Model Checking CobWeb Protocols for Verification of HTML Frames Behavior

David Stotts
Dept. of Computer Science
Univ. of North Carolina
Chapel Hill, North Carolina
stotts@cs.unc.edu

Jaime Navon
Dept. of Computer Science
Catholic University of Chile
Santiago, Chile
jnavon@ing.puc.cl

ABSTRACT

HTML documents composed of frames can be difficult to write correctly. We demonstrate a technique that can be used by authors manually creating HTML documents (or by document editors) to verify that complex frame construction exhibits the intended behavior when browsed. The method is based on model checking (an automated program verification technique), and on temporal logic specifications of expected frames behavior. We show how to model the HTML frames source as a CobWeb protocol, related to the Trellis model of hypermedia documents. We show how to convert the CobWeb protocol to input for a model checker, and discuss several ways for authors to create the necessary behavior specifications. Our solution allows Web documents to be built containing a large number of frames and content pages interacting in complex ways. We expect such Web structures to be more useful in "literary" hypermedia than for Web "sites" used as interfaces to organizational information or databases.

Categories and Subject Descriptors

I.7.2 [Document and Text Processing]: Document Preparation – hypertext/hypermedia, markup languages

General Terms: Verification, Documentation, Languages

Keywords: model checking, frames, HTML, browsing semantics, formal semantics, verification, temporal logic, literary hypertext

1. INTRODUCTION

CobWeb is a formal model of the interactions among users in a collaborative enterprise that owes it heritage to the Trellis hypermedia project [1, 2, 3] and related followon Web research [4, 5]. Based on this formalism, we built a collaborative Web browser, also called CobWeb [6]; we demonstrated its operation on CobWeb protocols that were specified and verified using the techniques described in [7] (and summarized in this paper). In the CobWeb browser, interaction rules governing how groups of users can collaboratively browse Web pages are expressed as CobWeb protocols stored externally to the code of the browser itself. This means that users can choose different interaction schemes dynamically (moderated meeting, shoulder surfing, etc.) and load them into CobWeb, altering the behavior they then see in subsequent browsing.

WWW 2002, May 7—11, 2002, Honolulu, Hawaii, USA.
ACM 1-58113-449-5/02/0005.

A CobWeb protocol is fairly general; we show in this paper how it can be used to solve a problem in single-user Web browsing. The multi-user components in CobWeb are used here to represent multiple HTML *frames,* and the interactions among users are used to represent the interactions of Web pages among the frames. The analysis methods we have developed for verifying correct behavior for collaborative interaction rules are used to show that the frames defined in HTML documents exhibit the desired behavior when browsed.

Frames, first introduced as a Netscape specific feature, have been a component of HTML since version 4.0. Though their utility is controversial, most people agree that good frame sets can be difficult to author. There are several ways to ensure that a document has a properly structured set of frames and correct interaction behavior. One is to be an excellent author, know all aspects of HTML thoroughly, and apply the features flawlessly. This approach works fine for very small frame sets, but for large sets is impractical to depend on. Another is to use structured document editors like FrontPage and let them structure your framesets according to some selection of predefined layouts and behaviors. This approach lacks flexibility and limits an author to fitting his ideas into someone else's designs. It works well for Web sites and other documents where novelty of structure is not necessarily a goal. Web sites need to be clear and not confusing to their users; for this reason many do not use frames in the first place.

We offer a third alternative, which is to specify the behaviors desired from one's frames and then check the authored HTML for compliance with the specs, using a verification tool.

This approach rescues the manual frames authoring method from being limited to small, simple frame structures. It will be most useful for people who wish to create Web pages as "documents" rather than "sites". With tool support for ease of structuring, the literary community would find frames particularly interesting for building complex Web-based hypermedia for exactly the reasons frames are difficult to write. Complex use of frames would allow creation of Web structures that are intricate, have surprising window behavior, create new windows as well as placing new content in the frames of existing windows... in short that provide the reader with interesting and surprising browsing experiences. Such documents are already being created, but usually with a "literary" system like Storyspace [8] instead of the Web.

In the following sections we give our techniques for supporting complex frame development. For completeness of the presentation, we first give some background on P/T nets, temporal logic, and model checking, which make up the CobWeb formal analysis methods we have developed. The reader who does not wish to dig into these technical details can still get a clear sense of the value of this solution by reading the closing sections showing how HTML frames are analyzed. We conclude with a discussion of other ways to use our analysis methods.

2. CPNs AND CobWeb PROTOCOLS

The type of hypermedia structure we are concerned with was previously studied in the Trellis project [1]. Our hypothesis then was that hypermedia documents could (and should) explicitly allow and manage concurrent browsing activities. The benefits included literary complexity and multi-target browsing activities as is now realized in the Web as frames. To manage the concurrency problems we developed an analysis method based on model checking for the original Trellis model that would identify hypermedia structures that had correct or incorrect (desired or undesired) behaviors [2]. Our recent research has extended this earlier analysis work to a more complicated hypermedia model, one that explicitly represents multiple users and collaborative activities [7]. We call the new model CobWeb (Collaborative Web), and we show here how CobWeb protocols and our model checking allow us to check HTML frames for correct (author-intended) structure and behavior.

