Modeling and Prototyping Collaborative Software Processes

P. David Stotts
Richard Furuta

Hypermedia Research Lab

JULY 1993

TAMU-HRL 93-006

Modeling and Prototyping Collaborative Software Processes

P. David Stotts
Richard Furuta

July 1993
TAMU-HRL 93-006

MODELING AND PROTOTYPING COLLABORATIVE SOFTWARE PROCESSES*

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

The correct and timely creation of systems for coordination of group work depends on the ability to express, analyze, and experiment with protocols for managing multiple work threads. We present an evolution of the Trellis model that provides a formal basis for prototyping the coordination structure of a collaboration system. Like its predecessor, the new Trellis model has the nicely exploitable duality of being both graph formalism and parallel automaton. The automaton semantics provide dynamic information about the interactions of agents in a collaboration; the graph structure forms the basis for the static link structure of a hyperdocument. We give several analysis techniques for the model, and demonstrate its use by expressing the interaction structure of some common forms of collaborative system.

1 Introduction

The Trellis project [SF89, SF90b] 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 software process models. Our design work has been guided since the early projects by a *simplicity-over-all* principle; this means we develop as simple a model as practical at first, and study how far towards a general solution it will take us before we add more capability, or "features" to the formalism. As a result, our interaction models strike a balance between fully-programmable/non-analyzable (like Apple's Hypercard product) and fully-analyzable/non-programmable (static directed graphs).

In this report 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*.

*This work is partially supported by the National Science Foundation under grant numbers IRI–9007746 and IRI–9015439, and by the Software Engineering Research Center (University of Florida and Purdue University).

When using Trellis in a CSCW context, the net structure serves several functions with a single notational framework: its structures shared applications; its synchronizes loosely coupled parallel executing applications; it provides a repository for application information and historical data; and it provides mechanisms for joint decision making and action. Semantic nets and link typing may be as useful for pure hypertext description. Object-based message passing languages are probably as appropriate for expressing parallel threads. Production systems are probably as useful for specifying group interactions. However, Trellis provides a single formalism for all these aspects of a collaboration support framework.

Due to the heavy interpretation as hypertext, Trellis hyperprograms are especially useful for processes in which human direction is an important aspect of the control flow. An example is the software development process we discuss in section 6. Such computations are referred to as being *enacted*, rather than as being executed, to distinguish the major role human input and human decisions (and for CSCW, human interactions) have in the unfolding of the actions described in the hyperprogram.

2 Formal definitions

The Trellis project is an ongoing effort to create interactive systems that have a formal basis and that support analytical techniques. The first such effort was a hypertext model [SF89], with a followup framework for highly-interactive time-based programming (termed *temporal hyperprogramming* [SF90b]). The model we present here is an extension of these earlier designs that explicitly distinguishes the various agents acting within a linked structure, and that provide an analyzable mechanism with which agents may exchange data. This new model basically follows the Trellis framework of annotating a form of *place/transition net (PT net)*, and using both graph analysis and state-space analysis to exploit the naturally-dual formalism.

The following short section outlines some of the basic concepts and terminology of PT nets, their structure, and common behaviors; readers already familiar with these notions may choose to skip it. Following that we introduce the group- and timing-specific net definitions, and finally the model of collaboration structures based on these nets.

2.1 Net theory basics

The notation used here is taken from Reisig [Rei85]. For the interested formalist, Murata [Mur89] gives a broad and thorough introduction to net theory and modeling applications. We present here just the basics required for understanding our application of this theory.

A PT net is a bipartite graph with some associated execution semantics. The two types of nodes in a net are termed *places*, represented visually as circles, and *transitions*, represented visually as bars. Activity in the net is denoted with *tokens*, drawn as dots in the places. Two nodes of the same type may not be joined by an arc. Given the arc structure of a net, the set of inputs to a node n is termed the *preset* of n, denoted $\bullet n$, and the set of output nodes is termed the *postset* of n, denoted $n\bullet$. Figure 1 shows the common representation of these PT net components (we will discuss the interpretation of this figure later); the varying patterns on tokens in this diagram represent *colors*, a mechanism for class typing discussed in detail later.

One widely used form of PT net is the *Petri net*.[1] A transition t in a Petri net is said to be

[1] We will use the general term *PT net* to describe the place and transition net syntax that is common to many

enabled if each place in $\bullet t$ is *marked*, i.e., contains at least one token. Once enabled, a transition t may *fire*, causing a token to be removed from each place of $\bullet t$ and depositing one token in each place of $t\bullet$. A *net marking*, or *net state*, is a vector of integers, telling how many tokens reside in each place. Execution of a Petri net begins with some initial marking, and continues through a sequence of state changes caused by choosing an enabled transition in the current state and firing it to get a new state. Execution certainly terminates if a state is reached in which no transitions are enabled, but it may also be defined to terminate with any marking that has some special significance to the user of the net.

2.2 Colored timed net

The Trellis model is based primarily on a synchronously executed, transition-timed Petri net as the structure of a hyperprogram. For use in CSCW, we have employed a form of net model known generically as *high-level* nets. High-level nets have been introduced in several forms by different researchers, including predicate-transition nets [GL81], colored Petri nets [Jen81], and nets with individual tokens [Rei83].

We present our ideas in a hybrid notation. We will use the Jensen's terminology of colored nets, but the simplified syntax presented by Murata in his high-level net summary [Mur89]. All forms of high-level nets can be translated into one another, and are thus equivalent, but the simple syntax we use creates explanations that are more clear. We will discuss these other syntaxes after the examples.

In colored nets, tokens have type (color) and may carry data structure. A token of one color is distinguishable from a token of another color; within a color class, however, individual tokens cannot be distinguished from one another. The timing of the original Trellis model has been retained and combined with color to produce this model:

Definition 1 *Colored timed net structure*

A colored timed net structure CTN *is a 5-tuple,* $CTN = < S,\ T,\ F,\ \kappa,\ \tau >$ *in which*

$S = \{p_1, \ldots,\ p_n\}$ *is a finite set of* places *with* $n \geq 0$*,*

$T = \{t_1, \ldots,\ t_m\}$ *is a finite set of* transitions *with* $m \geq 0$*, and* $S \cap T = \emptyset$*,*

$F \subseteq (S \times T) \cup (T \times S)$ *is the* flow relation*, a mapping representing arcs between places and transitions.*

$\kappa : \{\kappa_1, \ldots,\ \kappa_r\}$ *is a finite set of* colors *for typing tokens, where each color is a function* $\kappa_i : S \rightarrow \{0, 1, 2, ...\}$*;*

$\tau : T \rightarrow \{0, 1, 2, ...\} \times \{\infty, 0, 1, 2, ...\}$ *is a function mapping each transition to a pair of values termed* release time *and* maximum latency *respectively. For any transition* $t \in T$*, we write* $\tau(t) = (\tau_t^r, \tau_t^m)$ *and we require that* $\tau_t^r \leq \tau_t^m$*.*

In this model, we have simplified the notation used in Reisig [Rei85] by assuming that the weight on each arc is 1, and that the token capacity of each place is unbounded. A net marking is a vector of token counts, with each token count being a vector of color counts; a marking provides a snapshot, at some point during execution, of how many tokens of each color reside in each place.

forms of concurrent computation model. We reserve the term *Petri net* to describe a form of PT net with a specific (and familiar) execution semantics.

For a transition $t \in T$, its release time represents the number of time units that must pass once t is enabled before it can be fired; its maximum latency represents the number of time units that may pass after t is enabled before it fires automatically.

This temporal structure is very similar to that of Merlin's *Time Petri nets* [Mer74, MF76], with a few differences. The two time values for each transition here are integers, whereas Merlin used reals. We also have a need for the maximum latency to possibly be unbounded, using the special designation ∞ which is not in Merlin's model. Finally, times are not thought of as durations for transition firing in Trellis. Transitions are still abstractly considered to fire instantaneously, like the clicking of a button in a hypertext interface. Time values in Trellis are thought of as defining ranges for the *availability* of an event.

2.3 Collaboration protocol structure (CPS)

The timed Trellis model of hypertext uses the structure and execution rules of timed Petri nets to specify both the linked form and the browsing semantics of a hypertext. This logical structure then is interpreted through a layer of indirection to arrive at a displayed form for reader consumption and interaction. Hypertext content and linked structure are thus effectively separated by the timed Trellis model.

