

Interpreted Collaboration Protocols and their use in Groupware Prototyping

Richard Furuta

Computer Science Department
Texas A&M University
College Station, TX 77843-3112
Tel: (409) 845-3839
Email: furuta@bush.cs.tamu.edu

P. David Stotts

Computer Science Department
University of North Carolina
Chapel Hill, NC 27599-3175
Tel: (919) 962-1833
Email: stotts@cs.unc.edu

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. In Trellis, group interaction protocols are represented separately from the interface processes use them for coordination. Since these protocols are interpreted (rather than being compiled into applications), group interactions can be changed as a collaborative task progresses. Changes can be made either by a person editing the protocol specification “on the fly” or by a silent “observation” process that participates in an application solely to perform behavioral adaptations.

Trellis uniquely mixes hypermedia browsing with collaboration support. We term this combination a *hyperprogram*, and we say that a hyperprogram integrates the description of a collaborative task with the information required for that task. As illustration, we describing a protocol for a moderated meeting and show a Trellis prototype conference tool controlled by this protocol.

This work is partially supported by the National Science Foundation under grant numbers IRI-9007746, IRI-9015439, and IRI-9496187, by the Software Engineering Research Center (University of Florida and Purdue University), and by the Texas Advanced Research Program under Grant No. 999903-155.

(copyright notice here)

THE MOTIVATING PROBLEM

We present a technology, collectively called Trellis, for modeling, and analysis of coordination structures, and for prototyping CSCW system based on such coordination structures. Our method separates the collaboration protocol guiding the behavior of a CSCW system out from the code for the system implementation. The collaboration protocol is interpreted and serves as a central information engine in a CSCW system.

Some CSCW systems already provide “parameterized” access to the underlying protocol. For example, the Suite system [2] allows users to set and change the granularity of text exchange, from character level up to paragraph level. Our approach takes this ability to customize a protocol a step further, allowing dynamic alteration of any aspect of a protocol, even aspects that system designers do not (or cannot) anticipate. This capability is useful for very long-lived interactions, ones in which the nature of the shared tasks might evolve based on the outcomes of earlier actions or decisions.

It is difficult to anticipate in detail all the behaviors of CSCW systems, especially when parallel activities are involved. Our method of separating out the protocol and expressing it in a form different from the code gives support for examining formally and informally the interaction of parallel tasks and concurrent users. The graphical notation of a network is no more difficult to understand than direct code, and some researchers in visual languages have found that graphical notations enhance human ability to understand parallel computation threads. We exploit this informal enhancement in the Trellis methods.

We also have developed formal analysis methods that will find logical flaws in protocols while they are being developed. These methods are analogous to program proof techniques in formal software engineering, and they com-

plement code testing and debugging in our CSCW system prototyping process.

HYPERPROGRAMS AND CSCW

This report shows how Trellis hyperprograms are used to prototype groupware applications based on explicitly expressed, and dynamically alterable, collaboration protocols.

The Trellis project [12, 13] 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*. In a CSCW context, this task is a description of the coordinated interactions among multiple users; the information is some interpretation given to this coordination framework for a specific application domain.

The Trellis model serves several functions with unified notational framework: its structures shared applications; its synchronizes loosely coupled parallel execution threads; 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 well-understood interpretation as hypertext, Trellis hyperprograms are especially useful for processes in which human direction is an important aspect of the control flow, such as software process models. These forms of computation are referred to as being *enacted*, rather than executed, to distinguish the major role they have for human input and human decisions (and for CSCW, human interactions).

THE COLOR TRELLIS MODEL

Complete formal definitions of the color Trellis model can be found in a previous paper [16]. In this report, we are concentrating on the practical applications of Trellis for collaboration protocols, so we summarize the model informally.

The Trellis project is an ongoing effort to create interactive systems that have a formal basis and that support analysis techniques. The first such effort was a hypertext model [12], followed by a framework for highly-interactive time-based programming (termed *temporal hyperprogramming* [13]). 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 a mechanism by which agents may exchange data. This new model basically follows the Trellis framework of annotating a form of *place/transition net* (*Petri 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 Petri nets, their structure, and common behaviors. Readers familiar with this material may skip to the section on colors.

