Trellis: a Formally-defined Hypertextual Basis for Integrating Task and Information*

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

P. David Stotts
Computer Science Department
University of North Carolina
Chapel Hill, NC 27599-3175

1   Introduction

The Trellis project is investigating the structure and semantics of hypertextually-described interaction [SF89a, SF94]. As the work has developed over a number of years, we have broadened our scope from the study of hyperdocuments to encompass *hyperprograms*. A hyperprogram associates user-manipulatable information (the hypertext) with user-directed execution behavior (the process). Consequently a hyperprogram can be said to integrate task with information.

Generalizing, the hyperprogram's execution behavior can be defined by the collective actions of a group of entities, not only the direction of a single user. Such actions can be generated by, and can coordinate the activities of, a collection of human users, computer-based processes, and indeed reflect and are driven by the intentions of the hyperprogram's author. The structural characteristics of a *hyperdocument* permit the specification and documentation of a concurrent protocol and the characteristics of the specification's definition potentially enable the protocol's verification. It is the dynamic characteristics of the *hyperprogram* that permit the prototyping and deployment of the resulting specification.

The Trellis model is formally defined using timed, colored Petri nets. Consequently, hypertexts specified in Trellis describe the traditional hypermedia collection of data objects and relationships among the objects, but also define the process by which the objects are used. In hypertext, "browsing" denotes the traversal of the hypertext, and so we define then hypertext's "browsing semantics" as the process by which the traversal occurs.

As in many modern hypertext research projects, Trellis' notion of "hypertext" is very broad, encompassing all forms of computer representable "media." In addition to static content, such as text and graphics, this definition includes active content, such as video and audio, and procedural content, such as computations and embedded hypertexts.

Trellis implementations are based around a distributed client-server architecture. There is not necessarily a one-to-one correspondence between "user" and "client"—a single user's interface can be formed from the efforts of multiple clients, classes of users may be grouped together and represented by a single client, and clients may be communicating with computer processes rather than human users.

Thus a Trellis document defines an environment in which the constituent objects may be dynamic and in which the state of the document also is changing dynamically. The document's state can be, but is not required to be, related to the actions taken by the human user or users of the environment. However, a central precept is that the state changes are intentional—have, in essence, been "programmed" by an author.

A number of authors have noticed the interesting effect that as information presentation interfaces become more powerful they encompass the abilities of a interface specification system. For example, structured document preparation/presentation systems are being used to specify and implement general user interfaces [BG90, CJ90, EJMM90] (i.e., active documents). Creation of an interface in such an environment can, if appropriate, call on the capabilities of the original application. In particular, the specification of user interfaces using active document systems can be seen to be an authoring task. The experience, techniques, and skills developed over time to organize and write traditional documents can be reused directly into the new application domain.

We have examined similar applications of Trellis in our investigations of protocol specification and prototyping [SF94]. The formal basis of the Petri net permits the development of automatic verification tools. Since Trellis is dynamic, the specifications can be used directly in prototyping the protocol in application. The client/server implementation architecture permits distributed application of the protocol and permits the development of specialized, tuned interfaces that can be run simultaneously with more general development and debugging-oriented interfaces. Since Trellis specifications are interpreted rather than being hard-wired into the application, developers can make modifications "on the fly" as needs arise and can examine how the modifications change the behavior in a continuously-running environment.

Hybrid implementation environments retain the characteristics and capabilities of the parent environment. Consequently the hypertextual basis permits the natural incorporation of system documentation into the implementation—in essence the protocol implementation is self-documenting.

A unifying focus in hypertext research is the identification, definition and development of commonalities with previously orthogonal areas of study. Hypertext has been suggested as an important component of systems supporting the software engineering process [Big88, RA94] and computer-supported cooperative work (CSCW) [SF94, FS94c]. Hypertext can be generalized even further; systems such as Apple's HyperCard [App87, App88] are used to implement general-purpose computer applications.

A characteristic of these investigations is the central nature of *protocol specification* in addressing the domain's requirements. Software engineering protocols include formal and semi-formal

artifacts that provide, for example, requirements specifications. The relationships among these artifacts and the procedures that are followed in their manipulation are in turn defined by higher-level protocols. In the CSCW domain, the coordination of the interactions among humans and between human and computer can be formalized and refined through the expression of protocols [vB88, Hol88].

We have examined the applicability of protocol-centric specification in Trellis within many of these contexts [FS94b, FS94a]. In the applications we have studied we have found that the ability to rapidly prototype, self-document, incrementally modify, develop verification techniques for, and use the authoring metaphor in the creation of the protocol's specification are of significance.

In this chapter we first will give a little more detail about Trellis and its prototype implementations. We will then examine a series of example applications of Trellis illustrating its application in hyperprogram specification. An important opportunity presented by the use of a specification mechanism based on a formally-defined automaton is the ability to consider automated verification of properties of the specification. Section 4 reviews our efforts along those lines. Section 5 concludes the presentation.

2   Trellis: model and prototype architecture

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 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.