Colored P/T nets

For brevity we will not present a full formal explanation of colored P/T (place /transition) nets. Instead we give an example diagram and refer the reader to the Trellis papers for a full discussion of how net models can represent hypermedia documents with multi—head, multi-tail links [1, 4, 5]. A full description of colored P/T nets can be found in several different places, including [9].

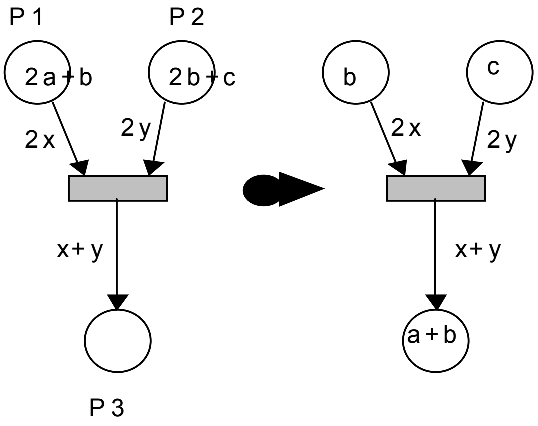


Figure 1: Dynamic behavior of CPN

Figure 1 shows a simple *colored P/T net* (CPN). It is a bipartite graph; the circles are *places* and the bar is a *transition*, and as implied by the connections shown, places connect to transitions and vice versa. Places contain *tokens* and tokens have *color* (type). Transitions can have several input places; this models the synchronization of the actions represented by the places. Transitions can have several output places as well; this represents the creation of concurrent activities. Computation is modeled as moving tokens around from place to place by *firing* transitions. A transition may fire when each of its input places has the correct configuration of token number and color; firing causes the matching tokens to be removed from the input places and tokens of a certain configuration to be deposited in the transition's output places.

The arcs in a CPN are annotated with color expressions, containing both color variables and color constants. In this example, place P1 contains 2 tokens of color "a" and 1 token of color "b"; place P2 contains 2 "b" tokens and 1 "c" token. Here we see "2x" on the arc from P1 to the transition. This means that 2 tokens of the same color are required to enable the transition. The other input arc has "2y" as its expression; this implies the 2 tokens of the same color are also needed from P2, and since the variable "y" is used their color must be different from the color from P1. The arc out of the transition to place P3 has expression "x+y" which says that one token of the color from P1, and one token of the color from P2 are placed in P3 when the transition fires.

To fire a transition we must find some binding of colors to the variables appearing in the arc expressions going into the transition; if no such mix of colors can be found for the set of tokens in the input places, then the transition is not enabled and may not fire. If such a binding exists, the transition can be fired and we proceed to assign values to the variables leaving the transition according to the same assignment. Each input place will see its number of tokens of each color reduced according to the needed tokens for the arc expression. Consequently, each output place will see its number of tokens incremented according to the expressions in the arcs leaving the transition towards them. In Figure 1, the right side shows the CPN state after firing the transition using the color assignment of X is "a" and Y is "b".

There are several different ways to interpret CPNs to represent the semantics of collaborative processes. A place could be associated to a hypertext page, a software process activity or state of participation in a meeting protocol. We provide here a description based on the CPN formalism that can be used not only for the scenarios of the previous section, but for a wide variety of situations involving collaborative work.

The CobWeb collaboration protocol model

We now give a definition of the CobWeb model of collaborative hypermedia. Informally, CobWeb is a CPN that is annotated with the various components of the system under study (Web pages, users, roles, etc.). More formally, a CobWeb collaborative protocol CCP is a tuple

CCP = (CPN, A, E, T, W, Ma, Me, Mt)

in which:

- CPN = (Σ, p, t, a, n, c, g, e, in) is a colored P/T-Net structure
- A is the set of activities
- E is the set of events
- T is the set of actors
- W is the set of workpieces
- Ma is a mapping from p into A
- Me is a mapping from t into E
- Mt is a mapping from a color set $x \in \Sigma$ into T

The CPN is defined further as follows: CPN = (Σ, p, t, a, c, g, e, in) where

- The Σ is a finite non-empty set of colors
- p is a finite set of places
- t is a finite set of transitions
- a is a set of arcs of the form (s,d) drawn from (p X t) and (t X p)
- c is a color function from p into Σ
- g is a set of expressions composed of color variables and color constants
- e is a mapping of a to g
- in is an initializing function mapping multi-sets of colors in Σ to places in p

Mapping CobWeb to Web frames

As mentioned earlier, CobWeb is general enough to model the interactions of multiple users browsing in a hyperdocument (collaborative browsing). It is also useful, however, for single-user documents like most Web sites; in this simpler context, the colors in the CobWeb model give us the extra leverage to solve problems related to concurrent browsing activities undertaken by this single user. For solving the HTML frames problem, we give some Web-specific interpretations to the components of CobWeb. The colors will be used to represent the frames themselves. Places in the CPN will be mapped to the Web pages that are to appear in frames. Transitions are mapped to the links that appear in these Web pages. In this context, there is only one actor (the single reader) so we do not mention that part of CobWeb. The workpieces are the individual web pages specified by the URIs involved in the frames source. Events are simply link followings, so they are all uniform.