Net theory basics

The notation used here is taken from Reisig [11]. For the interested formalist, Murata [9] 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 Petri 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 Petri 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.

A transition t in a Petri net is said to be 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 re-

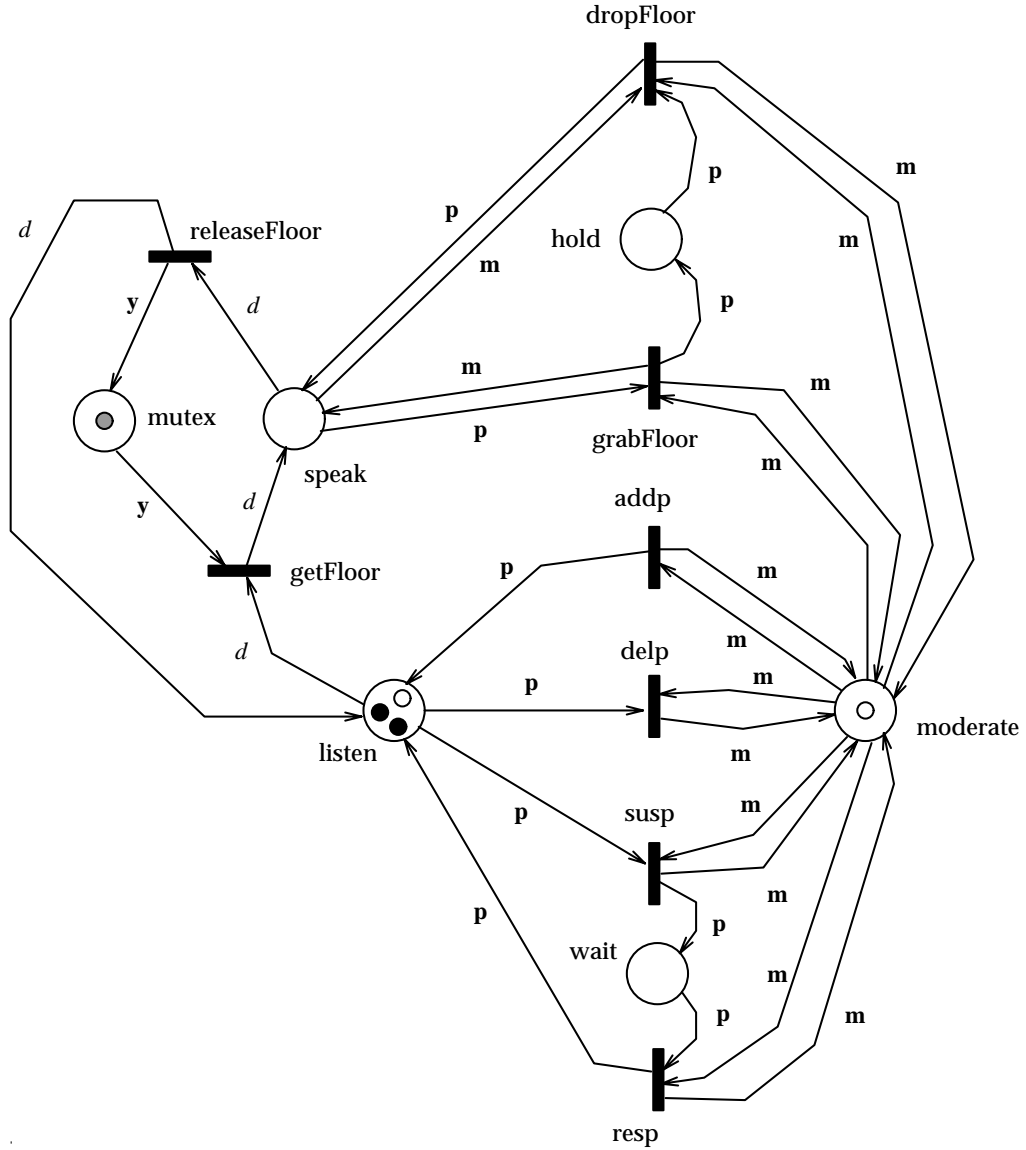


Figure 1: CTN for a simple meeting protocol.

side 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.