2.1   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, also called Petri nets.[1]

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. 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*

---

[1]Readers unfamiliar with basic net theory can find a thorough exposition in the books by Reisig [Rei85] and Peterson [Pet81], and in the survey paper by Murata [Mur89].
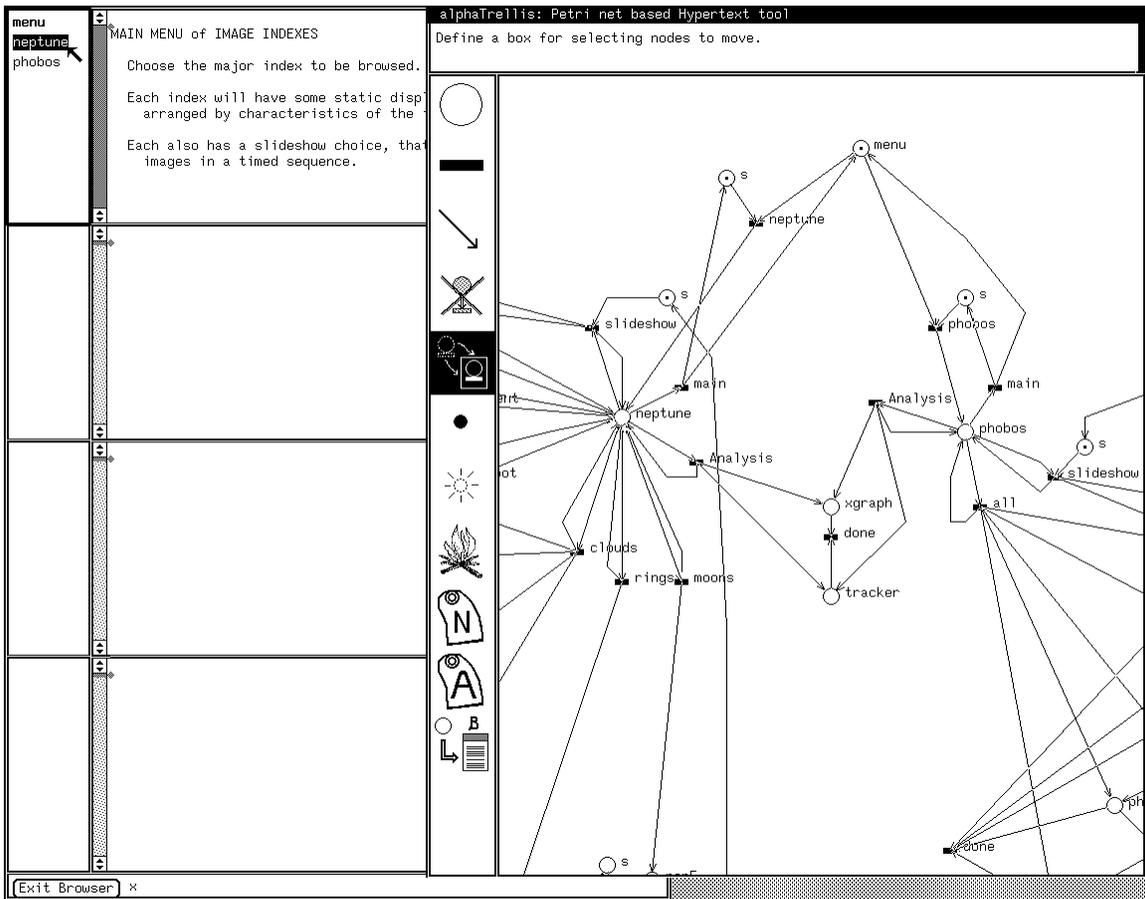
Figure 1: Screen from αTrellis showing an image browsing index.

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

4

arrows show information flow
Clients with arrows out of the model only are "observers,"
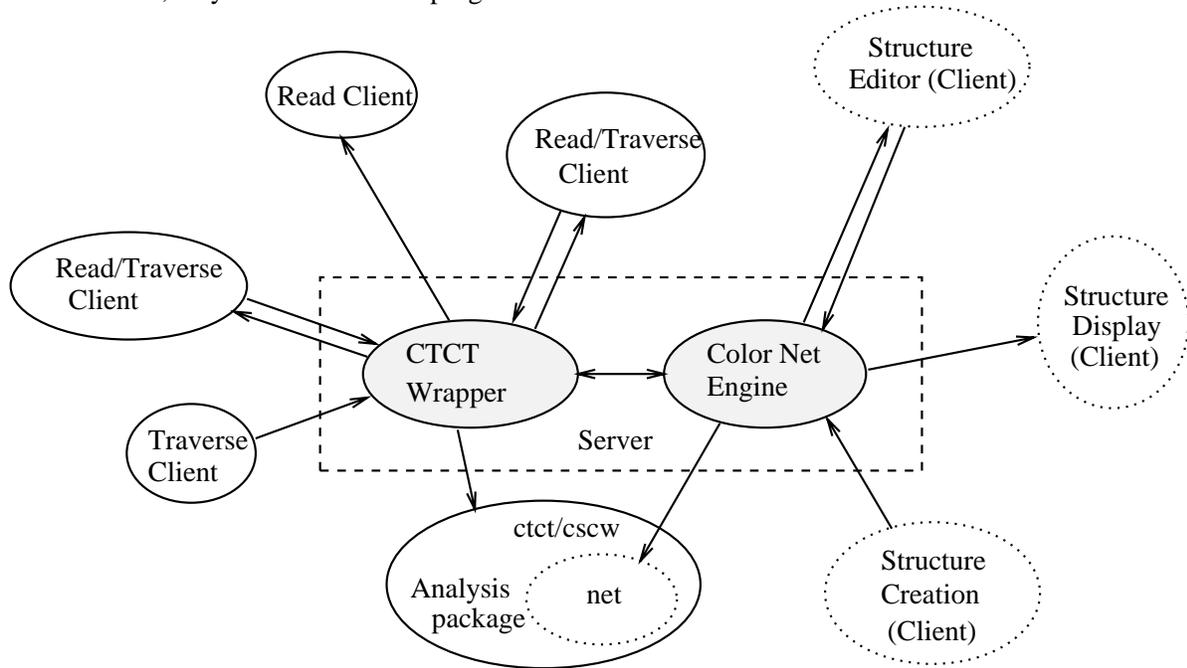that is, they cannot affect the progress of the collaboration