Definition 2 *Collaboration protocol structure*

A collaboration protocol structure *is* $CPS = < CTN,\ M_0,\ C,\ W,\ B,\ P_l,\ P_d >$ *in which*

$CTN = < S,\ T,\ F,\ \kappa,\ \tau >$ *is a colored timed net,*

$M_0 : S \rightarrow < c_1, c_2, ..., c_r >$ *is an* initial marking *(or* initial state*) for* CTN*, where* $r = |\kappa|$ *and* $\forall p \in S, M_0(p)_i = c_i = \kappa_i(p)$*,*

C *is a set of document* contents*,*

W *is a set of* windows*,*

B *is a set of* buttons*,*

P_l *is a logical projection for the document,*

P_d *is a display projection for the document.*

A CPS consists of a CTN representing the document's linked structure, a marking to tell how many tokens of each color start in each net place, several sets of human-consumable components (*contents*, *windows*, and *buttons*), and two collections of mappings, termed *projections*, between the CTN, the human-consumables, and the display mechanisms. A window from W is a logically distinct locus of information. A button from B is an action that causes the current display to change in a specified way. Content elements from C can be many things: text, graphics, tables, bit maps, executable code, sound, or, most importantly, *another CPS*.

A logical projection P_l provides mappings from components of a CTN to the human-consumable portions of a group work environment as mentioned above. Each place in the CTN has a content element from C mapped to it, as well as an element of W for the abstract display of the content. Each transition in the net has a logical button from B associated with it. The display projection P_d is a set of mappings that take the logical components and produce tangible representations, such as screen layouts, sound generation, video, etc. P_d determines how things like text and buttons are

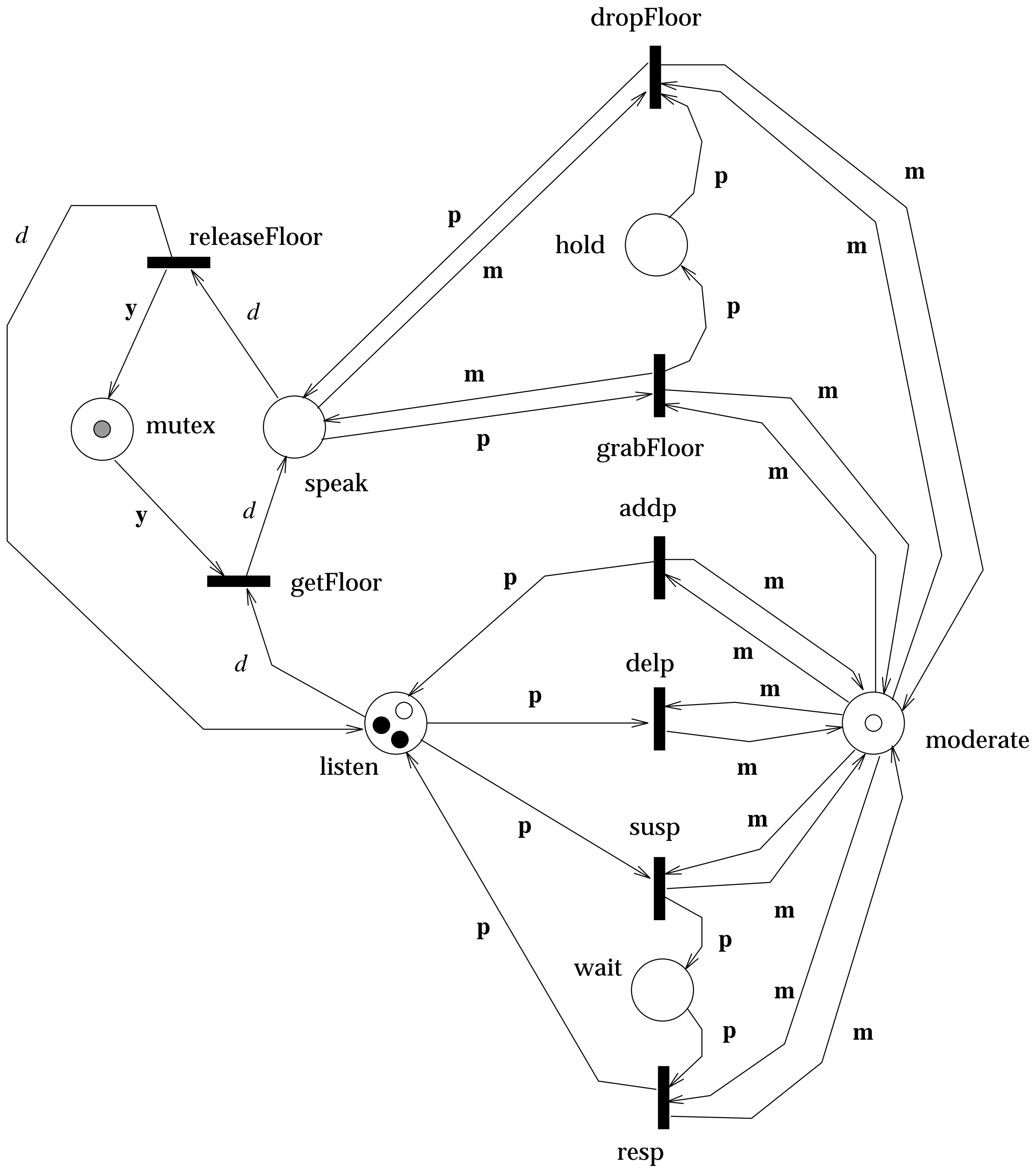

Figure 1: CPS for a simple meeting protocol.

visibly displayed, *e.g.*, whether a user selects a link from a side menu or from highlighted words (or icons) within the content display.

The net marking M_0 enables a CPS to represent both the logical structure of a collaboration and the current state of inter-activity within it. Together with the execution rules of the CTN, every marking is a characterization of the possible paths in a collaboration from the browsing point it represents. Different browsing patterns (for, say, different classes of reader) can then be enforced on a single CPS simply by choosing appropriate different initial markings.

2.4 Execution rules for a CPS

The execution behavior of a CTN provides an interpretation of the collaborators' experiences when interacting under the control of a CPS. As in the original Trellis model, a token in a place p indicates that the contents of the place $C_l(p)$ are displayed for viewing (or editing, or some other interaction). Content elements come into and go out of view (or begin and end execution) as tokens move through the net. Transitions are fired by selecting logical hypertext buttons. When a transition t is fireable in the timed Petri net, its logical button $B_l(t)$ is displayed in some selectable area of the screen, such as on a (highlighted) word in a text section, or in a separate button menu.

The general execution behavior of the CTN in a CPS requires pattern matching to be done on all arc expressions that are inputs to a transition. The transition is enabled if there is one or more consistent color substitutions for the expressions. When the transition fires, one of the valid substitutions is chosen, the proper color tokens are removed from the input places, and output tokens are produced according to the substitution and the expressions on the output arcs.

Rather than being excessively formal, we will explain CTN execution behavior informally thorough the examples in the next section. We will explain the projections and the interpretation of net annotations during execution in section 5 on prototyping.

3 CTN Examples

In the next few sections we present the basic functions of a CPS through an extended example. Following this illustration, we describe the methods we are using to analyze and verify the behavior encoded in a CPS. After analysis, we explain a major application for CPS—enacting and improving the process of software system development.

3.1 Example: Simple moderated meeting

Figure 1 shows a CPS that encodes a simple moderated meeting. To enhance the clarity of this example, we have made some simplifying assumptions about the actions in such a meeting; we discuss more realistic complexities following an initial explanation.

We envision a meeting with two classes of agent: *participants*, and a *moderator* (who may also act as a participant). Participants can be in either of two states: listening, or speaking. When listening, they can request and possibly obtain the floor to speak; when speaking, they can release the floor, to return to listening. The moderator has more extensive abilities. In addition to acting as a participant, the moderator can: add or delete participants in the meeting; suspend participants for a time, and return them to a meeting (we presume that suspension is different from being deleted,

as something like a history would be kept for suspended participants); grab the floor, preempting the current speaker, and drop the floor, returning the preempted participant to speaking.

In the CPS shown, we have represented the participants all with one color; that is, we have used color to represent the entire class rather than individuals. Consequently, the net is simpler for an initial discussion, but no participant can be distinguished from another. We will remedy this shortly. We have assigned a second color for the single moderator, and we have used a third color for a token providing mutual exclusion of potential speakers.

