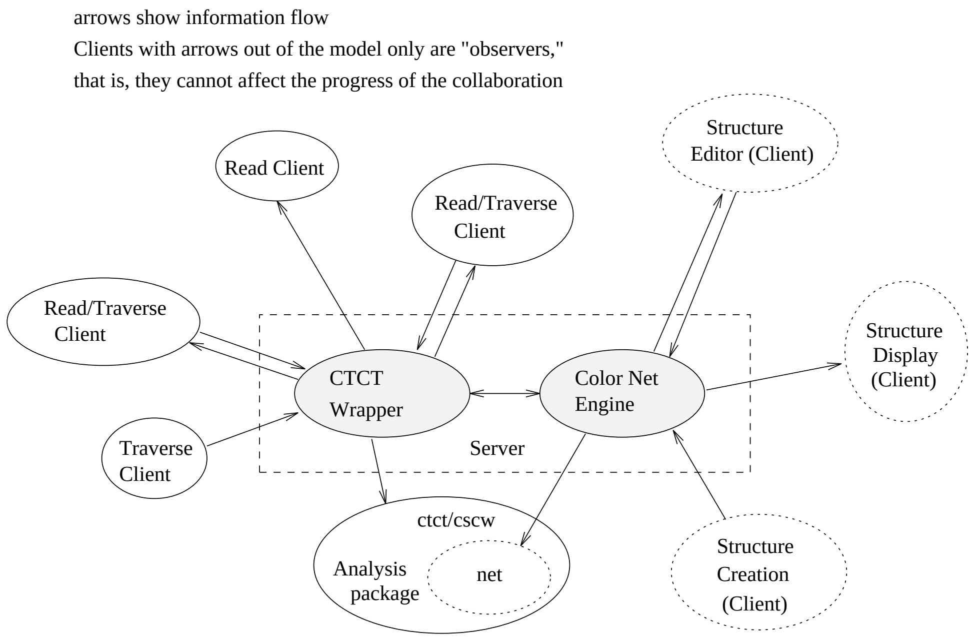

arrows show information flow
Clients with arrows out of the model only are "observers,"
that is, they cannot affect the progress of the collaboration
Read Client
Read/Traverse Client
Structure Editor (Client)
Read/Traverse Client
Structure Display (Client)
CTCT Wrapper
Color Net Engine
Traverse Client
Server
ctct/cscw
Analysis package
net
Structure Creation (Client)

Figure 2: Trellis client/server system architecture.

1.2 Client/server system architecture

Figure 2 shows the high-level structure of a Trellis-based implementation. The components illustrate the essential features that make Trellis inherently extensible, making it customizable by an organization using it for modeling. The models are basically information servers. The information contained in a model (quality metrics, design structure, reliability models, documentation network) can be accessed and used for varying purposes, depending on the particular interface clients that are written to access the information servers.

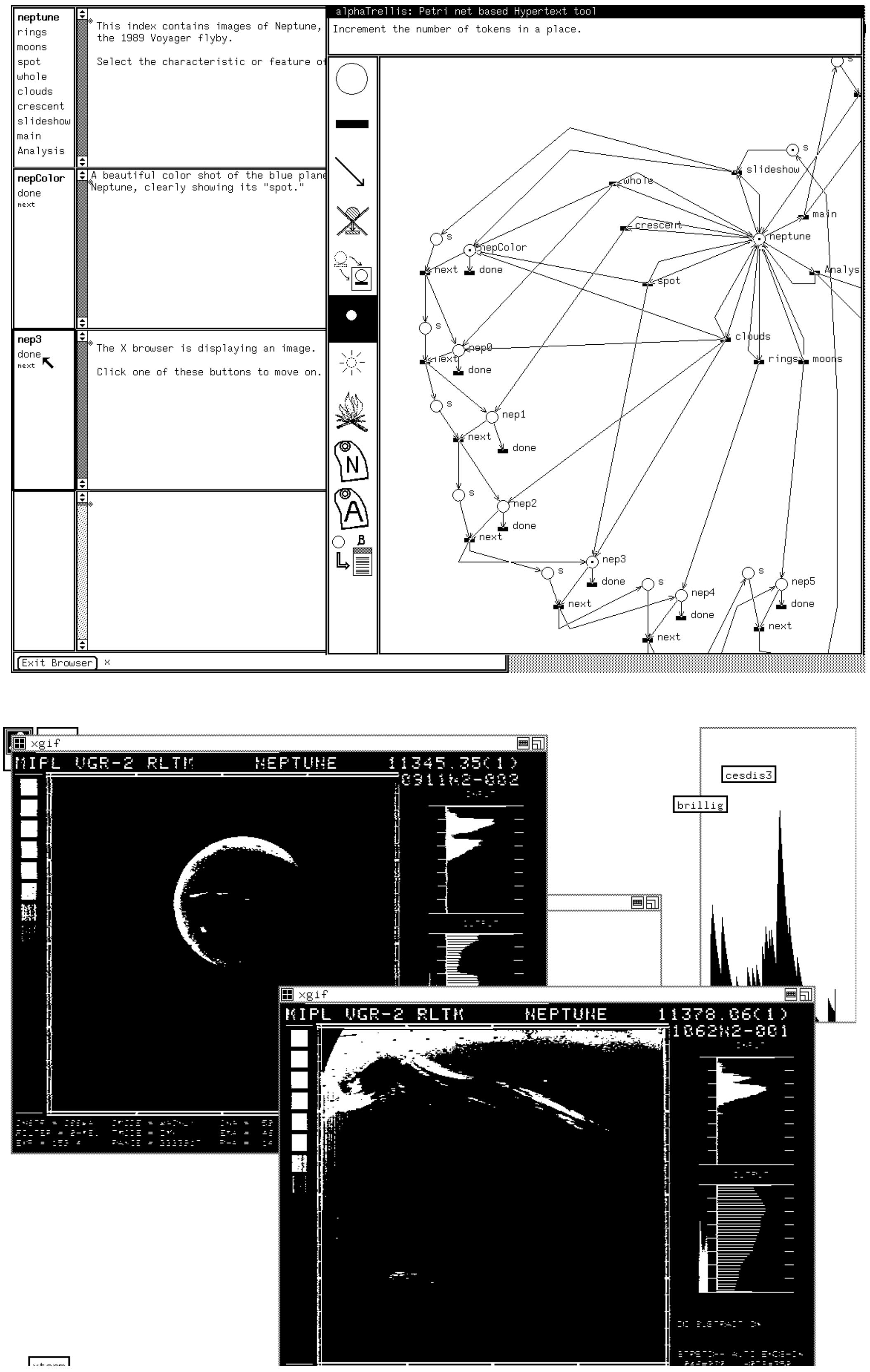
The heart of a Trellis system is an information engine, which is a process allowing construction and execution of a Trellis model (annotated, timed PT net as discussed previously). An engine (model) has no visible interface, but does respond to remote procedure call (RPC) requests for its services. Services provided include those operational one mentioned above, as well as basic functions such as constructing or altering the model's components. The current Trellis engine is implemented in C++ and has over 50 methods comprising its services.