Figure 2: Trellis client/server system architecture.

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.

2.2 Client/server system architecture

Figure 2 shows the high-level structure of a Trellis-based system. 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.

5

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.

3   Hyperprograms

In this section we will examine four applications of Trellis in varying domains. The first three are presented using an older prototype, αTrellis, and show Trellis' application in specifying an image browsing index, in representing the message flow in a parallel programming language, and in simulating solutions to classical problems in process synchronization. The presentations will introduce additional characteristics of Trellis implementations.

The fourth example, cast now in our newer prototype, χTrellis, illustrates the implementation of a protocol for managing a collaborative meeting.

3.1   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
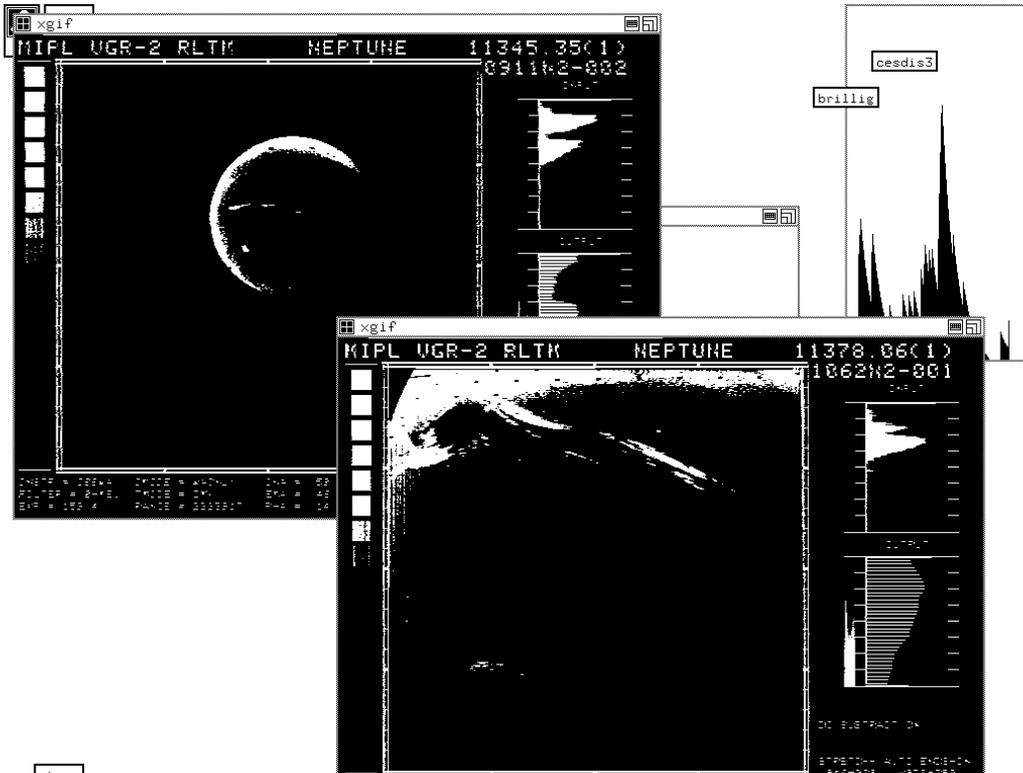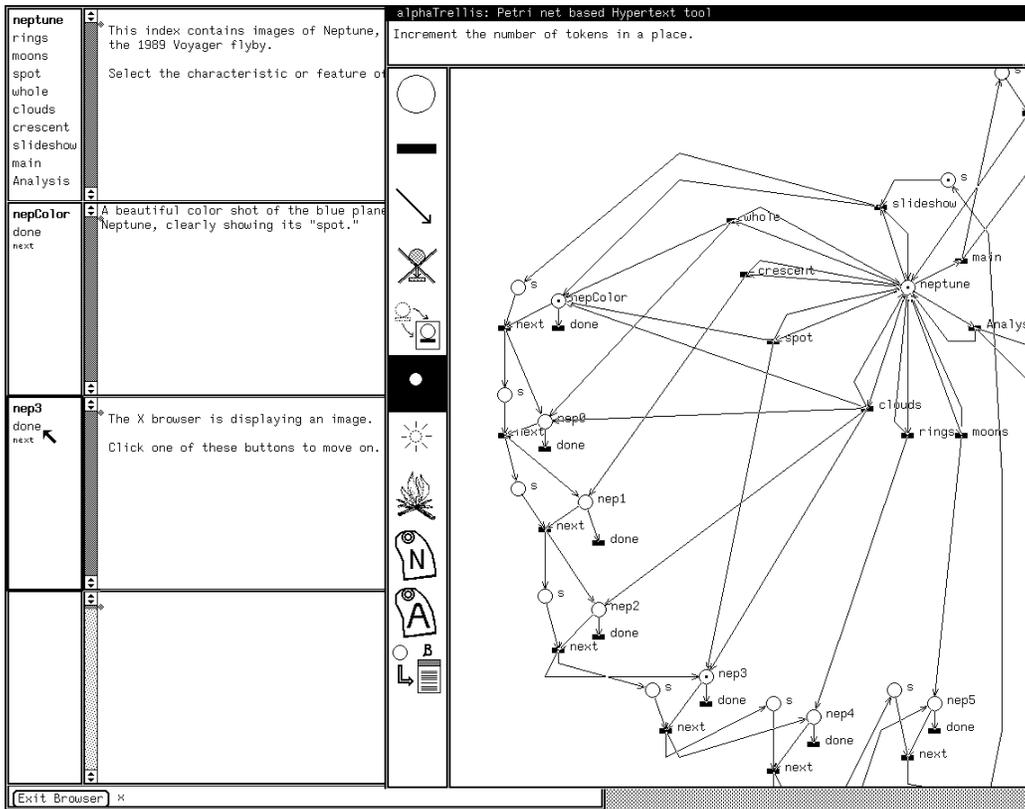
Figure 3: Concurrently displayed Neptune images classified by features.