Color time nets

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 sev-

eral forms by different researchers, including predicate-transition nets [4], colored Petri nets [6], and nets with individual tokens [10].

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 [9]. 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. The net notation used in Trellis we refer to as *colored timed nets*, or *CTN*.

In CTNs, tokens have type (color) and may carry data. A CTN marking is a snapshot (at some point during execution) of how many tokens of each color reside in each net place. 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.

Color expressions appear on the arcs of a CTN, defining which combinations of token classes are required for an event (transition) to occur (fire). In Figure 1 these appear in boldfaced and italics, consisting of variable and constant names and algebraic operations. Execution of the net requires pattern matching of the expressions on the input arcs of a transition to the colors of the tokens in the input places. A 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 through the extended examples to follow.

A note on event timing

The timing of the original Trellis model has been retained and combined with color to obtain the CTN notation. Each transition in a CTN has two time values, representing roughly a delay and a timeout. For a transition t , its release time t_r represents the number of time units that must pass once t is enabled before it can be fired; its maximum latency t_m represents the number of time units that may pass after t is enabled before it fires automatically. Each time value may range from 0 to ∞ , with the constraint that $t_r \leq t_m$. The degenerate case, a $(0, \infty)$ transition, has a zero delay and an infinite timeout, which is the behavior of transitions in an untimed net. Our Trellis implementation of CTNs defaults each transition to a $(0, \infty)$ timing, which can be changed by the designer of a net.

This temporal structure is very similar to that of Merlin’s *Time Petri nets* [7, 8], 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. In a previous report we showed how transition timings are used to adapt net execution to user browsing behavior [14].

Though we use timed events in Trellis hyperprograms, space prevents us from discussing them further in our

examples here. Details on use of timed events have been reported elsewhere [13, 15].

Annotations on a CTN

A Trellis hyperprogram is completed by giving annotations for the components of the CTN. Considering our earlier claim that Trellis integrates task with information, one can consider the CTN as the task description, and the different annotations as the information required by the task.

One important category of annotation is *content*. Fragments of information (text, graphics, video, audio, executable code, other hyperprograms) are associated with the places in a CTN. Another category of annotation is *events*, which are mapped to the transitions of the CTN. A third category of annotation is the *attribute/value (A/V) pair*; a list of A/V pairs is kept with each place, each transition, each arc, and for the CTN as a whole.

Annotations can be used by the Trellis implementation to prototype group interaction tools. In our earlier experiments, we showed how annotations can naturally be used to produce hypermedia documents that coordinate multiple readers. Such collaborative hyperdocuments are the basis for our prototyping approach, discussed in detail in a later section.

COLLABORATION PROTOCOLS

In the next few sections, through an extended example, we illustrate the basic execution and interpretation of a CTN as a collaboration protocol. Following this, we discuss analysis of collaboration protocols and verification of the behavior encoded in one. After analysis, we discuss the current Trellis implementation and show how it is used to prototype groupware based on CTN collaboration protocols.

Example: Simple moderated meeting

Figure 1 shows a CTN 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 modera-

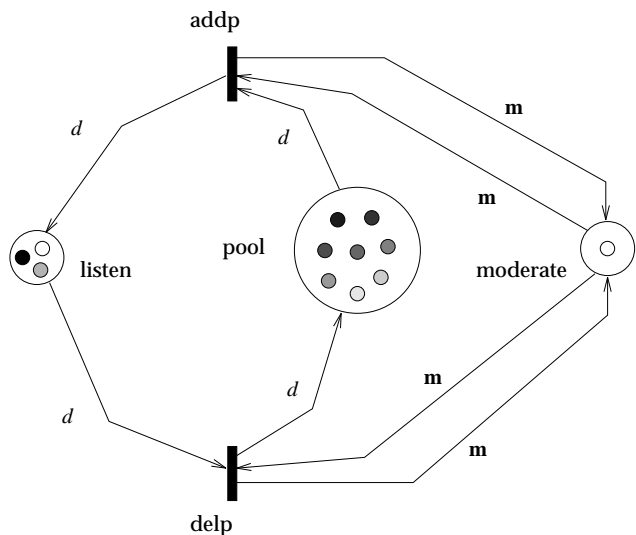


Figure 2: Detail for participant allocation.