A client in a Trellis system executes as a separate process (perhaps remotely), communicating with an engine via RPC. Clients provide visible interfaces for models (engines). Different interface clients might be written, for example, to

- operate in different window systems;
- provide different views or present different aspects of a model;
- structure models for specific purposes or, by conventions of different application domains;

or for numerous other reasons. All clients, however, interact with the same models, using the RPC API provided by the engine to add information, add links, add agents, change the state of the model, etc.

Though it is not shown explicitly in the figure, multiple Trellis engines (models) can be concurrently active, and each may be accessed by multiple clients concurrently. Any particular client can concurrently access several models. In addition to these forms of concurrency, the engine itself is structured as a parallel automaton (that is, encodes parallel threads of activity) as previously mentioned.


neptune
rings
moons
spot
whole
clouds
crescent
slideshow
main
Analysis

This index contains images of Neptune, the 1989 Voyager flyby.

Select the characteristic or feature of

nepColor
done
next

A beautiful color shot of the blue plane Neptune, clearly showing its "spot."

nep3
done
next

The X browser is displaying an image.

Click one of these buttons to move on.

Exit Browser

alphaTrellis: Petri net based Hypertext tool

Increment the number of tokens in a place.

Figure 3: Concurrently displayed Neptune images classified by features.

1.3 Example: image browsing index

Figure 1 shows a screen from αTrellis, an early Trellis prototype implemented for Unix platforms and the SunView window system. Two of the three αTrellis browsing clients are visible, with the graphical editing client on the right, overlapping the windows of the text browsing client on the left. αTrellis was an experimental, proof-of-principle vehicle to demonstrate the utility of the automaton-based view of hyperdocuments; as such, we focussed on implementing the net engine functionality and analysis methods rather than developing complex interface clients.

In the αTrellis text browser, the screen shows four text windows. When a net place is marked its content element is displayed in one of these text windows. In this example, the text showing in the top window of the browser is the content of the place "menu" which is showing as marked in the graphical editor window. Each enabled transition is displayed as a selectable button in a menu to the left of the text window (transitions "neptune" and "phobos" in the example net). Selection of a button in a browser menu causes the associated transition to fire in the net, changing the net state and thereby causing a change in the information elements that are displayed in the text windows.

The PT editor client on the right presents a graphical view of the underlying model. With it, a user can build or alter the structure of the net, annotate the net with content element names, and also execute the net. The two clients execute as two independent processes, and the net engine executes as a third process. Each client communicates with the engine by RPC. When one client causes a change in the net (e.g., by firing a transition), the other client will be notified and will reflect the change also.

Figure 3 shows a third αTrellis client (displaying on a different monitor with X windows) operating concurrently with the editor and browser clients. This client monitors the execution activity in a hyperprogram and displays graphics images on the X windows screen whenever a marked place has a bit-mapped image as its content. This example shows an image browsing index we constructed as part of a NASA experiment at CESDIS (Goddard Space Flight Center, MD). The images of Neptune and Phobos are linked and cross linked in the net structure according to common characteristics. The net as built concurrently displays all images that share a characteristic. The state shown in figure 3 results when the "neptune" button highlighted back in figure 1 is selected, followed by the "spot" button in the text frame atop figure 3. These two transition firings leave the token in place "neptune", and also place tokens into "nepColor" and "nep3"; the associated content images are displayed remotely by the graphics client as shown in the botton frame of figure 3.