Color constants

In this simple protocol, the moderator is fixed for the duration of the meeting (we will explain a more complicated alternative to this, as well, following). To understand the notation on the net, consider the action "add participant" that the moderator can perform. This is represented in the net as the transition labeled "addp". There is one input arc to this transition, labeled **m** coming from place "moderator". The label **m** in boldface indicates a color *constant* which we have selected for the moderator token. The "addp" transition has two output arcs: one labeled **p** to place "listen", and another labeled **m** back to place "moderator". As before, **p** is a color constant representing the participant class.

When a token of color **m** is present in place "moderator", the operation can be invoked (*i.e.*, the moderator can invoke it whenever desired... no other preconditions exist). Firing the transition consumes the **m** colored token, but it also places one back into the moderator place (*i.e.*, the moderator does not give up his role by adding a new participant). Firing also places a new **p** colored token into place "listen", thereby increasing the number of participants by one.

Color variables

So far we have seen behavior that is accomplished with color constants indicated on arcs. However, the real power of the CPS notation comes in allowing color *variables* to appear on arcs. Such a structure appears in figure 1 on the left side, in the net region containing the "getFloor" and "releaseFloor" operations. Note that an **m** colored token is located in place "listen" along with all the **p** colored tokens. This, along with color variables on arcs, implements our claim that the moderator should be able to act as a participant also.

The arc leading from place "listen" into transition "getFloor" is labeled with the expression d, where the italics indicates a color variable. The arc leading out of "getFloor" to place "speak" is also labeled with d. Note that the arc leading to "getFloor" from place "mutex" is annotated with the color constant **y**. The meaning of this net fragment, then, allows "getFloor" to fire with some variability in its input token colors—not just with specific input colors, as in the previous example. Transition "getFloor" may fire if there is specifically a **y** color token in place "mutex" (*i.e.*, if there is no one currently speaking), and if there is some token of *any* color (call it d) in place "listen". When it fires, the **y** color token is consumed from place "mutex"; in addition, a token of whatever color d stands for is removed from "listen", and a token *of that same color* is deposited into place "speak". This means that the single operation "getFloor" may be used to move either an **m** color token or a **p** color token into place "speak".

The same sort of color variable behavior controls the firing of transition "releaseFloor" when someone wished to stop speaking.

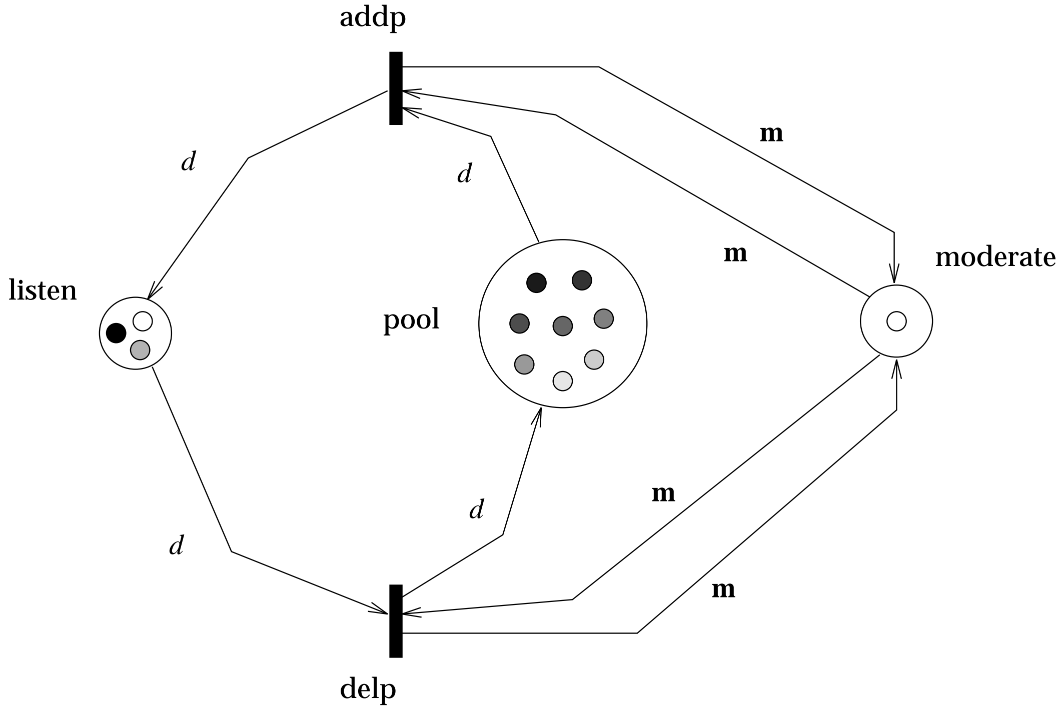

Figure 2: Detail for participant allocation.

3.2 Example: Distinguishing participants in a meeting

Let's consider other CPS structures that add more detail to the simple protocol previously discussed.

One reasonable change is to allow a different color for every meeting participant. This can be done by creating a finite pool of differently colored tokens that is held in reserve. When a new participant is to be added, a "new" color is allocated from the pool and added to the meeting; when a participant is removed from the meeting, the color is returned to the pool.

This alteration is depicted in the CPS fragment of figure 2. In addition to the pool of colored tokens, the net shown in figure 1 has been changed to include varying token colors in place "listen". Also, arcs leading from the moderator operations to place "listen" are now labeled with the variable expression d instead of with the constant **p**.

It is important for analysis purposes (explained later) that the pool of potential participants be finite. That is, the CPS must specify all colors that might be used by meeting participants, and no truly new color can be injected into the net as a whole during execution. However, the finite number of participants can be arbitrarily large. This limit presents a practical problem only if the meeting protocol to be modeled must allow an unbounded number of participants. Note that the simple example we presented first, in which one color was used for the entire class, does allows an unbounded number of (indistinguishable) participants. Whether or not a truly unbounded number of participants is a reasonable requirement for a CSCW tool is a point for separate discussion.

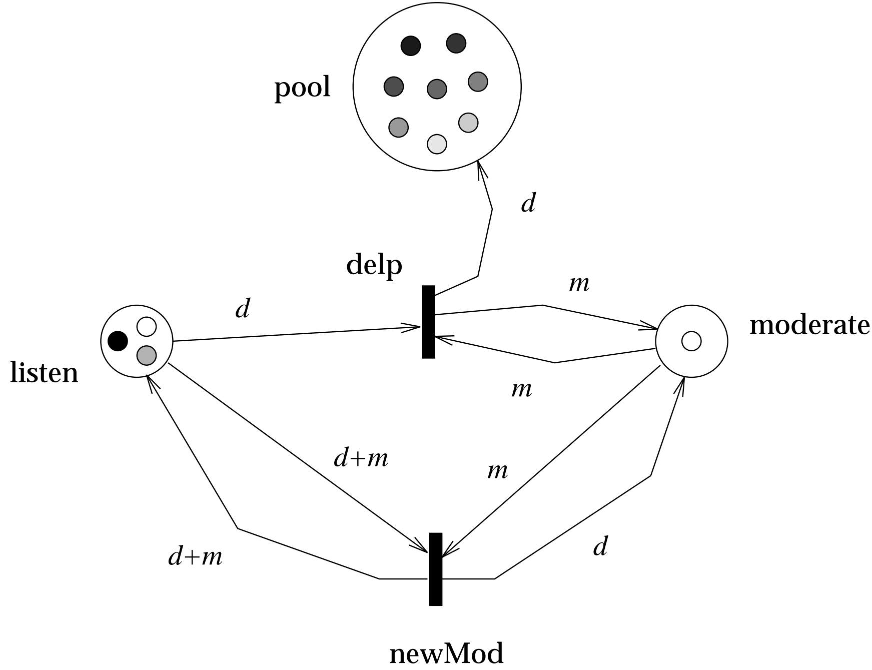

Figure 3: Detail for changing the moderator.

3.3 Example: Changing the moderator of a meeting

Another practical addition to our meeting protocol is the ability to change moderators while the meeting is in progress. For this example, we will build on the one from figure 2 with the finite pool of participants. We continue to assume that each participant, moderator or otherwise, is assigned a unique token color.