tor 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 CTN 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 “addPerson”. There is one input arc to this transition, labeled **m** coming from place “moderate”. The label **m** in boldface indicates a color *constant* which we have selected for the moderator token. The “addPerson” transition has two output arcs: one labeled **p** to place “listen”, and another labeled **m** back to place “moderate”. As before, **p** is a color constant representing the participant class.

When a token of color **m** is present in place “moderate”, 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 CTN 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.

Variant: Distinguishing participants

Let’s consider other CTN 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”

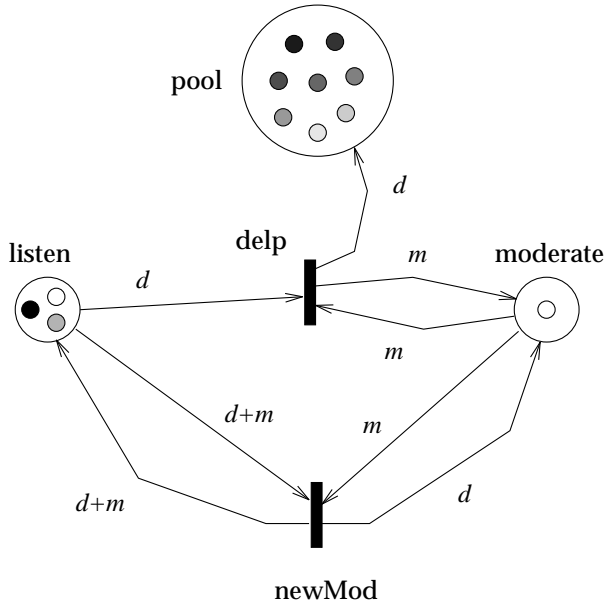


Figure 3: Detail for changing the moderator.

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 CTN 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 CTN 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 allow 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.

Variant: Changing the moderator

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 CTN details for moderator swapping. In this fragment, we have altered the labels on arcs between the “moderator” place and the previously existing operations (like “addPerson” and “delete”) 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 “addPerson” specifies that execution of such an operation must maintain whatever color m represents (*i.e.*, the moderator cannot change simply by executing “addPerson” and the other previously discussed meeting control functions).

We have added another operation, “newModerator”, to specifically perform moderator swapping. The arc leading into “newModerator” 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 “newModerator” 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 “newModerator” 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 place “listen”. Finally, as written, the CTN allow a moderator to swap only with someone who is listening—a speaker cannot become the new moderator without first releasing the floor.

PROTOCOL ANALYSIS

The need to analyze a CTN 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 analysis section, let us consider for a moment what can happen when the CTN for the simple meeting 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 CTN 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 CTN structure were being discussed. This surprise, though small, illustrates our point

about analysis quite well. In this case, the CTN 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 CTN. 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** color token was needed in place “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?

Formal analysis methods

We have developed analysis techniques for CTNs that allow a designer to determine if the behavior of a net matches formal specifications of desired behavior [17]. Our current analysis software is a modification of a *model checking* system developed by Clarke for verifying the behavior of concurrent programs. Since the Trellis model is based on a concurrent automaton (Petri net), hyperprograms are expressions of concurrent processes and are quite amenable to the model checking methods of verification. The model (the Petri net) is annotated with atomic predicates that indicate true conditions at each node (such as “participant is speaking” or “floor can be grabbed”). A temporal logic is used to specify properties the protocol should exhibit during execution (such as “if a participant is placed on hold, he should be given back the floor before any other participant gets the floor”), and the the model checker examines the state space of the automaton (coupled with the atomic predicates at each state) to see if the specification is upheld.