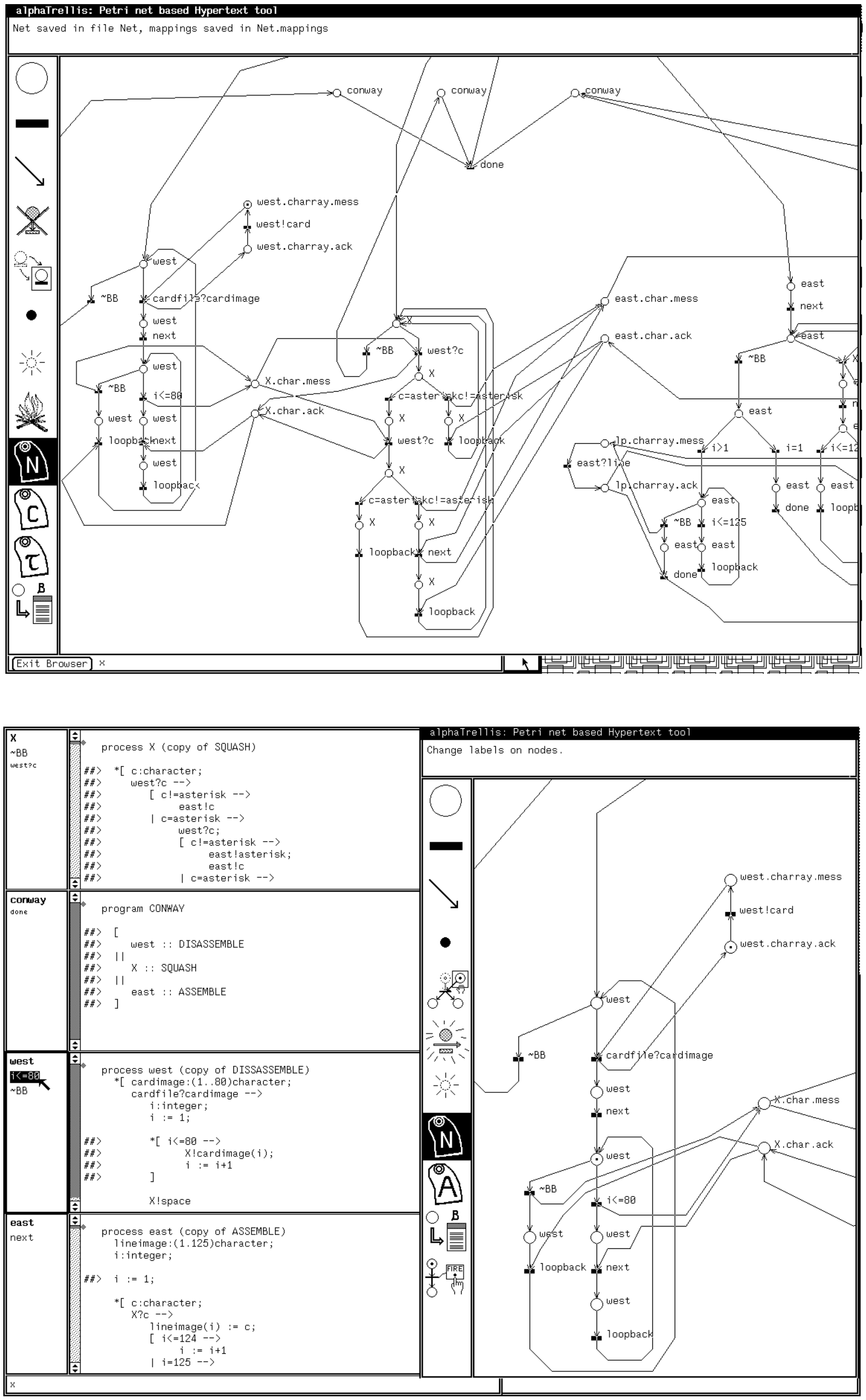
The graphics client does not allow a user to alter the net structure like the editor client does; it does not even allow a user to fire transitions like the text browser does; it just sits and listens, acting when necessary according to its purpose. To initiate some engine activity, a user would have to employ with the interface provided by one of the other two clients.

1.4 Example: Parallel program browsing

The αTrellis application illustrated in this section shows both the usefulness of the model for representing parallel threads of activity, and the usefulness of our hypertextual interpretation of the PT net for supporting human reasoning through browsing. A CSP program browser [SF90a] is shown in figure 4. This specific example uses a CSP program published in Hoare's original paper published in the *Communications of the ACM*.

We wrote a translator to parse CSP programs and generate as output the storage format of Trellis hyperprograms. The translation converted the control structures of CSP statements and the message buffers between CSP processes into PT net structures with the appropriate control behaviors. Each place in the PT net represents a statement from the source program. We annotated the places with CSP source code; each place is mapped to a copy of the CSP process that contains its statement, with that statement highlighted (this gives a reader some context for the statement).

The result is a browsing system for simulating the parallel execution of CSP programs. The simulation proceeds by selecting buttons in the text browser to "execute" statements one at a time. The simulation proceeds at user speed and at a user's discretion, following a user's train of thought as browsing progresses.



```
Net saved in file Net, mappings saved in Net.mappings
```

```
Change labels on nodes.
```

```
   process X (copy of SQUASH)

##>  *[ c:character;
##>     west?c -->
##>        [ c!=asterisk -->
##>             east!c
##>        | c=asterisk -->
##>             west?c;
##>             [ c!=asterisk -->
##>                  east!asterisk;
##>                  east!c
##>             | c=asterisk -->
```

```
   program CONWAY

##>  [
##>     west :: DISASSEMBLE
##>  ||
##>     X :: SQUASH
##>  ||
##>     east :: ASSEMBLE
##>  ]
```

```
   process west (copy of DISSASSEMBLE)
     *[ cardimage:(1..80)character;
        cardfile?cardimage -->
           i:integer;
           i := 1;

##>        *[ i<=80 -->
##>              X!cardimage(i);
##>              i := i+1
##>        ]

           X!space
```

```
   process east (copy of ASSEMBLE)
     lineimage:(1.125)character;
     i:integer;

##>  i := 1;

     *[ c:character;
        X?c -->
           lineimage(i) := c;
           [ i<=124 -->
                i := i+1
           | i=125 -->
```

Figure 4: αTrellis used for browsing a CSP parallel program.