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 (by firing a transition, for example), 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 bottom frame of Figure 3.
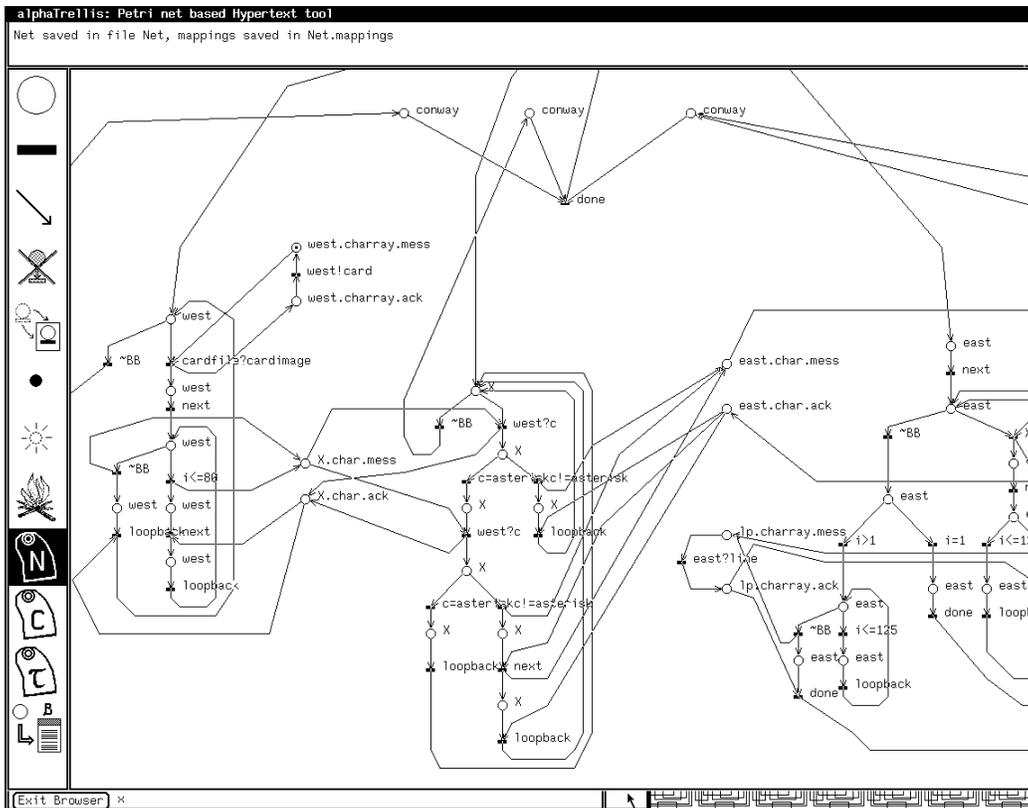
The graphics client does not allow a user to alter the net structure like the editor client does; it does not even allow a user to fire transitions like the text browser does; it just sits and listens, acting when necessary according to its purpose. To initiate some engine activity, a user would have to interact with the interface provided by one of the other two clients.