Figure 3 shows more CPS details for moderator swapping. In this fragment, we have altered the labels on arcs between the "moderator" place and the previously existing operations (like "addp" and "delp") to have the variable expression m. Labeled in this way, the moderator is not fixed as always being the constant color **m** as before, but instead can be any color; having m on all arcs between place "moderator" and operations like "addp" specifies that execution of such an operation must maintain whatever color m represents (*i.e.*, the moderator cannot change simply by executing "addp" and the other previously discussed meeting control functions).

We have added another operation, "newMod", to specifically perform moderator swapping. The arc leading into "newMod" from place "moderator" is labeled with the expression m, and the arc leading into the transition from place "listen" is labeled $d + m$. This shows that the "newMod" function can only be invoked if the "listen" place contains both a participant with the same color as the moderator, and another participant with a *different* color from the moderator (we assume no aliasing in color substitutions). When fired, the "newMod" transition leaves the token counts in "listen" unchanged, but it takes whatever color was in "moderator" (represented by variable m) and replaces it with a token of whatever color is represented by d. Since we know the value of d is different from the value of m (the "no aliasing" assumption), we know that the moderator has changed. Neither participant leaves the meeting—they just exchange capabilities. Also note that the new moderator color is drawn not from the pool, but from the actual participants found in

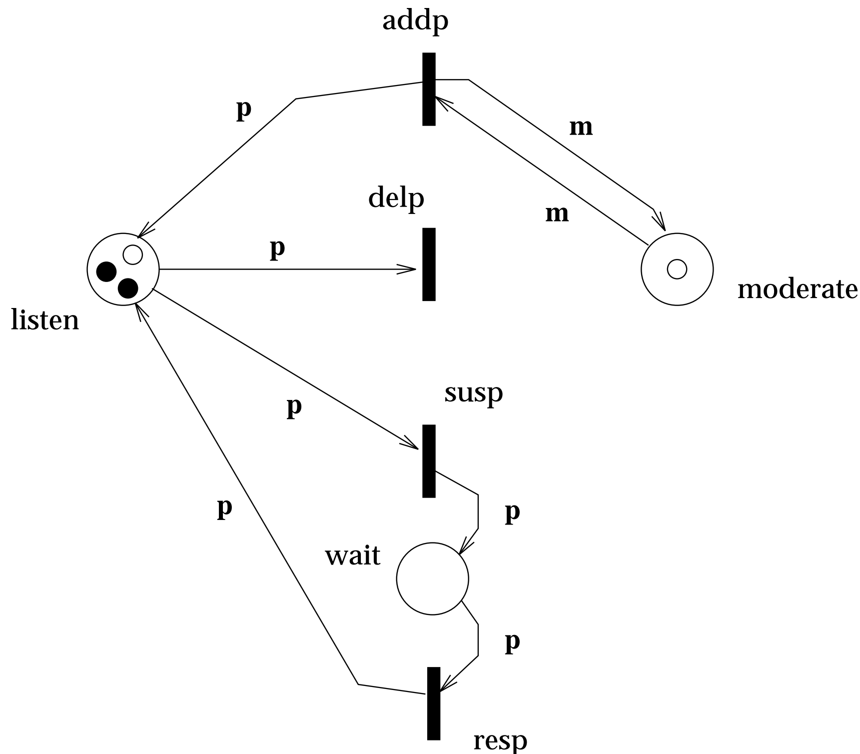

Figure 4: No permission required to leave.

place "listen". Finally, as written, the CPS allow a moderator to swap only with someone who is listening—a speaker cannot become the new moderator without first releasing the floor.

3.4 Example: Other miscellaneous behaviors

In this section, we return to the simple protocol of figure 1 to illustrate some other behaviors that shed light on the CPS method of specification. Though we present them in the context of the protocol with mostly color constants, the basic interactions will translate into the more complex CPS examples as well.

Note that in the initial protocol, a participant in essence requires the permission of the moderator to leave a meeting, or to be suspended for later rejoining. This condition exists because the moderator place is required to contain a token for all such operations to take place.

In the example, though, the moderator place *always* contains a token (a condition that can be verified in several ways, including the method we will present following). The moderator is "always home" so to speak, and no net structure is present that would ever cause permission to leave not to be granted. The behavior, then, of our initial simple example is equivalent in one respect to a net in which no moderator permission is required. Figure 4 shows such a fragment; here, moderator permission is required for addition of new participants, but once in a meeting, a participant may leave or suspend (and resume) itself without other permissions.

Of course, the net as originally written might still be preferable as a meeting protocol. Even though the moderator in the original example never denies permission to leave, a designer may well

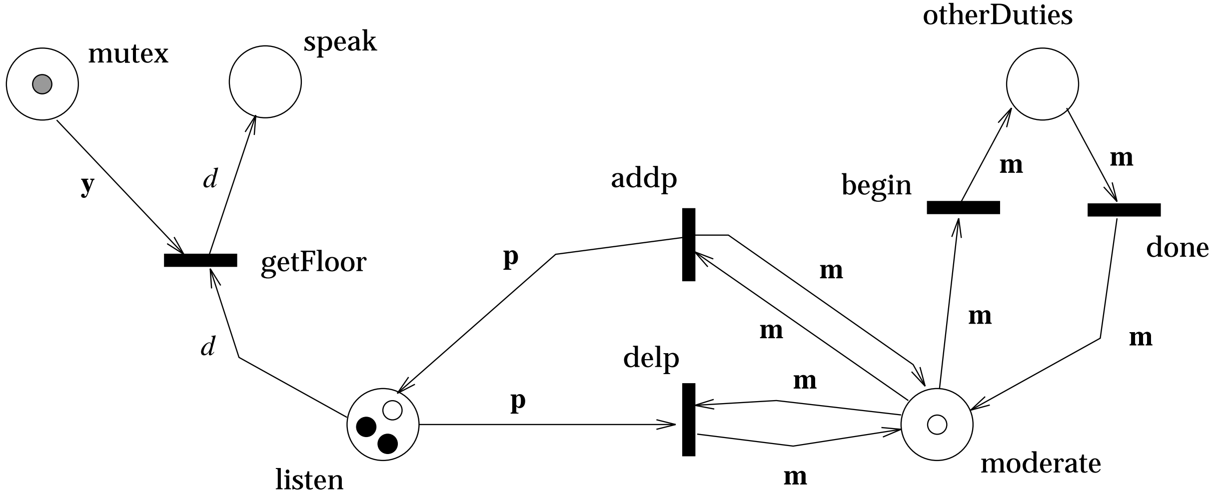

Figure 5: Moderator with other duties, requires permission.

wish the moderator to be involved in such operations if only as a matter of recordkeeping.

Figure 5 shows a further variation on the "permissions" theme. In this CPS fragment, additional duties have been added that may take the moderator away from the main meeting floor for a time. We have indicated this with a place labeled "otherDuties". Firing the "begin" operation will remove the **m** colored token from the "moderate" place, disabling the "addp", "delp", etc. operations until the moderator executes "done" to return from the other duties. It makes good sense to specify that alteration of the makeup of a meeting must be done when the moderator is not busy with other things. However, it does not make sense to specify that the meeting must come to a halt until the other duties are completed. Note that in our CPS, normal getting and releasing of the floor by participants may still go on while the moderator is otherwise occupied.

We repeat that these behaviors can easily be added into the CPS examples that use color variables too.

3.5 Transition predicates and other high-level nets

As mentioned in section 2.1, there are several functionally equivalent syntaxes for high-level nets. The alternate form provide, in essence, more compactness of expression but do not add modeling power to a CPS. In predicate-transition nets [GL81], for example, tokens carry information (which we have generically called color) and every transition carrys a *predicate* describing how the input tokens may combine to produce output tokens. The use of predicates can allow one transition in a high-level net to represent behavior that would require several transitions in our simple notation. We will not go into more detail in this report in describing the equivalences. Our current Trellis model does not support transition predicates, but it could easily be extended to do so; either way, the analysis methods we describe below for the current CPS formalism are certainly applicable to alternate high-level net syntaxes as well.

4 Analysis Techniques

The need to analyze a CPS should be apparent to the reader that has spent some time considering the possible behavior of even the simple protocol given in figure 1. As motivation for this section, let us consider for a moment what can happen when the simple meeting CPS is executed.