The top view of figure 4 shows the editor client full-screen to illustrate the graphical structure of this particular model. The bottom view shows the editor client with a closeup of the net, with the text browser displaying the CSP code segments that are active at this point in simulated execution.

1.5 Example: Process simulation in Trellis

We just saw user-directed simulation of processes in Trellis. This section illustrates a facility of the model that allows non-user directed control in a process simulation. The method uses timed transitions in the net, a Lisp interpreter in the αTrellis engine, and chunks of Lisp code (called *agents*) on net transitions [SF90b, SF92]. When a transition is fired, its Lisp agent (if one is present) is executed. Trellis in this form is like the concurrent language Linda, in that a sequential kernel language (Lisp) is separate from the parallel control flow language (timed PT nets).

Lisp agents are responsible for, among other things, setting control traps and triggers as the net executes. A transition in a Trellis model has two time values–a minimum and a maximum. By default, the minimum is 0 units and the maximum is ∞. When a transition first becomes enabled, the minimum time must pass before the engine will honor a request to fire it; if the maximum time passes after enabling without any client firing the transition, then the engine will fire it automatically. In effect, the minimum time is a delay, and the maximum time is a time-out. Under these semantics, the "$0,\infty$" default timings cause a transition to behave exactly like an untimed transition (i.e., no delay, never times out).

In the αTrellis engine, the time on a transition can either be a constant or it can be the value of a Lisp variable. Thus execution of a Lisp agent can change transition time values while a hyperprogram is being browsed. To illustrate, consider a net structure that pops up a help window automatically if a reader sits idle for a certain period of time. An author cannot hope to assign an initial time-out value to this popup transition that is comfortable for all readers (for some it will be too fast, for others too slow). However, he can assign a reasonable initial value and then put an *adaptation* agent on transitions in the net. This agent will compute a running average of the time each reader spends looking at each content element. Then, every so many button clicks, the Lisp variable for the popup transition timing is set to some multiple of the reader's average.

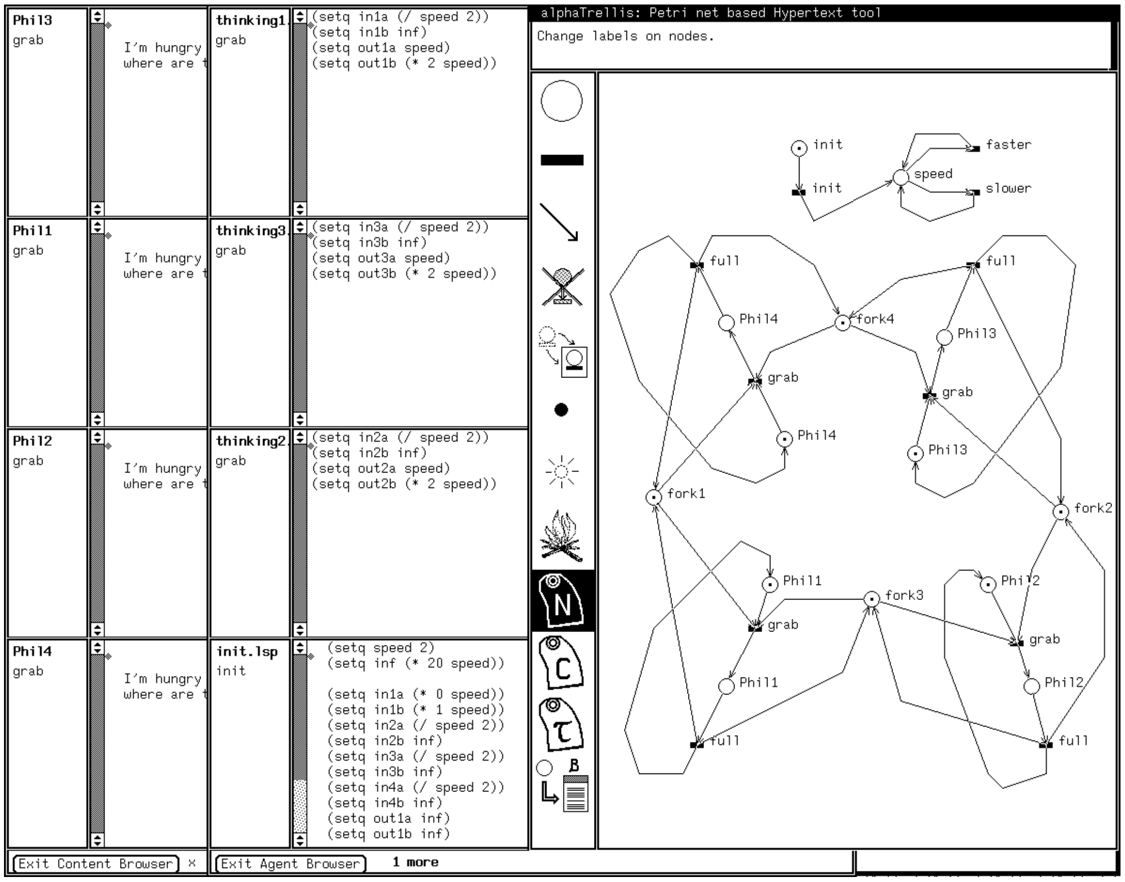
The αTrellis engine has a very fine grained internal clock. Individual hyperprograms can contain their own clocks, at their own speeds, by including a place/transition loop with the transition timed to fire at some interval; when it fires, its Lisp agent increments a Lisp variable "clock". Then, the averaging agent will look like this:

```
(setq clicks (+ clicks 1))
(cond ( (eq clicks ccount)
          (setq clicks 0)
          (setq dwell (/ (+ dwell (/ (- clock oclock) ccount)) 2))
          (setq min (* 3 dwell))
          (setq max (* 6 dwell))
          (setq oclock clock)      ) )
```

The Dining Philosophers process

The canonical concurrency example of the Dining Philosophers will further illustrate Lisp agents in Trellis process simulation. An implementation of four philosophers is shown in Figure 5. Shown is the initial screen when the hyperprogram is invoked, with a view giving the names of the various components. Two text browsing clients are executing with the αTrellis editor client; the leftmost browser shows place content elements (which in this example are inconsequential) and the middle browser shows Lisp agents on the enabled transitions.