3.2   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.

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.

3.3 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)     ) )
```
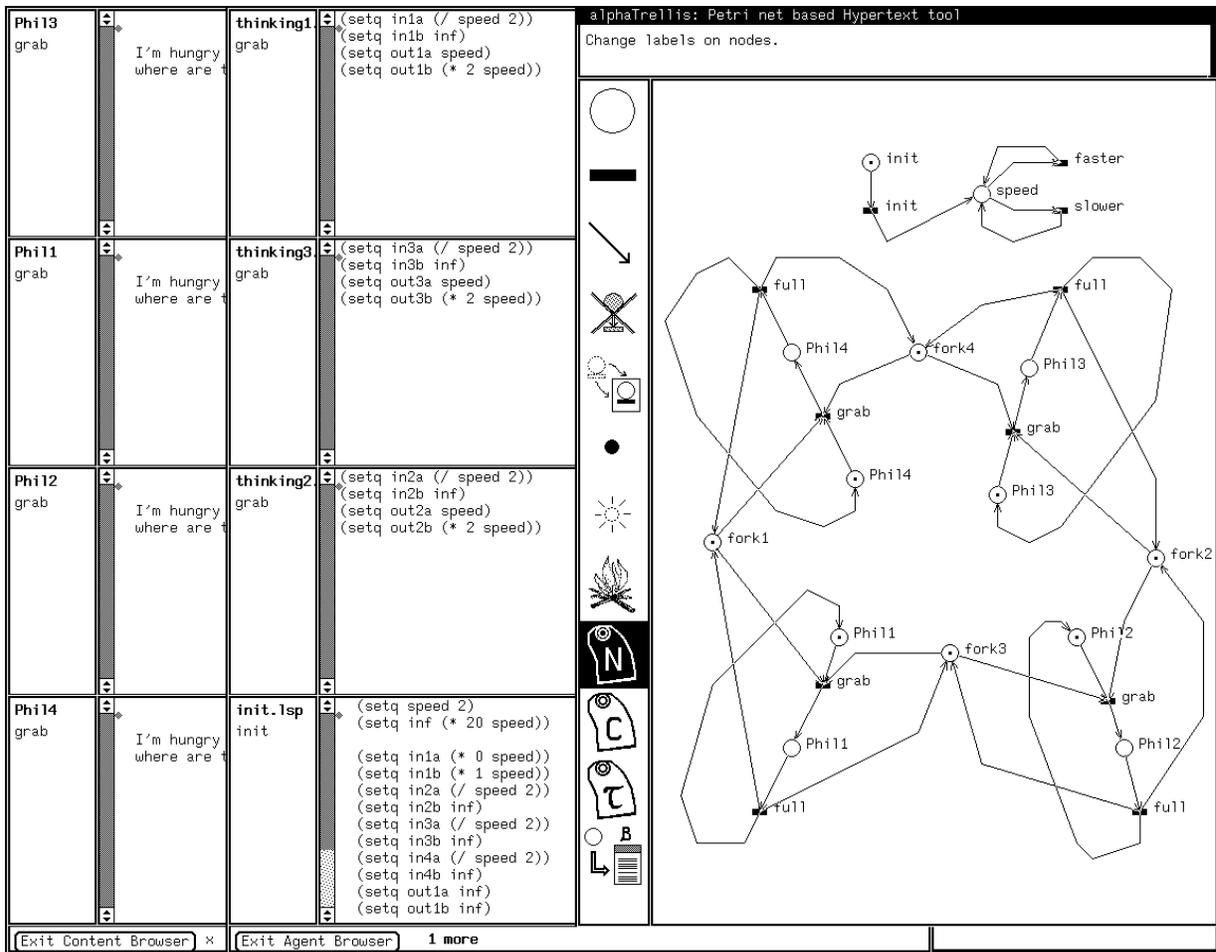
Figure 5: Initial Dining Philosophers screen.

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 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
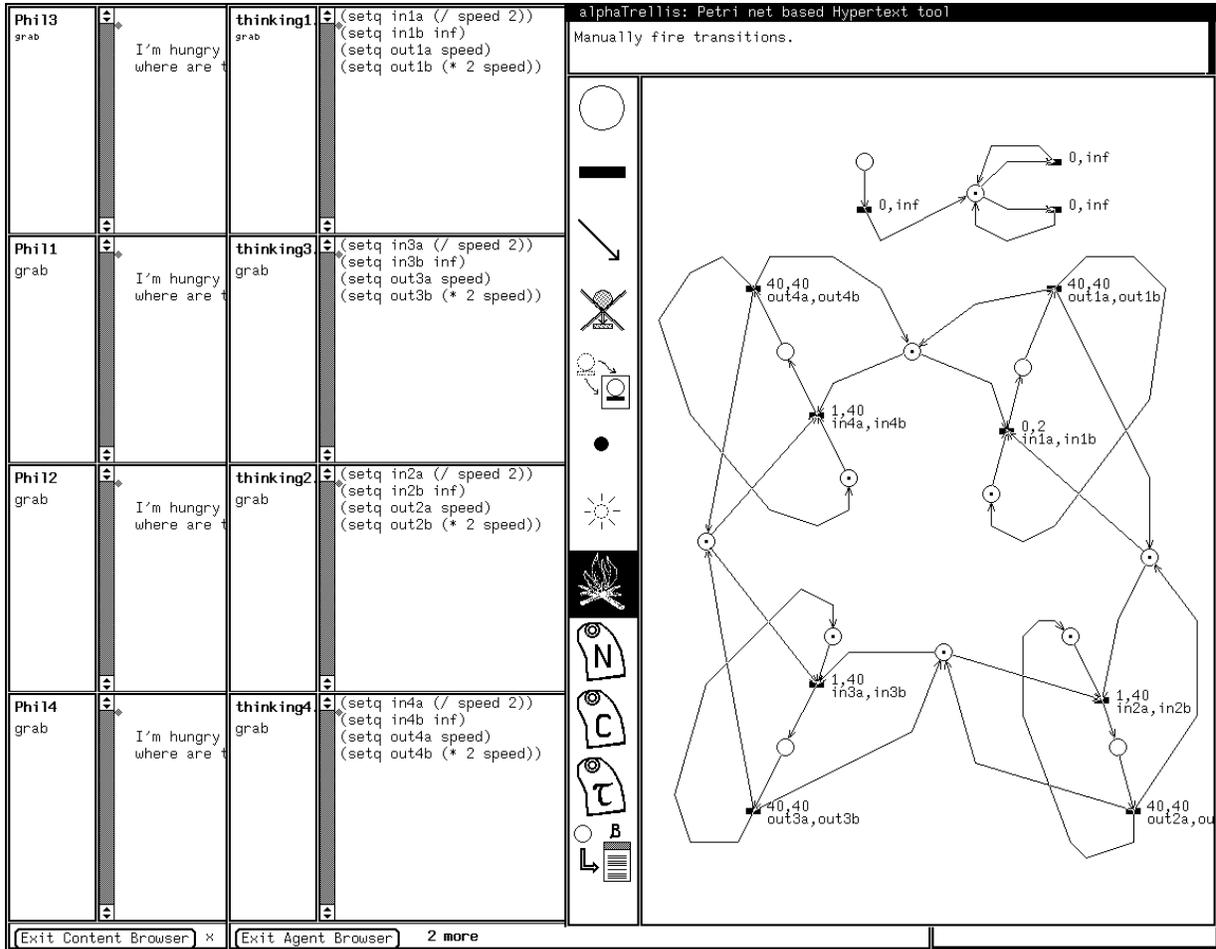
11

Figure 6: Timings after firing Init button.

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.

3.4 χTrellis: Collaboration protocols and hypermedia

The preceding examples introduced the αTrellis prototype in some detail because our early experiments in modeling have been performed with it. Our current efforts are shifting to a newer prototype called χTrellis, which operates in the X-windows environment.

Figure 7 shows a snapshot of χTrellis, illustrating the use of Trellis specifications for directing a meeting among several participants. The editor client that shows the related net is in the upper left corner of the screen and client-displays corresponding to the individual participants in the meeting are found around the periphery. More detailed information about this implementation can be found in a separate paper [FS94c], so the discussion here will just touch on some general points.