Our current system works by expanding CTNs into normal Petri nets (termed “unfolding” in the net literature), then computing the state space of unfolded net as the model for the checker. Interpretation is difficult in this approach; the temporal logic properties make statements about the unfolded model, but the results must be interpreted back in the original Petri net protocol. To ease this problem, we are working on model checking algorithms that work directly on the CTN and do not require an unfolding step.

PROTOTYPING GROUPWARE

In this section we explain the basic architecture of a

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

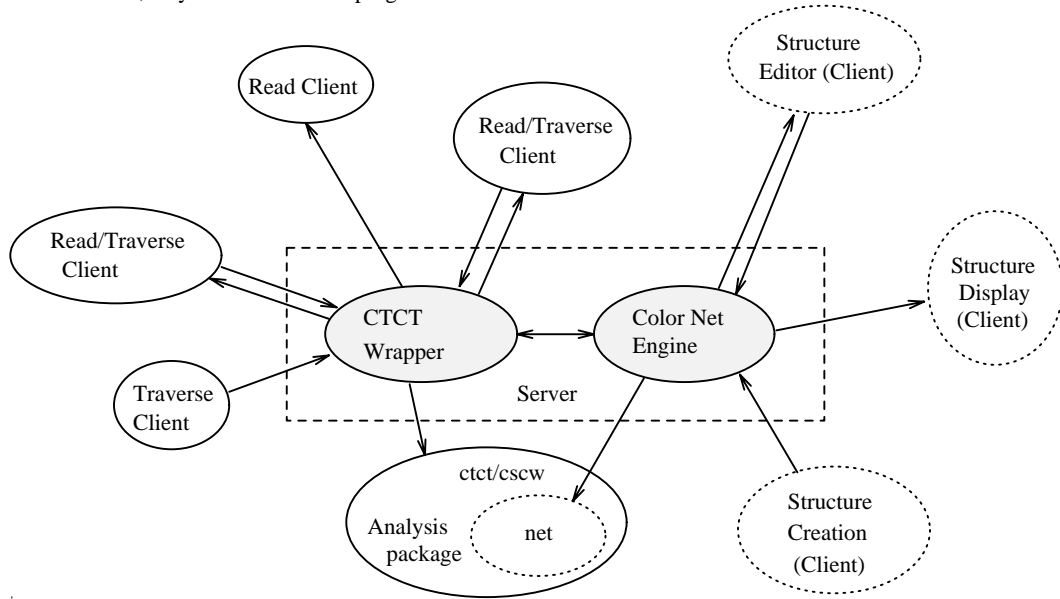


Figure 4: System architecture of a Trellis implementation.

Trellis-based implementation and show how it can be used for prototyping and enactment of a CTN. This should illustrate more clearly the earlier observation that a hyperprogram integrates task with information.

Recall that a CTN is a color time Petri 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 interactions 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, in our Trellis-based hypertext system, annotations on net places are Unix file names, with “button” names attached to transitions. When a token enters a place during net execution, the file for that place is presented for viewing. The names of enabled transitions leading out of the place are shown as a menu of selectable words next to the file contents. 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 and visible.

In a Trellis implementation, this cooperative separation between net and interpretation is realized by a distributed *client/server* network, as shown in figure 4. Every Trellis model is an instance of an information server—an engine that accepts remote procedure call (RPC) requests for its services. The engine has no visible user interface, but does have an API that allows other re-

mote processes to invoke its functions for building, editing, annotating and executing a CTN. 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.

In the early stages of a collaborative tool design, a Trellis hyperdocument is built that encodes the desired interactions, as in the moderated meeting protocol discussed earlier. Text and graphics (or video, if needed) are created to explain each activity represented in the CTN; as part of the authoring process for this hyperdocument, the net components are annotated with the names of the files containing these explanatory content elements. Testing and analysis of the protocol can then be done with model checking; usage trials can be done through hypertextual browsing via the Trellis client interfaces. At this stage, the prototype groupware tool is an “active hyperdocument;” user interactions are accurately simulated, but the necessary information exchange and data manipulation are simply explained with text and graphics stubs.