The internal clock for this hyperprogram is the detached pair of loops visible at the top of the editor window, with transitions labeled "slower" and "faster" to alter the execution speed of dining. Initially, the timings on philosopher fork events are ($0,\infty$), so no action takes placec until the user is ready. When the button labeled "init" is selected (and its agent "init.lsp" executed), the timings on transitions are altered to the ones shown in figure 6. The timings on philosopher fork events then allow a reader two



Phil3
grab

I'm hungry
where are t

thinking1.
grab

```
(setq in1a (/ speed 2))
(setq in1b inf)
(setq out1a speed)
(setq out1b (* 2 speed))
```

Phil1
grab

I'm hungry
where are t

thinking3.
grab

```
(setq in3a (/ speed 2))
(setq in3b inf)
(setq out3a speed)
(setq out3b (* 2 speed))
```

Phil2
grab

I'm hungry
where are t

thinking2.
grab

```
(setq in2a (/ speed 2))
(setq in2b inf)
(setq out2a speed)
(setq out2b (* 2 speed))
```

Phil4
grab

I'm hungry
where are t

init.lsp
init

```
(setq speed 2)
(setq inf (* 20 speed))

(setq in1a (* 0 speed))
(setq in1b (* 1 speed))
(setq in2a (/ speed 2))
(setq in2b inf)
(setq in3a (/ speed 2))
(setq in3b inf)
(setq in4a (/ speed 2))
(setq in4b inf)
(setq out1a inf)
(setq out1b inf)
```

Exit Content Browser ×   Exit Agent Browser   **1 more**

alphaTrellis: Petri net based Hypertext tool

Change labels on nodes.

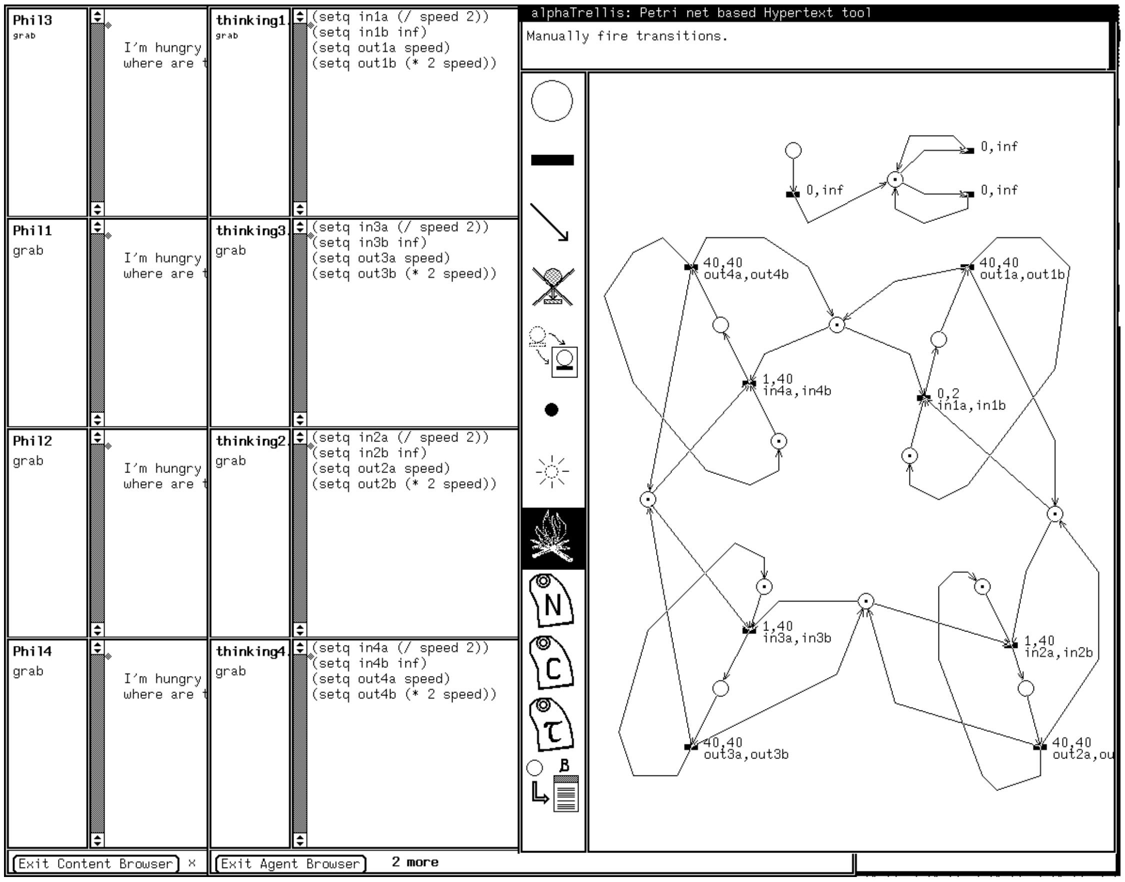
Figure 5: Initial Dining Philosophers screen.

ways to influence process execution. A philosopher can be made to directly pick up a fork by selecting the corresponding button in the text browser (or in the agent browser); however, if the user just sits back and watches, one philosopher (the one with lower timing values) will time out and pick up a fork automatically. Picking up the fork causes execution of a Lisp agent that further alters the timings so the fork will then be put down automatically. Putting the fork down runs a Lisp agent that alters the timings again so the next philosopher picks up a fork, etc. In this way, the Lisp agents implement a round-robin scheduling policy that will take over whenever the reader does not directly select the execution order. Other scheduling policies can obviously be simulated by writing different agents to adjust the timing triggers appropriately.