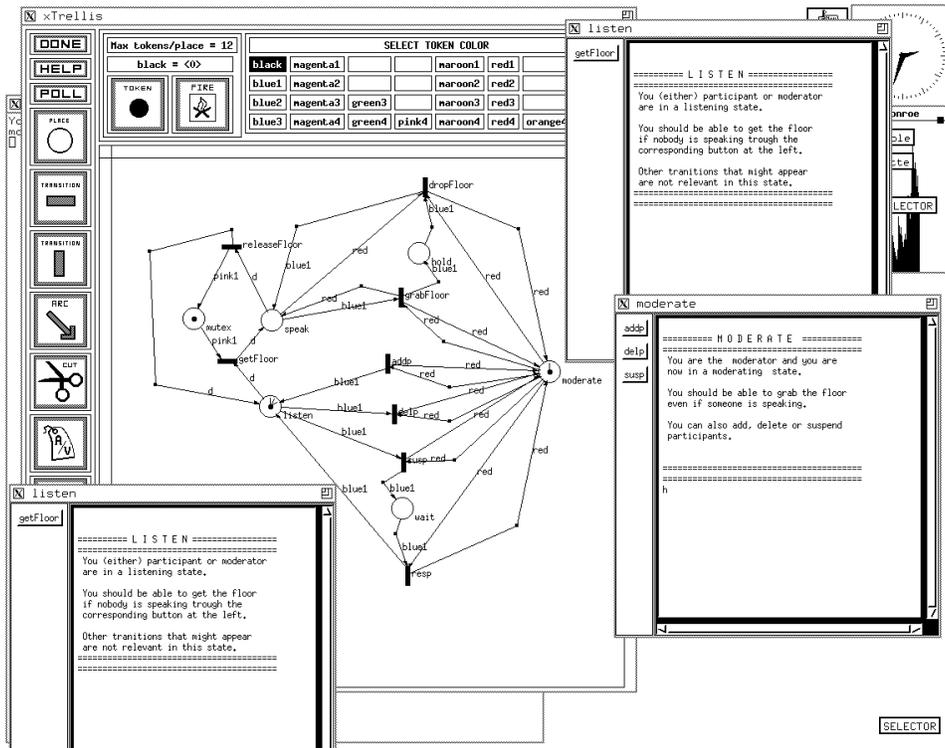
12

Figure 7: Meeting protocol implementation

Colored tokens are used in this specification to distinguish classes of participants. The specific classes of interest that defined in this example are the discussion's moderator and its participants (in addition, some internal classes are defined to implement higher-level structures).

The Trellis specification, shown in the upper left hand window in the figure, is designed to limit some operations to the moderator and to permit the moderator to override participants if desired. The remaining windows in the display represent the view seen by a meeting participants and by the moderator. In actual use, display of these documents would be distributed over multiple, separated, workstations and would serve as "traffic control" for an ongoing meeting.

As the more detailed paper explains, the specification can be modified flexibly to reflect different meeting policies. Examples include creating a separate class of token for each participant (rather