There are two pairs of operations that are intended to be used in alternation by individual speakers: “getFloor” followed by “releaseFloor”, and “grabFloor” followed by “dropFloor” (by the moderator only). If a normal participant executes “getFloor”, the “dropFloor” operation cannot be executed thereafter since the arc leading from place “speak” to that transition requires an **m** colored token.

If the moderator executes the “getFloor” operation, as a normal participant would, it might appear that the moderator could then execute the “dropFloor” operation, in violation of the informal expectation. In fact, the net structure prevents this by requiring a **p** colored token to be in place “hold” for firing transition “dropFloor”. In other words, “dropFloor” can only be executed if the “grabFloor” operations has first placed a participant on hold. It would appear, then, after a quick informal analysis that the net maintains our intentions.

However, more careful reasoning about the protocol uncovers this interesting behavior. If a moderator first executes “grabFloor” and puts a speaker on hold, there is no requirement in the net that the next operation be “dropFloor”. Once an **m** colored token is in place “speak”, the “releaseFloor” operation can be executed, no matter how the **m** token got there. In essence, if a moderator grabs the floor, it can then behave as if it obtained the floor through the normal channel. If such a moderator follows “grabFloor” with “releaseFloor”, a second **m** colored token will be deposited into place “listen”.

If a participant does “getFloor” to begin speaking, this scenario can then be repeated. The moderator can again execute “grabFloor” followed by “releaseFloor”, putting a second participant on hold and putting a third **m** colored token in place “listen”. This behavior can continue until all participants are put on hold, and “listen” contains a number of moderator tokens equal to one greater than the number of participants on hold.

This behavior can also be undone. While participants are on hold, the moderator can execute “getFloor” with one of the **m** colored tokens in place “listen”, and then (against the alternation assumption) follow that with “dropFloor”, releasing one of the held **p** colored tokens and eliminating one of the extra **m** tokens. The participant, now speaking again, can execute “releaseFloor” to rejoin the “listen” pool. The moderator can repeat this cycle, releasing in turn all held participants.

Several points should be made about this situation. First, even simple protocols can exhibit complex behavior. Secondly, complex or not, the behavior of a CPS easily can be unexpected. We did not intend for the example protocol to have the behavior described; the “covert” operations were discovered well after its design as other aspects of the CPS structure were being discussed. This surprise, though small, illustrates our point about analysis quite well. In this case, the CPS behavior is not particularly harmful; however, its operation does not match the specifications we had in mind, and its extra behavior does not map well onto the natural and expected actions of a meeting.

Thirdly, informal reasoning cannot be counted on to reliably uncover all the possible behaviors of a CPS. We draw an analogy to program testing vs. program verification; testing (sampling) is necessary, but not sufficient for full confidence. In our example, we first concluded that a moderator could not execute “getFloor” followed by “releaseFloor”, arguing that a **p** token was needed in place

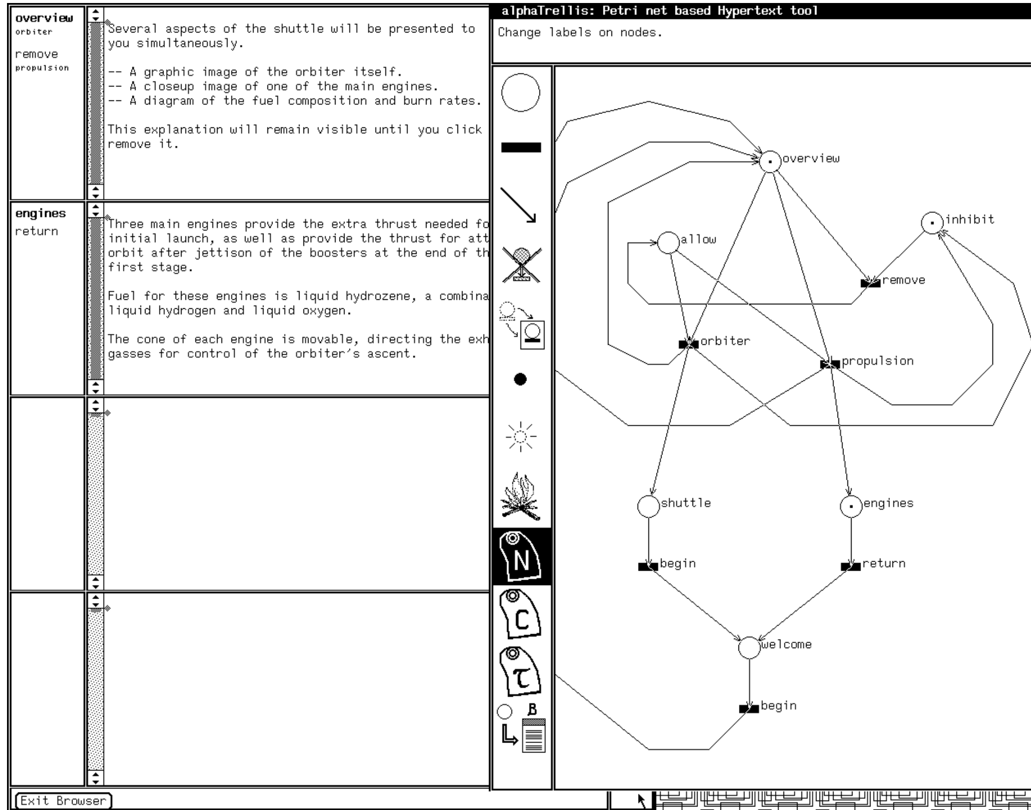

Figure 6: Trellis CPS with programmed browsing behavior.

"hold". We then went on to contradict this conclusion, discovering another vector by which that precondition could in fact be obtained. With such informal reasoning, one cannot be sure all important behavior has been deduced. When is it safe to stop reasoning?

In the next section we present a formal analysis method we have developed for a version of Trellis that is based on a non-colored PT net. Following that, we discuss how this analysis is being extended to the colored nets in a CPS.

4.1 Model checking for Trellis hyperdocuments

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 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 [SF89].