Figure 5 shows this stage in prototyping a conferencing tool based on the meeting protocol from Figure 1.

xTrellis

DONE

HELP

POLL

PLACE

TRANSITION

TRANSITION

ARC

CUT

Max tokens/place = 12

red1 = <5>

TOKEN

FIRE

SELECT TOKEN COLOR

black	magenta1	green1	pink1	maroon1	red1	orange1	purple1
blue1	magenta2	green2	pink2	maroon2	red2	orange2	purple2
blue2	magenta3	green3	pink3	maroon3	red3	orange3	purple3
blue3	magenta4	green4	pink4	maroon4	red4	orange4	purple4

NET ID

NEW

listen

getFloor

```

===== L I S T E N =====
You are either a participant or the
moderator, and ...
You are in a listening state.
You should be able to get the floor
if nobody is speaking by selecting
the corresponding button at the left.
Other transitions that might appear
are not relevant in this state.
=====
P

```

moderate

addp

delp

susp

swapMod

```

===== M O D E R A T E =====
You are the moderator and you are
now in a moderating state.
You should be able to grab the floor
even if someone else is speaking.
You can also add, delete or suspend
participants.

```

listen

getFloor

```

===== L I S T E N =====
You are either a participant or the
moderator, and ...
You are in a listening state.
You should be able to get the floor
if nobody is speaking by selecting
the corresponding button at the left.
Other transitions that might appear
are not relevant in this state.
=====

```

SELECTO SELECTO Console

Figure 5: xTed CTN editor client for Trellis.

The figure shows one instance of the Trellis CTN editor client, and two instances of the Trellis text browser client. All three client processes are communicating with a single Trellis engine, each giving a different view of the information stored in that engine. One of the text browsers has been executed using the “moderator” color (meaning the browser only displays content elements for places that are marked with tokens of its color); it consists of the two windows in the upper right corner. The other browser has been executed using the “participant” color, and is the single text window in the lower left. Notice that the moderator window has extra buttons in its button menu, indicating operations the moderator can perform that a normal participant cannot. Notice also that the moderator browser shows windows for both the “moderating” activity and the “listening” activity, since

the CTN shows moderator-colored tokens in both places (the moderator can also be a participant in a conference under this protocol).

Selective display of events

One use of A/V pairs is in hiding events from being visible in some windows, as desired by the protocol designer. Consider again the hyperdocument shown in Figure 5. A decision must be made about where to display the buttons representing the enabled CTN transitions. Notice in the net that the transition “delete” (delete participant) is enabled for firing, and that it has as input places both the “moderator” place and the “listen” place. The color expressions specify that if a participant-colored token (blue in this case) is in place “listen” and a moderator-colored

token (red in this case) is in place “moderator” then the delete operation can be performed to take a participant out of the conference. Both a listening participant and a presiding moderator are required as preconditions to the delete operation.

But who is allowed to invoke this operation? By default in the Trellis browser shown, an enabled transition is displayed as a button in the window for each of its input places. In this case, under the default behavior, either the moderator or the participant could then initiate the delete operation by selecting the appropriate button in their respective menus. Notice, however, that in our example the “delete” operation only appears as a button in the moderator window.

This selective display is accomplished with A/V pairs. To override the default, the constant “all” is stored in the engine as the value of attribute “HideButton” on the arc connecting the “listen” place to the “delete” transition. The browser interprets this A/V pair as telling it not to display the button in the “listen” window no matter what color it is using. Thus, neither participants nor the moderator will see the “delete” button in the “listen” window, and they will not be able to invoke the delete operation from it.

Notice that the moderator does see the “delete” button in the “moderator” window. There is no “HideButton” attribute defined on the arc from that place to the “delete” transition, and the default display behavior applies.