than an anonymous pool of participant-class tokens as in the initial example), modifications and additions of moderator-limited actions, and permitting the moderator to exchange roles with a participant. Figure 8 shows a modified specification reflecting these changes. Since protocols are interpreted, dynamic modifications can be made, either by a person editing the specification "on the fly," or through the actions of Trellis dynamic adaptation agents, which could modify the effective appearance of the protocol based on "observation" of the meeting participants' actions.

Figure 8: Meeting protocol implementation

4   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
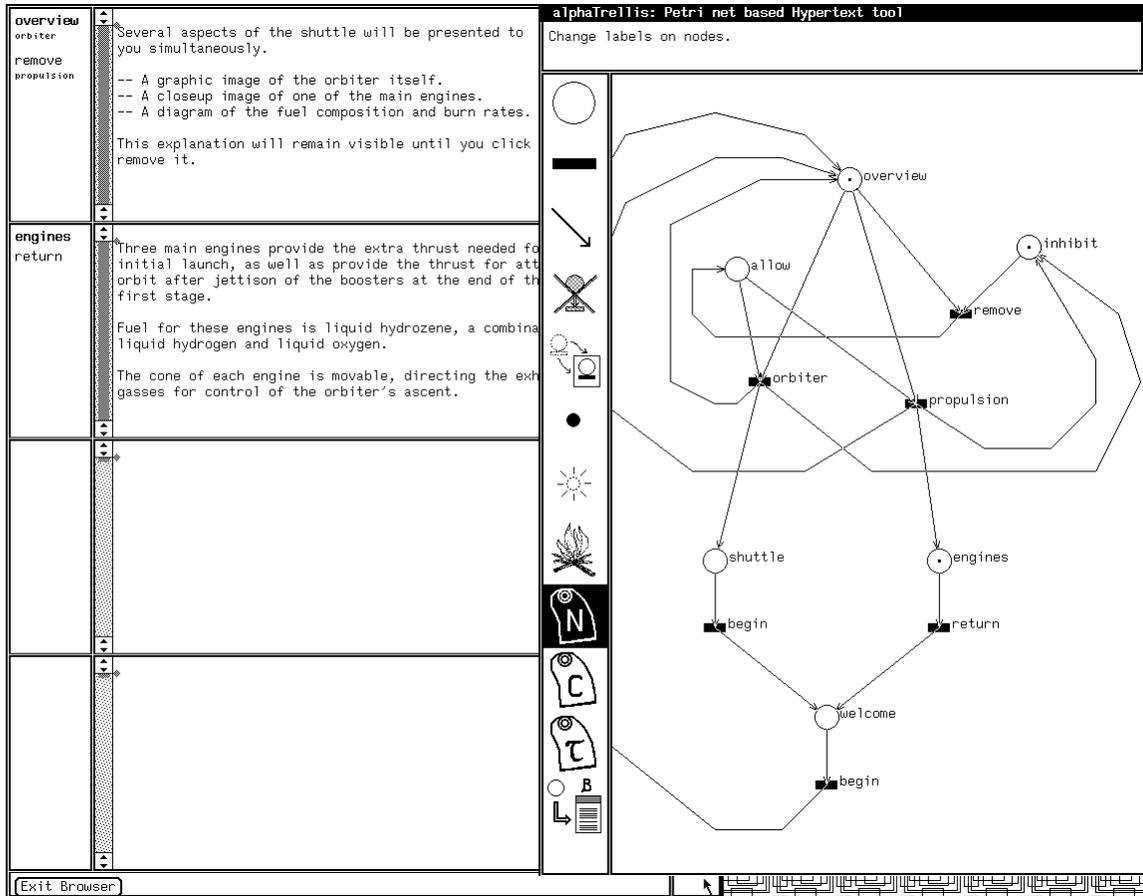
Figure 9: Small Trellis structure with programmed browsing behavior

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 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 useful state machine from the *coverability graph* explained in an earlier Trellis paper [SF89a].

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 (and eventually to IMP/ACT) models. The Trellis document shown in Figure 9 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?

15