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 6 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?

  ```
  |= 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 we built to represent a CSP parallel program [SF90a] contains about 50 places and transitions, and generates a state machine of 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.

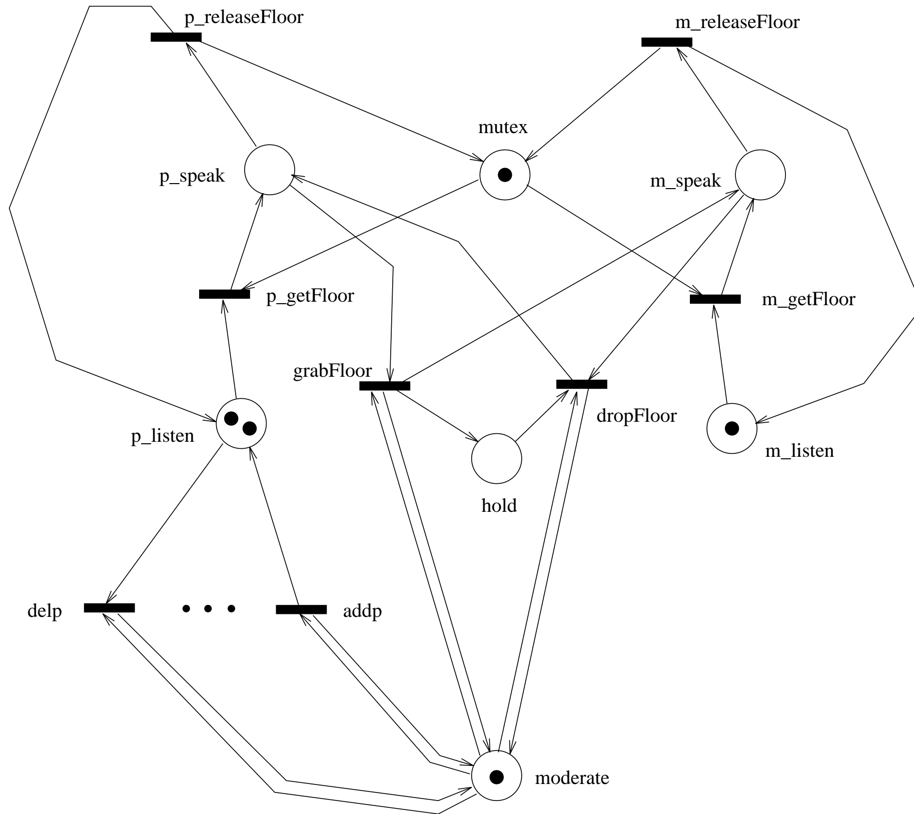

Figure 7: Expansion of colored net to Petri net for analysis.

4.2 Extension to CPS analysis

We are currently building tools to adapt the basic model checking form of net analysis to the colored PT nets used in Trellis CPS models. The basic approach depends on the well-known result from PT net theory that high-level nets provide more compact, more expressive, modeling notations but do not extend the basic power of classical PT nets. In essence, a high-level net can represent a net fragment that would require several structurally-similar net fragments in classical notation. Correspondingly, techniques are known for "unfolding" a high-level net into an equivalent non-colored PT net.

Figure 7 shows such an unfolding of a portion of the colored net used in our simple meeting protocol. We analyze a CPS by generating a state space automaton for the equivalent unfolded net and applying the model checker as just described. Our current research efforts are concentrated on a tool for helping a developer to interpret the unfolded net and CTL queries in terms of the original colored net.

We have included timing on transitions as part of the Trellis model, but in this paper we have not dealt with that aspect in modeling or analysis. We should note that if the untimed subset of Trellis is used (that is, if all transitions are $(0, \infty)$), then the complete analysis we have described here is possible. Analysis in the presence of timing is a subject for other papers.

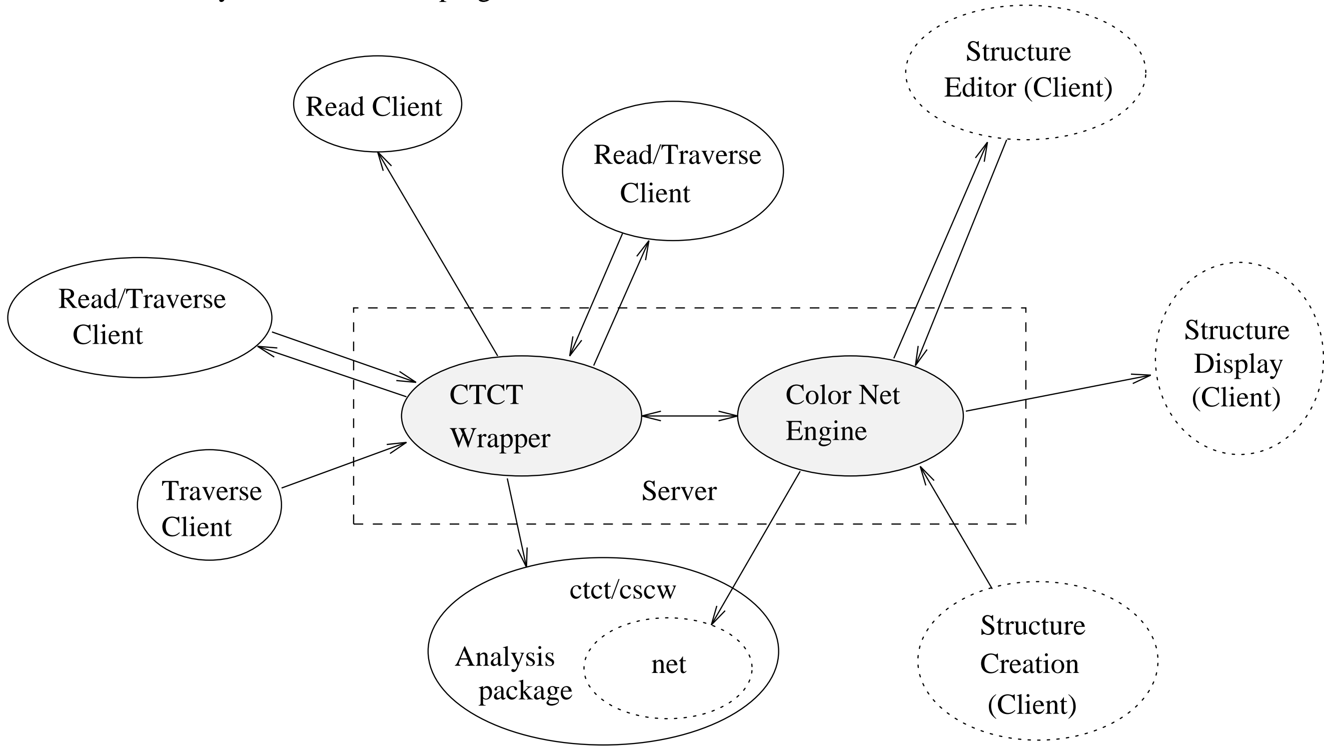

Figure 8: System architecture of a Trellis implementation.

5 Trellis: Prototyping and enacting a CPS

In this section we explain the basic architecture of a Trellis-based implementation and show how it can be used for prototyping and enactment of a CPS. This should illustrate more clearly the earlier observation that a hyperprogram integrates task with information.

Recall that a CPS is a colored timed PT net (CTN) that is annotated with fragments of information (text, graphics, video, audio, executable code, other hyperprograms). The CTN encodes the basic individual actions and group interations of a CSCW application, but appropriate visual interfaces are needed to provide users with a tangible interpretation of the net and its annotations. For example, annotations on net places might be Unix file names, with display names attached to transitions. When a token enters a place during net execution, the file for that place would be presented for viewing. The names of enabled transitions leading out of the place would be shown as a menu of selectable *buttons* next to the file. Selecting a button (with a mouse, usually) would cause the net to fire the associated transition, moving the tokens around and changing which content elements would then be active.

In a Trellis implementation, this cooperative separation between net and interpretation is realized by a distributed *client/server* network, as shown in figure 8. Every Trellis model is an information server—an engine that accepts remote procedure call (RPC) requests for its services.

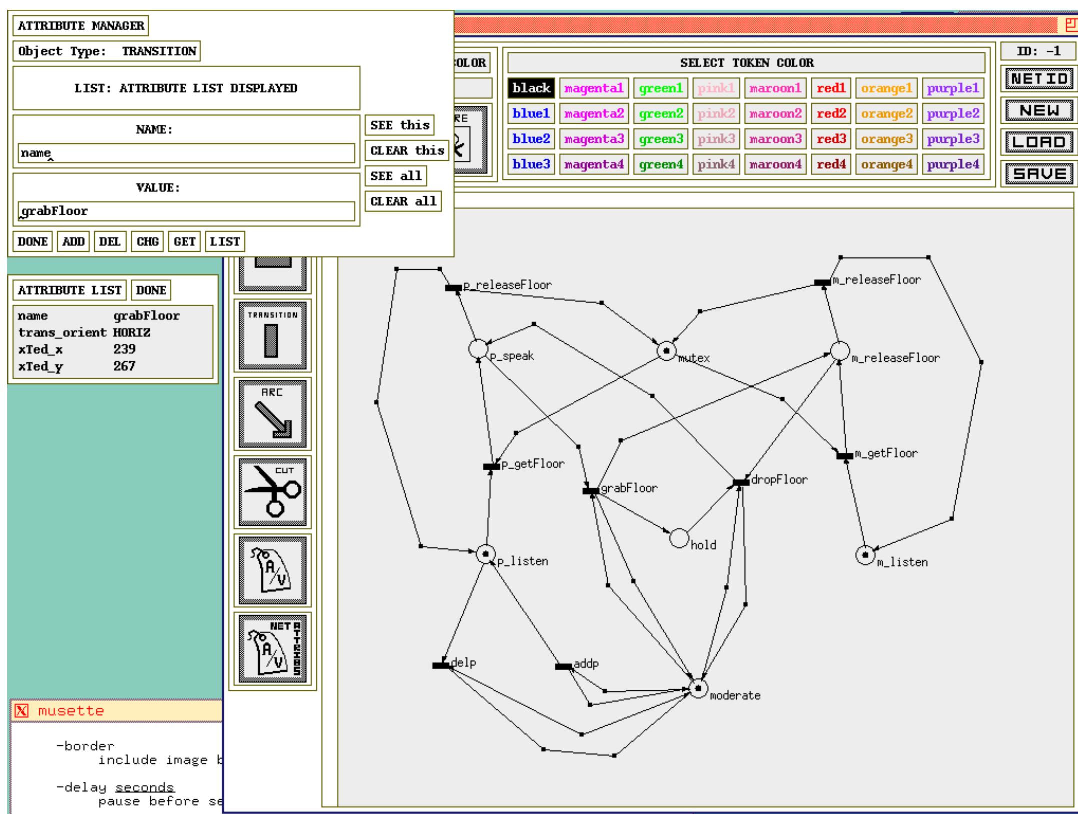

Figure 9: xTed CPS editor client for Trellis.

The engine has no visible user interface, but does have an API that allows other remote processes to invoke its functions for building, editing, annotating and executing a PT net. Interface clients are separate processes that have visible user interfaces and communicate with one or more engines via RPC. Clients collectively provide the necessary views, interactions, and analyses of a net for some specific application domain. Simply put, Trellis clients are the syntax of an application, whereas Trellis engines are the dynamic semantics; clients and servers provide an application's look and feel respectively.

Figure 6 shows two clients for the original Trellis system, αTrellis (based on non-colored PT nets). Here, a graphical editor client (on the right) allows construction and execution of a net, and provides a visual representation of the structure of the automaton. A text browsing client is show on the left; it renders the annotations on marked net places as visible text when the net is active. Each client is executing as a separate process, and both are communicating via RPC with the engine process. Figure 9 shows the colored net editor xTed that has been written to interact with the new Trellis engine for CPS construction.

In the early stages of a collaborative tool design, a Trellis document is built that encodes the desired interactions, as in the meeting protocol discussed in section 3. Text and graphics (or video, if needed) are created to explain each portion of the CPS; as part of the authoring process for the hyperdocument, the net components are annotated with the names of the files containing these explanatory content elements. Testing and analysis can then be done on the CPS using the CTN

structure; usage trials can be done through browsing with the collective Trellis client interface. At this stage, a CPS prototype is an "active hyperdocument" with interactions simulated, but with actual information exchange and data manipulation simply explained with text and graphics stubs.

Once a CPS design is stable, a more substantial implementation can proceed by replacing the text and graphics that specify behavior of each part of the CPS with executable components that realize the actual behavior and provide the designated services. In the curent Trellis engine a Lisp interpreter is provided to assist in this stage of prototpying. The result is a program with a hypertextual interface that controls group use of some information base. An example of a Trellis hyperprogram using the Lisp interpreter is given in [SF92].

6 CPS Application: Software process modeling

In addition to the meeting structure we have shown, and in addition to CSCW tools in general, many different forms of software can be based on CPS models. We have investigated several and are currently constructing others. The common thread in all these applications is multiple users that need to be aware of their mutual existence, and an associative link structure relating together the components of the applications (agents, activities, information, etc.).

Our domains of Trellis investigation include: hypertext and hypermedia, in which multiple readers can share and interact with the linked information elements [SF91]; image browsing indexes, where images are classified according to common characteristics, and the manner of filtering and sharing images among collaborating astronomers is encoded in the net structure defining the index; parallel program browsers, where the coordination of a CPS is applied to program processes rather than to people [SF90a, SF92].

One application we will discuss is some detail here is modeling the process of developing software systems, from requirements to product, from initial design and specification through maintenance. This project, called σTrellis, is a specific version of the Trellis engine, and a specific collection of interface clients tailored to the capture, analysis, and management of information crucial to the software development process, and tailored to representation of the behaviors of the agents involved in the process. In section 6.1, we outline the basic requirements and resulting components of a software process model. In section 6.2 we demonstrate how a Trellis CPS satisfies these requirements. In section 6.3 we conclude with an outline of the main structure, capabilities, and uses of the σTrellis system and methodology.

6.1 Basic capabilities and requirements

Our process models are being designed by integrating ideas drawn from the materials on process from the SEI [Hum88]; the MVP project of Rombach [LR90]; the material contained in the proceedings of the past software process workshops; and from personal communications with representatives of the 14 industrial affiliates of the Purdue/Florida/NSF Software Engineering Research Center.