2 Net process analysis: model checking

Trellis and its implementations provide a formal structure for hyperprograms, and net analysis techniques have been developed for exploiting this formalism. One very promising approach involves our adaptation of automated verification techniques called *model checking* [CES86] from the domain of concurrent programs. This approach allows verification of browsing properties of Trellis hyperprograms expressed in a temporal logic notation called CTL. An author can state a property such as "no matter how a document is browsed, if Node X is visited, Node Y must have been visited within 10 steps in the past." The model checker efficiently verifies that the PT net structure maintains the validity of the formula denoting the property.

In model checking, a state machine (the model) is annotated with atomic properties that hold at each state (such as "content is visible" or "button is selectable"), and then search algorithms are applied



```
(setq in1a (/ speed 2))
(setq in1b inf)
(setq out1a speed)
(setq out1b (* 2 speed))
```

```
(setq in3a (/ speed 2))
(setq in3b inf)
(setq out3a speed)
(setq out3b (* 2 speed))
```

```
(setq in2a (/ speed 2))
(setq in2b inf)
(setq out2a speed)
(setq out2b (* 2 speed))
```

```
(setq in4a (/ speed 2))
(setq in4b inf)
(setq out4a speed)
(setq out4b (* 2 speed))
```

Figure 6: Timings after firing Init button.

to the graph of the state machine to see if the subcomponents of a formulae hold at each state. By composing the truth values of these subformulae, one obtains a truth value for the entire formula. For PT nets, we obtain a state machine from the *coverability graph*.

The details of our use of CTL are discussed elsewhere [SFR92]. For this rationale, it is sufficient to give an idea of how the method is applied to Trellis models. The Trellis document shown in Figure 7 is a small net that expresses the browsing behavior found in some hypertext systems, namely that when a link is followed out of a node, the source content stays visible and the target content is added to the screen. The source must later be explicitly disposed of by clicking a "remove" button.

After computing the coverability graph and translating it into the input format required by the checking tool, the model can be queried for desired browsing properties. These examples use the syntax of Clarke's CTL model checker, and show its output:

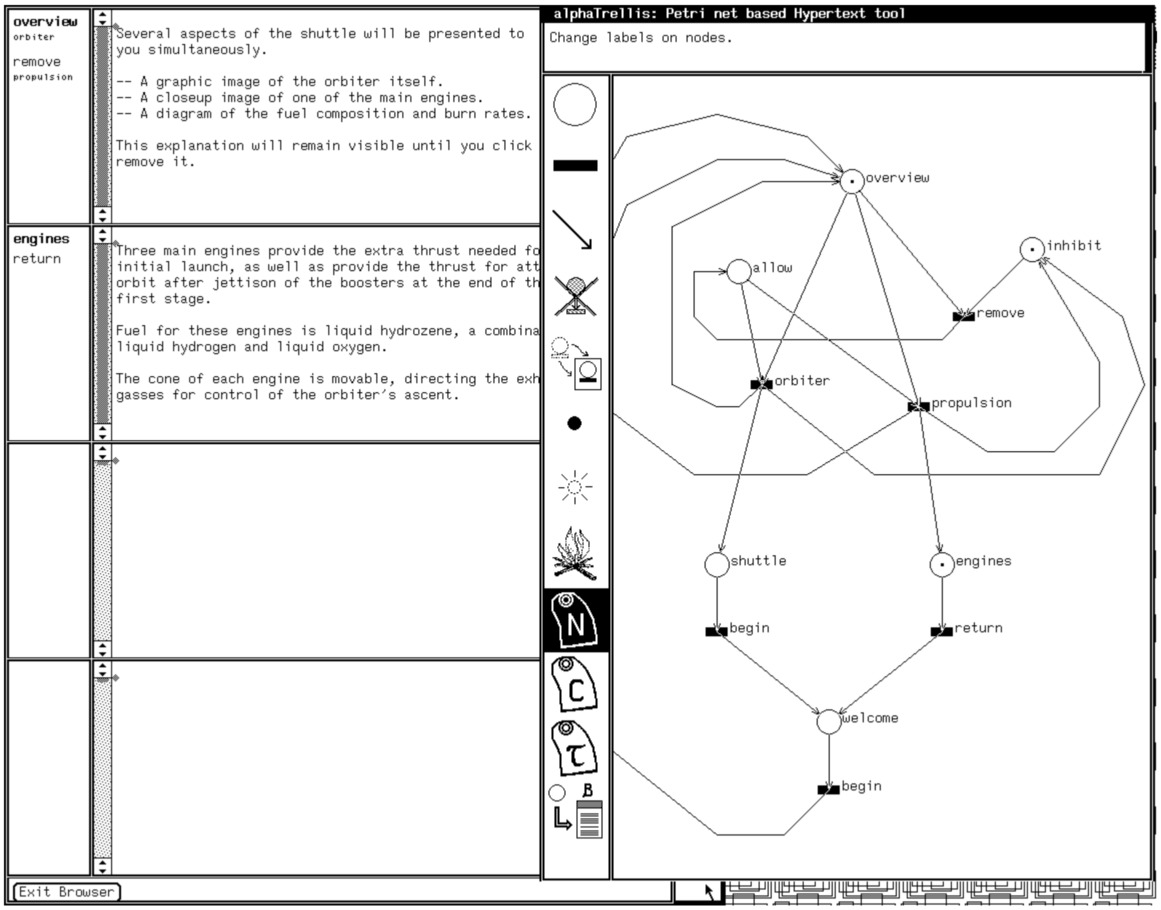
- Is there some browsing path such that at some point both the "orbiter" and "propulsion" buttons are selectable on one screen?

  ```
  |= EF(B_orbiter & B_propulsion).
  The formula is TRUE.
  ```

- Is it impossible for both the "shuttle" text and the "engines" text to be concurrently visible?

  ```
  |= AG( ~C_shuttle | ~C_engines ).
  The formula is TRUE.
  ```

- Can both the "allow" access control and the "inhibit" access control ever be in force at the same time?