A different hiding behavior could have been achieved by storing the A/V pair (“HideButton,blue”) on the arc from place “listen” to transition “delete.” In this case, rather than hiding the “delete” button from all listeners, only browsers executing on behalf of participants (color “blue”) would have the button hidden. When the moderator browser displayed the “listen” place contents, the “delete” button would be shown; the moderator would then have the option to invoke the delete operation from either its “moderator” window or from its “listen” window. For our example, the protocol designer thought it less confusing for special moderator functions to be accessible only in the “moderator” place window.

Evolution to working code

Once the design of a collaboration protocol is stable, a more substantial implementation can proceed by altering the content annotations on the CTN. The passive text and graphics stubs are replaced with executable content components—code written to realize the desired actual behavior and to provide the services required by the groupware tool. Evolution of the content annotation from stubs to code is assisted again by the third class of

annotations: attribute/value pairs.

For migrating static stubs to active code, A/V pairs can be used to exchange data. An A/V pair is simply stored in the engine as two character strings, attached to some net component. Thus, a conference tool can be created from the hyperdocument shown by taking a copy of the browser code and altering it to present “chat” windows instead of simple text file displays. The “chat” behavior is implemented by having the browser instance capture the keyboard input from its user and store that as a character string in the engine (attached to the appropriate net component, such as an active place) under some agreed-upon attribute name like “TextToYou.” The browser would also query the engine to obtain the value of attribute “TextToYou” attached to some other net component (representing some other participant). After retrieving this value, the browser displays the text in its interface as input from this other participant.

Dynamic protocols

More sophisticated behavior can be added to the emerging groupware tool as prototyping proceeds, until the desired level of functionality results. This final prototype has its multi-user interactions governed by the Trellis engine implementing the collaboration protocol. Since the CTN defining this protocol is interpreted by the engine (as opposed to being compiled into the application) the CTN definition can be altered, even as a collaborative work effort is underway.

One way to dynamically alter a protocol is to use the Trellis CTN editor. This client can be accessing an engine no matter what other clients are using that engine. Thus, a human can edit the net specification as the protocol is being executed, making changes and testing them as a collaboration progresses. Such alterations are probably most useful in the debugging stage of protocol development, but might also be important during use of a collaborative system based on a Trellis protocol, especially if the effort has a duration of days, weeks, or months. Changing the way a meeting is run would not require termination of the meeting and recompilation of the groupware tool. In effect, Trellis offers the protocol of an interaction as persistent data rather than a “burned-in” CSCW system aspect.

Significant changes can be made as well, not just minor variations. Some changes can be anticipated and programmed into the protocol, as illustrated by our example that allows a participant and the moderator to exchange roles. In this way, Trellis offers the kind of protocol “parameterization” offered by systems such as Suite [2]. In Suite, several ways in which users might want to change a protocol are anticipated, and the ability

to change these aspects is built into the user interface. For example, users can decide to couple their text exchanges tightly (see every character others type as they are typed) or loosely (only see new text when others enter complete lines or paragraphs). This coupling can be changed during collaboration.

However, not everything can be anticipated by a protocol designer, especially for complex activities. In Trellis, the entire collaboration protocol is available for alteration, not just certain designer-chosen aspects like message coupling. Editing a Trellis CTN can allow new forms of interaction to be created, tested, and then “linked” in with a few new arcs in the net. Access control can be designed and added with net structure, even as a collaboration progresses. Sections of protocol behavior can be “detached” as needed and reattached as needed, all by an “observer” using the net editor. Annotation can be altered dynamically, allowing new information to be substituted for old.

Trellis is a client server system, so rather than altering a protocol with explicit human editing, a “silent” client can listen to a collaboration, gathering data about the interactions other clients are having with the engine and making decisions about what alterations to make to the protocol. Such a client makes changes by appropriate RPC calls to the engine (add place, add arc, add transition, delete, annotate, etc.) during the collaboration.

RELATED RESEARCH

In general, the previously cited papers defining the various forms of high-level Petri 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 [3] to model human/computer interactions in CSCW settings. Among the many group conferencing tools and other CSCW systems that have been built, the Suite project [2] is most in accord with our goals of dynamic protocol changes and easy prototyping of collaborative interactions. Suite is a system for construction of CSCW tools; its prototyping facilities are more sophisticated than those of Trellis. In Suite, the group interactions of a collaboration can change dynamically, but only by adjusting values of predefined parameters. The main protocol and its parameters are compiled into an application; this protocol is specified in a programming language (C) rather than being specified in the more ab-

stract fashion of Trellis. 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 color Petri nets.