Process models represent document, design, and code descriptions, relationships among these components, constraints on their temporal and logical creation points, data flow paths among them, and pre/post conditions describing their effects on the overall information content of a development effort. Any process modeling mechanism must assist these areas of software development (from Rombach [LR90]):

- understanding and communication
- measurement
- analysis of captured data
- planning
- execution (actual development)
- process improvement

To produce models meeting these goals, a formalism must satisfy these requirements:

- express relationships among product, process, quality, and resource components;
- support instrumentation mechanisms (data capture);
- support analysis (data reduction);
- support composition of existing models;
- support tailoring to specific projects or sites;
- support execution of models as guidance to the process.

6.2 Trellis CPS meets these requirements

The software process is a mixing of humans, information objects, programs as data, and programs as computations. This mixing with humans is the key to a good modeling formalism. Our detailed discussion of the Trellis technology was meant to emphasize the fact that, while Trellis allows information structures with programmable behavior, it supports and even emphasizes user control of execution ("browsing") over these structures. Since Trellis is basically a man/machine interaction model, it already provides many of the facilities identified above as requirements for a software process modeling formalism.

Human interaction is where Trellis differs strongly with other efforts at representing and enacting process. Most other efforts have concentrated on programming-language-like features and implementations for a model; the σTrellis methodology will emphasize the man-machine interaction and will make "notation" of secondary importance. σTrellis conceives a process model as being mostly human-driven, and therefore presents a largely hypertextual interface rather than a programming-language-like interface.

After several discussions with Rombach's MVP group, we now believe that two major missing aspects of other process modeling efforts are support for human-directed model enactment, and the personal annotation facilities of a hypertext system. We believe that Trellis is uniquely suited for rectifying these problems. Trellis has been well-explored as a hypertext vehicle; it also has inherent concurrency semantics, collaborative multi-user formalism, and a client-server architecture–all of which make it process-oriented, unlike other hypertext systems.

The process of software development is inherently a coordinated activity of (often) many people (engineers, customers, managers). Any system that purports to coordinate the activities of multiple agents, be they computers or humans, must provide several basic services, as does the new Trellis model:

- *activities:* the basic units of collaborative work;
- *agents:* the effectors of activity;
- *information:* the data to be acted upon during collaborative work;

- *relations*: for expressing how agents and information components depend on, or affect, each other;
- *exchange mechanism:* for sharing information, data or parameters among agents;
- *threads:* the collections of possible agent behaviors;
- *synchronization mechanism:* for coordination of parallel activity threads.

A good model should provide these fundamentals with as little other structure as possible, in order to maximize the analytical power one can bring to bear on the model. With extra baggage, designed with good intentions to provide extra "expressiveness" or more "flexibility," one often ends up with a model that become analytically intractable.

Our previous work with Trellis shows it to be an effective and analyzable model for hypermedia documentation. The early parts of this report show how Trellis can encode the behavior of group interactions. As mentioned earlier, this unique blending of both *task* and *information* in one formal structure makes Trellis CPS hyperprograms singularly appropriate for both specification and enactment of CSCW applications.

6.3 Past, present, and future information

In this section we outline the main components of σTrellis, in which the past, present, and future of a development effort is represented in a unified framework: *past* means that historical information about the progress of a system effort is captured and made available for browsing; *present* means that the model always reflects the current state of development; *future* means that an important goal of modeling is to improve the processes that are represented as σTrellis models.

σTrellis consists of:

- a new Trellis engine, adapted from the current colored engine and tailored with capabilities specific to the needs of a software development process;
- a model editing client for X windows, derived directly from the current Trellis editor xTed;
- a model browsing client for X windows, allowing exploration of the linked information database of a model; this client will be hypertextual, allowing the display of documents associated with the process, annotations provided by participants in a process, video/audio data gathered from meetings, data gathered during development, etc;
- a model enactment client for X windows, executing the colored timed PT net in a CPS under human direction, in Trellis fashion; enactment may be *simulated*, for training, or it may be *actual*, for control of a real development effort;
- several model analysis clients, allowing measurement, examination, and reasoning about both the static (links) and dynamic (net execution) aspects of a model and its process; one of these clients is a *model checker* that adapts the method discussed in the technical rationale to the new colored timed PT nets;
- a *process capture* client, allowing a model to be built indirectly by measurement and observation of an actual development (as opposed to direct model construction with an editor); we think of this as a *descriptive* technique, since the resulting model will indicate what *was done* rather than what should be done.

The model browsing and enactment clients offer different views of models from different perspectives (user, manager, engineer; functional, structural, temporal dependencies; past, present, future form of the process; etc.).

The editing client is augmented with other forms of prescriptive model construction; for example, translators will be written as appropriate to convert information in machine readable source notations directly into model format. Thus we do not expect that all parts of a realistic model will be hand constructed with a graphical editor.

Process improvement

The last item above, the process capture client, is especially interesting. We think it is important in our early work to create a *prescriptive* capability in σTrellis (which the model editor gives), so that ideal or experimental processes can be unambiguously defined and studied. However, it is the process capture client that will given real improvement leverage to an organization that develops large software. The capture client gives σTrellis a *descriptive* capability—that is, the ability to attach to a development project, unobtrusively gather data, measurements, observations about what activities happen, when they happen, where, by whom, etc., and then to construct a model from this data of the actual process. The capture client obviously will not be of the same nature as the other, highly visual clients; rather, it will monitor development in the background, adding structure to an σTrellis model as development progresses by making RPC invocations Trellis engine services.

An important characteristic of an effective software process is *predictability* [Hum88]. This attribute requires the ability to measure the crucial aspects of development, and to apply statistical (repeatable) controls to the procedures involved with the measured quantities. Trellis models serve as a basis for implementing various measurements and evaluations of system development practices.

Improvement will then come from comparison of ideal or defined models with the captured models. Ideal models will give property measures that are expected or desired; captured models will be subjected to the same measures, and where differences are found from the ideals, engineers can be put to work on specific refinements.