```
        |= 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?

```
        |= 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.

5 Conclusion

The emergence of hypertext as an authoring-based prototyping mechanism suggests a modification of the view of the role of hypertext in CSCW as one in a set of independent technologies tied together into a collective environment by some meta-structuring mechanism. In our work, the hypertext system retains its traditional job of information structuring but also gains the job of process structuring. In the added role, the hypertext structure adopts the role of the environmental meta-structuring mechanism and can also fill the role of implementation mechanism for certain of the specific tools found in the environment.

The Trellis net-based representation provides a means for specifying a protocol that can be used directly for verification, training, and simulation. When the hypertextual basis is dynamic, it also provides the means for rapid prototyping of the protocol. On the other hand, the visual presentation of the net requires attention be paid to the complexity of the image, requiring investigation of structuring through abstraction and hierarchy.

We close by noting that the Trellis implementation decision to interpret the underlying net suggests further experimentation with an incremental methodology in protocol development *and* in incremental protocol verification. As prototyping proceeds, more sophisticated behaviors are added, refined, and verified. There is a nice symmetry between this view of protocol development and the world-view of a hypertext as a dynamic web, growing and developing as new paths are encountered and envisioned.

References

[App87]   Apple Computer, Inc. *HyperCard User's Guide*. Apple Computer, Inc., 1987.

[App88]   Apple Computer, Inc. *HyperCard Script Language Guide: The HyperTalk Language*. Addison-Wesley, 1988.

[BG90]   Eric A. Bier and Aaron Goodisman. Documents as user interfaces. In Richard Furuta, editor, *EP90*, pages 249–262. Cambridge University Press, September 1990. Proceedings of the International Conference on Electronic Publishing, Document Manipulation, and Typography, Gaithersburg, Maryland, September 1990.

[Big88]   James Bigelow. Hypertext and CASE. *IEEE Software*, 5(2):23–27, March 1988.

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

[CJ90]   Gil C. Cruz and Thomas H. Judd. The role of a descriptive markup language in the creation of interactive multimedia documents for customized electronic delivery. In Richard Furuta, editor, *EP90*, pages 277–290. Cambridge University Press, September 1990. Proceedings of the International Conference on Electronic Publishing, Document Manipulation, and Typography, Gaithersburg, Maryland, September 1990.

[EJMM90] Paul M. English, Ethan S. Jacobson, Robert A. Morris, Kimbo B. Mundy, Stephen D. Pelletier, Thomas A. Polucci, and H. David Scarbro. An extensible, object-oriented system for active documents. In Richard Furuta, editor, *EP90*, pages 263–276. Cambridge University Press, September 1990. Proceedings of the International Conference on Electronic Publishing, Document Manipulation, and Typography, Gaithersburg, Maryland, September 1990.

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

[FS94a]   Richard Furuta and P. David Stotts. Interpreted collaboration protocols and their use in groupware prototyping. In *Proceedings of Computer Supported Cooperative Work '94*. Association for Computing Machinery, October 1994. To appear.

[FS94b]   Richard Furuta and P. David Stotts. A hypermedia basis for the specification, documentation, verification, and prototyping of concurrent protocols. Technical Report TAMU-HRL 94-003, Texas A&M University, Hypertext Research Lab, June 1994.

[FS94c]   Richard Furuta and P. David Stotts. Interpreted collaboration protocols and their use in groupware prototyping, 1994. Internal report.

[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.

[Mur89]   Tadao Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.

[Pet81]   James L. Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice-Hall, Inc., 1981.

[RA94]   Tina Roth and Peter Aiken. Hypertext support for software development: A retrospective assessment, 1994. Internal report.

[Rei85]   Wolfgang Reisig. *Petri Nets: An Introduction*. Springer-Verlag, 1985.

[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.

[SF94]   P. David Stotts and Richard Furuta. Modeling and prototyping collaborative software processes. In Shimon Y. Nof, editor, *Information and Collaboration Models of Integration*, pages 365–390. Kluwer Academic Publishers, June 1994. Based on the NATO Advanced Research Workshop on Integration: Information and Collaboration Models, Il Crocco, Italy, June 6–11, 1993. Also available as Technical Report TR93-020, Computer Science Collaboratory, Univ. of North Carolina at Chapel Hill, 1993; and as Tech Report TAMU-HRL 93-006, Hypermedia Research Laboratory, Texas A&M University, July 1993.

[SFR92]   P. David Stotts, Richard Furuta, and J. Cyrano Ruiz. Hyperdocuments as automata: Trace-based browsing property verification. In D. Lucarella, J. Nanard, M. Nanard, and P. Paolini, editors, *Proceedings of the ACM Conference on Hypertext (ECHT '92)*, pages 272–281. ACM Press, 1992.

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