Several aspects of the shuttle will be presented to you simultaneously.

-- A graphic image of the orbiter itself.
-- A closeup image of one of the main engines.
-- A diagram of the fuel composition and burn rates.

This explanation will remain visible until you click remove it.

Three main engines provide the extra thrust needed fo initial launch, as well as provide the thrust for att orbit after jettison of the boosters at the end of th first stage.

Fuel for these engines is liquid hydrozene, a combina liquid hydrogen and liquid oxygen.

The cone of each engine is movable, directing the exh gasses for control of the orbiter's ascent.

Change labels on nodes.

Figure 7: Small Trellis structure with programmed browsing behavior

```
|= EF(C_inhibit & C_allow).
The formula is FALSE.
```

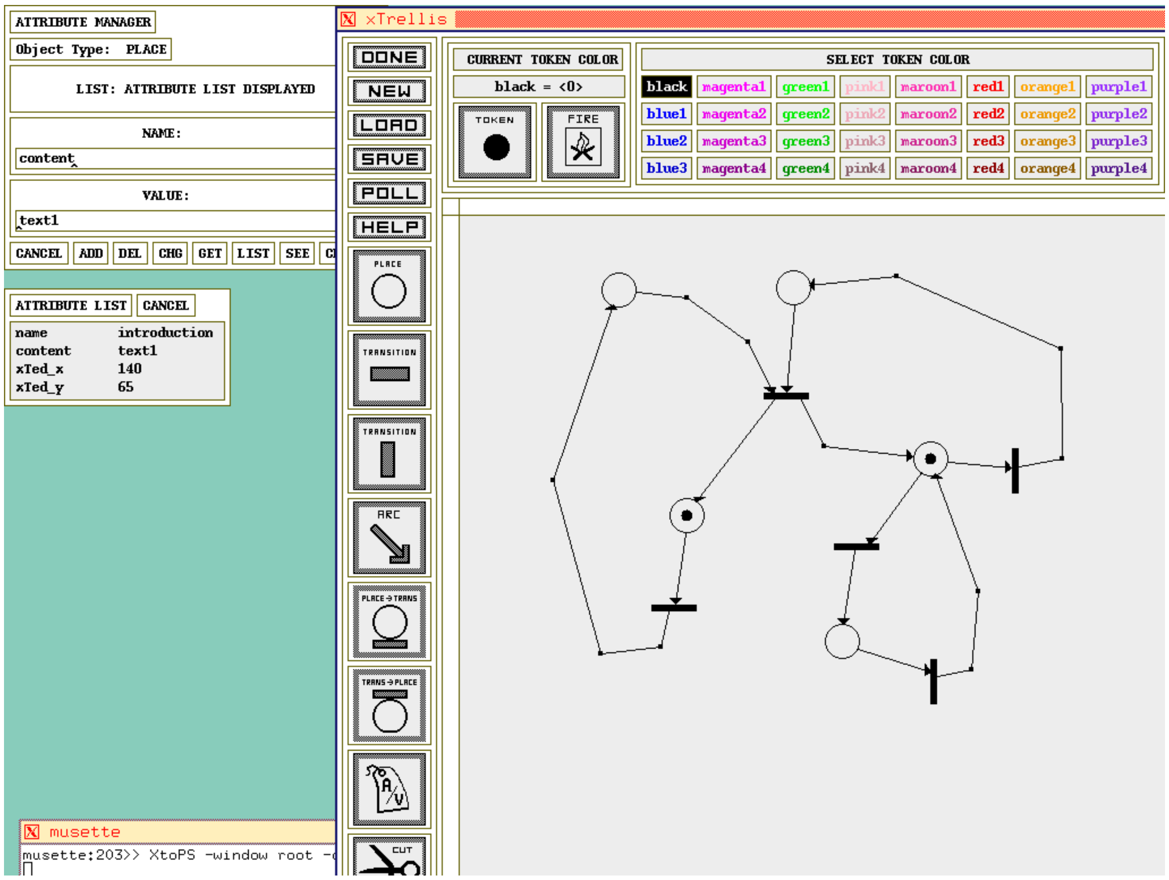
- Is it possible to select the "orbiter" button twice on some browsing path without selecting the "remove" button in between?

```
|= EF(B_orbiter & AX(A[B_remove U B_orbiter])).
The formula is FALSE.
```

This particular Trellis model is very small compared to those encountered in realistic applications. Our checker has also been tested on larger Trellis documents—for example, the one shown back in Figure 4. The state machine derived from this net contains over six thousand states. Using a DECstation 5000/25, the performance of the model checker on formulae like those above is mostly on the order of a few seconds each, with the most complicated query we tried (not shown) requiring about 15 seconds to answer. We suspect that authors of Trellis models will find such performance not at all unreasonable for establishing the presence or absence of critical browsing properties, and we also expect that future implementations will exhibit improved performance.