The use of Petri nets as a specification medium for man-machine interaction appears previously in the literature. For example, van Biljon [18] has described a special grammar-based notation for designing man-machine dialogues as languages, which are then realized with a hierarchy of Petri nets as recognition automata. Another example is the work of Holt [5], 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 Trellis is related to other hypertext engines that have been used in experimental software support systems. The HAM (hypertext abstract machine) [1] 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—color time Petri nets. This gives the model an elegant structure that can be both programmed and analyzed.

References

- [1] CAMPBELL, B., AND GOODMAN, J. M. HAM: A general purpose hypertext abstract machine. *Commun. ACM* 31, 7 (July 1988), 856–861.
- [2] DEWAN, P., AND CHOUDHARY, R. A high-level and flexible framework for implementing multi-user user interfaces. *ACM Transactions on Information Systems* 10, 4 (Oct. 1992), 345–380.
- [3] FISCHER, M. Decision making based on practical knowledge. In *Proc. of the 1991 Coordination Theory and Collaboration Technology Workshop* (June 1991), National Science Foundation, pp. 89–97.
- [4] GENRICH, H. J., AND LAUTENBACH, K. System modeling with high-level Petri nets. *Theoretical Computer Science* 13 (1981), 109–136.
- [5] HOLT, A. W. Diplans: A new language for the study and implementation of coordination. *ACM*

- Transactions on Office Information Systems* 6, 2 (Jan. 1988), 109–125.
- [6] JENSEN, K. Coloured Petri nets and the invariant-method. *Theoretical Computer Science* 14 (1981), 317–336.
- [7] MERLIN, P. M. *A Study of the Recoverability of Computing Systems*. PhD thesis, 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).
- [8] MERLIN, P. M., AND FARBER, D. J. Recoverability of communication protocols—implications of a theoretical study. *IEEE Transactions on Communications COM-24*, 9 (1976), 1036–1043.
- [9] MURATA, T. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE* 77, 4 (Apr. 1989), 541–580.
- [10] REISIG, W. Petri nets with individual tokens. *Informatik-Fachberichte* 66, 21 (1983), 229–249.
- [11] REISIG, W. *Petri Nets: An Introduction*. Springer-Verlag, 1985.
- [12] STOTTS, P. D., AND FURUTA, R. Petri-net-based hypertext: Document structure with browsing semantics. *ACM Transactions on Information Systems* 7, 1 (Jan. 1989), 3–29.
- [13] STOTTS, P. D., AND FURUTA, R. Temporal hyperprogramming. *Journal of Visual Languages and Computing* 1, 3 (1990), 237–253.
- [14] STOTTS, P. D., AND FURUTA, R. Dynamic adaptation of hypertext structure. In *Proceedings of Hypertext 91* (Dec. 1991), ACM, pp. 219–231.
- [15] STOTTS, P. D., AND FURUTA, R. Hypertextual concurrent control of a Lisp kernel. *Journal of Visual Languages and Computing* 3, 2 (June 1992), 221–236.
- [16] STOTTS, P. D., AND FURUTA, R. Modeling and prototyping collaborative software processes. In *Information and Collaboration Models of Integration* (1994), S. Y. Nof, Ed., Kluwer Academic Publishers, pp. 365–390. Also published 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.
- [17] STOTTS, P. D., FURUTA, R., AND RUIZ, J. C. Hyperdocuments as automata: Trace-based browsing property verification. In *Proceedings of the 1992 European Conference on Hypertext (ECHT92: November 30–December 4, Milan, Italy)* (1992), ACM Press, New York, pp. 272–281.
- [18] VAN BILJON, W. R. Extending Petri nets for specifying man-machine dialogues. *International Journal of Man-Machine Studies* 28 (1988), 437–455.