Other sources of improvement we expect from using σTrellis will be in training of new engineers by "replay" of past developments as simulated process enactments; from unambiguous and accessible definition and documentation of a process for those participating in it; and from the ability to incorporate more directly information from past development into new efforts (reuse).

Product models, too

A final point to make is that σTrellis provides a vehicle for broader modeling than we have emphasized here. Though we are concentrating initially on the development process, the basic Trellis engine we are building to support σTrellis can be applied in other aspects of software systems as well; we will call these other aspects *product models*. For example, we mentioned that Trellis can be used to express and browse the control structure of program source code [SF90a, SF92]. As a related example, module designs can be expressed as Trellis models, too; σTrellis clients can then be written to apply, say, Zage's design quality metrics to the design models. As another example, a high-level system structure derived from requirements can be expressed as a Trellis model; an interface client can then be written to apply, say, the COCOMO estimation method to the model.

No special techniques are required to integrate product models with process models in σTrellis . Since the same Trellis engine is used for representing each, the hierarchy in Trellis will directly allow product models to be components of a development process model, and *vice versa*. The interface clients will operate on both types of model for construction, browsing, and analysis.

7 Comparison with related research

In general, the previously cited papers defining the various forms of high-level PT nets all mention the appropriateness of the model for representing interactions among users and computations. Our project goes beyond such recognition by providing a modeling framework that includes unique analysis methods, as well as a system design for prototyping and simulation of collaborative tools.

Other research projects have looked at various aspects of the domain we are studying. Fischer is using IO automata [Fis91] to model human/computer interactions in CSCW settings. The Suite project [DC92] is system for construction of CSCW tools; its prototyping facilities are more sophisticated than those of Trellis, but no emphasis is given in Suite on formal methods or analysis of the underlying protocol.

A commercial package, *Design/CPN*, is available from MetaSoftware providing extensive editing capabilities for building hierarchical colored Petri nets. Temporal logic has been used to describe structural aspects of hypertext [BK90], but the goal in that work is to define subgraphs of a structure rather than the dynamics of browsing, as in Trellis. The only project we know of other than Trellis that uses temporal logic for PT net analysis is by Sinachopoulos [Sin89]; the emphasis in this work is on timing in a timed net model. No other project we know of uses model checking for PT net state space analysis.

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 supporting σTrellis 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 timed 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.

Intellectual leadership in the field of software process modeling comes from SEI, with its process assessment procedures, and with the writings of Watts Humphrey [Hum88].

In terms of experimental projects, Kellner has described a study of how StateMate can be used to present several different views of a development process [Kel89]. This work describes an

experiment at SEI, and most closely parallels the approach of σTrellis; we feel the basic idea of unified static and dynamic properties in one model deserves an industrial trial.

The MVP-L project [Rom91] has looked as an Ada-like syntax for expressing salient process properties in a form that can be machine translated into plans, code, documents, etc. Marvel [KFP88] has looked at using rule-based systems for assistance in the software development process.

There are numerous other process projects documented in the proceedings of the annual software process workshop. We have mentioned only a few to give an idea of their nature. These efforts have taught us useful views of processes, but they have not been comprehensive in their support for the human/computer and human/human interaction that is central to a collaborative effort. They have been mostly language oriented, or have looked at applying some particular technological area (like expert systems) to process, while retaining a flavor of traditional computing research.

σTrellis differs by offering hypertextual interaction with a model (i.e., associative linking and retrieval of its components) and by having a direct formal representation of collaborative interaction among agents in a process. The Trellis CPS model directly integrates the dynamics of process with the information entities and relationships of software development.

σTrellis also differs from existing projects in that the Trellis CPS implementation framework allows RPC interaction with the model. Any new interface a customer needs can be constructed fairly easily and will communicate with existing models. Thus σTrellis is an open system.

References

[BK90] C. Beeri and Y. Kornatzky. A logical query language for hypertext systems. In A. Rizk, N. Streitz, and J. André, editors, *Hypertext: Concepts, Systems, and Applications*, pages 67–80. Cambridge University Press, November 1990. Proceedings of the European Conference on Hypertext.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8:244–263, 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.

[DC92] P. Dewan and R. Choudhary. A high-level and flexible framework for implementing multi-user user interfaces. *ACM Transactions on Information Systems*, 10(4):345–380, October 1992.

[Fis91] M. Fischer. Decision making based on practical knowledge. In *Proc. of the 1991 Coordination Theory and Collaboration Technology Workshop*, pages 89–97. National Science Foundation, June 1991.

[GL81] H. J. Genrich and K. Lautenbach. System modeling with high-level Petri nets. *Theoretical Computer Science*, 13:109–136, 1981.

[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.

[Hum88] W. S. Humphrey. Characterizing the software proess: A maturity framework. *IEEE Software*, 5(2):73–79, March 1988.

[Jen81] Kurt Jensen. Coloured Petri nets and the invariant-method. *Theoretical Computer Science*, 14:317–336, 1981.

[Kel89] M. I. Kellner. Software process modeling: Value and improvement. *Technical Review 1989*, pages 23–54, 1989.

[KFP88] G. Kaiser, P. H. Feiler, and S. S. Popovich. Intelligent assistance for software development and maintenance. *IEEE Software*, May 1988.

[LR90] C. M. Lott and H. D. Rombach. A mvp-l1 solution for the software-process modeling problem. In *Collected Solutions from the 6th International Software Process Workshop* (Hakodate, Japan), October 1990.

[Mer74] Philip M. Merlin. *A Study of the Recoverability of Computing Systems*. Ph.D. dissertation, University of California at Irvine, Department of Information and Computer Science, Irvine, CA, 1974. Also available as Technical Report 58, Department of Information and Computer Science, University of California at Irvine (1974).

[MF76] Philip M. Merlin and David J. Farber. Recoverability of communication protocols–implications of a theoretical study. *IEEE Transactions on Communications*, COM-24(9):1036–1043, 1976.

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

[Rei83] W. Reisig. Petri nets with individual tokens. *Informatik-Fachberichte*, 66(21):229–249, 1983.

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

[Rom91] H. D. Rombach. Mvp-l: A language for process modeling in-the-large. Technical Report CS-TR-2709, Department of Computer Science, University of Maryland, College Park, MD, June 1991.

[SF89] 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.

[SF90a] P. David Stotts and Richard Furuta. Browsing parallel process networks. *Journal of Parallel and Distributed Computing*, 9:224–235, 1990.

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

[SF91] P. David Stotts and Richard Furuta. Dynamic adaptation of hypertext structure. In *Third ACM Conference on Hypertext Proceedings*, pages 219–231. ACM, New York, December 1991.

[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.

[Sin89] A. Sinachopoulos. Logics for Petri-nets: Partial order logics, branching time logics and how to distinguish between them. *Petri Net Newsletter*, pages 9–14, 8 1989.

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