3 χTrellis: Collaboration protocols and hypermedia

The preceding section introduced the αTrellis prototype in some detail because our early experiments in process modeling have been performed with it. In the past year, with support from the Purdue/Florida/NSF Software Engineering Research Center, we have begun re-developing the Trellis technology into a newer prototype called χTrellis.



Object Type: PLACE

NAME: content

VALUE: text1

name	introduction
content	text1
xTed_x	140
xTed_y	65

black = <0>

```
musette:203>> XtoPS -window root -
```

Figure 8: χTrellis collaborative model editor, showing attribute/value annotations.

χTrellis consists of a more expressive net engine and clients written for X windows. The χTrellis color net editor client is shown in figure 8. It runs as a process separate from any model it is editing, and communicates with models via RPC. Multiple copies of this client can run at one time and interact in editing a single Trellis model. Changes made in one editor show up in the other interfaces.

While the clients are constructed to allow collaborative access to the engine, the engine itself is designed to build models that formally encode collaborative group activity. The new engine is based on a colored PT net formalism, while retaining the timing and Lisp agents of the earlier engine. Colors provide a typing scheme for tokens so that two tokens can be distinguished by the net and acted on differently as appropriate. Instead of a common firing rule for all transitions in a net, each transition in a colored net has its own firing *predicate*. A predicate describes what combinations of input token colors are required for enabling, and what colors of output tokens results from firing. For example, a transition's predicate may specify "fire if each input place contains a different color token, and produce black tokens on all outputs;" or, "fire if the first input place has a blue token and the second place has any color $< x >$, producing a token of that same color $< x >$ as output."

Using colors to represent different members of a group, or different task categories, or different instances of some object, a χTrellis model can formally represent the interactions among members of a collaborating group. For example, a net might force each activity thread in some section to be traversed by different participants (no one member doing more than one job); it might force one participant to be a manager and the other to be an engineer (different access classes); it might specify that design reviews are valid if performed by at least two individuals from a large class of possible participants.

The newer χTrellis client have a unified handling of text and graphics data, unlike the earlier αTrellis clients. We are developing voice and video clients as well.

4 Brief comparison with ongoing research

The use of PT nets as a specification medium for man-machine interaction appears previously in the literature. For example, van Biljon [vB88] has described a special grammar-based notation for designing man-machine dialogues as languages, which are then realized with a hierarchy of PT nets as recognition automata. Another example is the work of Holt [Hol88], who has designed a PT-net-based graphical specification language for coordination of multiple cooperating agents in an information processing organization. Trellis is a more complicated model than these previous proposals, because it encompasses more than just the control aspects of man-machine interactions. It contains an inherent notion of information presentation (text, graphics, executable code), has timing for events, and in later versions includes a Lisp interpreter as a attribute processing facility.

The underlying Trellis information engine is related to other hypertext engines that have been used in experimental software support systems. The HAM (hypertext abstract machine) [CG88] was developed in 1986 by Textronix, and was used as the basis for a hypertextual software support system called Neptune. The uniqueness of Trellis is the basis on a parallel collaborative computation model—colored PT nets. This gives the model an elegant structure that can be both programmed and analyzed. Scacchi and Garg have also used a hypertext mechanism in a software engineering context [GS87]. Their project, though, concentrated on the object-base aspects of a software project and did not have a formal model for representing process and enactment.

References

[CES86] E. Clarke, E. A. Emerson, and S. Sistla. Automatic verification of concurrent systems. *ACM TOPLAS*, 8(2):244–263, April 1986.

[CG88] Brad Campbell and Joseph M. Goodman. HAM: A general purpose hypertext abstract machine. *Communications of the ACM*, 31(7):856–861, July 1988.

[FS89] Richard Furuta and P. David Stotts. Programmable browsing semantics in Trellis. In *Hypertext '89 Proceedings*, pages 27–42. ACM, New York, November 1989.

[GS87] P. Garg and W. Scacchi. On designing intelligent hypertext systems for information management in software engineering. In *Proceedings of Hypertext '87 (Chapel Hill, NC, November 1987)*, pages 409–432, 1987.

[Hol88] Anatol W. Holt. Diplans: A new language for the study and implementation of coordination. *ACM Transactions on Office Information Systems*, 6(2):109–125, January 1988.

[SF89a] P. David Stotts and Richard Furuta. Petri-net-based hypertext: Document structure with browsing semantics. *ACM Transactions on Information Systems*, 7(1):3–29, January 1989.

[SF89b] P. David Stotts and Richard Furuta. αTrellis: A system for writing and browsing petri-net-based hypertext. In *Proceedings of the Tenth International Conference on Application and Theory of Petri Nets*, pages 312–328, June 1989. Bonn, W. Germany.

[SF90a] P. D. Stotts and R. Furuta. Browsing parallel process networks,. *Journal of Parallel and Distributed Computing*, 9(2):224–235, 1990.

[SF90b] P. David Stotts and Richard Furuta. Temporal hyperprogramming. *Journal of Visual Languages and Computing*, 1(3):237–253, 1990.

[SF92] P. D. Stotts and R. Furuta. Hypertextual concurrent control of a lisp kernel. *Journal of Visual Languages and Computing*, 3(2):221–236, June 1992.

[SFR92] P. D. Stotts, R. Furuta, and J. C. Ruiz. Hyperdocuments as automata: Trace-based browsing property verification. In *Proceedings of the 1992 European Conference on Hypertext (ECHT92: November 30–December 4, Milan, Italy)*, pages 272–281. ACM Press, New York, 1992.

[vB88] Willem R. van Biljon. Extending Petri nets for specifying man-machine dialogues. *International Journal of Man-Machine Studies*, 28:437–455, 1988.