3. TEMPORAL LOGIC AND MODEL CHECKING

There are two main components to the frames-problem solution we are presenting: a set of specifications describing the behavior an author wants an HTML frame set to exhibit when browsed; and a structure that captures the essence of the frame behavior as it was authored. We obtain both of these solution components from the domain of program verification. *Model checking* is a term applied to a group of automated techniques, first investigated by Clarke at CMU [10]. The basic idea is to capture the critical aspects of a system under study in a finite state machine that has been annotated with atomic predicates known to be true at each state. This model is then traversed to determine if the sequences of states that are possible at run-time exhibits certain desired properties; these properties are expressed in *temporal logic*, an extension of predicate calculus that allows assertions to be made about sequences of states (instead of a single state). Research in model checking has focused on finding efficient ways to produce and represent models, and on temporal logic languages for which efficient (non-exponential time) checking algorithms exist.

Temporal logic and dynamic properties of systems

Temporal logic allows expressing the ordering of events in time without introducing time explicitly into the formalism. Although Pnueli [11] and others first used temporal logic for reasoning about concurrent systems, their correctness proofs were constructed by hand and for that reason only small systems could be verified. The introduction of temporal logic model checking algorithms in the 80's allowed the reasoning to be automated [10]. Besides being automatic, model checking has another important advantage over proof—checker based methods: if a formula is not true of a model, we can produce an execution trace that shows why the formula is not satisfied.

The logic we use for specifications is CTL (Computation Tree Logic), the same branching-time temporal logic that was used initially by Clarke's original model checker [10]. Formulae in CTL are built from three components: atomic propositions, Boolean connectives and temporal operators. Each temporal operator has a path quantifier and a temporal modality. The quantifier indicates whether the operator denotes a property that should be true of all execution paths from a given state (A) or whether the property needs to hold only on some path (E). The temporal modality describes the ordering of the events along an execution path. For given formulae Φ and Φ′ their meaning is as follows:

- FΦ is true of a path if there exists a state on the path for which Φ is true
- GΦ is true if Φ is true in every state of the path
- XΦ is true if Φ is true in the next state in the path
- ΦUΦ′ is true if Φ is true along the path until some state in which Φ′ is true

Here are some examples of CTL formulae using this notation:

- AFΦ (on every execution path a state will eventually arrive in which Φ is true)
- AG(Φ => AFΦ′) (on every path, in every state, if Φ is true, then eventually Φ′ will also be true)
- EG(Φ => A(Φ′ U Φ″)) (there is some path in which in every state if Φ is true then eventually Φ″ is true and until then Φ′ remains true)

In informal terms, CTL temporal logic allows one to express properties about how the relationships among variables in a program will change as execution progresses. As a program executes, it changes from state to state; CTL allows assertions about these sequences of changes, such as these:

- "If a program ever gets to a state in which X > 100, then it will never thereafter get to a state where X <= 100" [*CTL*: AG(X>100 ➔ AX (~ EF (X<=100)))]
- "There is at least one possible execution sequence in which the value of Z never goes negative" [*CTL*: EG (Z >= 0)]
- "No matter how the program executes, at some point K will be greater than J and will remain so until M is negative" [*CTL*: AF (K > J ^ A (K>J U M < 0))]

Our CobWeb research has been concerned with Web documents and other hypermedia structures, so the assertions we wish to check are related not to program variables, but rather to entities in Web documents and browsing, and their interrelationships. Examples of CTL assertions in this context appear in the HTML frames example below.

Managing state space explosions in models

In the past, many forms of model checking have been developed and used for different purposes but most of the tools have suffered from the configuration explosion or state explosion problem. Early algorithms computed the state space explicitly and stored the full finite state machine of the model to be checked. This meant that only small to moderate systems could be verified, as construction of the model itself was prohibitive in both time and space, no matter how efficient the checking algorithm itself might be. Symbolic model checking has provided a solution. First presented by Clarke and McMillan [12], the technique is based on binary decision diagrams (BDDs) for representing a state space logically as a large system of Boolean expressions. With BDDs it is not necessary to spend the time and storage space to needed to explicitly compute and store each state in the state space of a model. The use of BDDs to represent the transition relation of a state machine symbolically makes it possible to handle problems with hundreds of state variables and up to 10^50 states or more.

McMillan put these ideas into practice in a tool called SMV [13] that we have used for our CobWeb analyses. SMV extracts a BDD-encoded finite—state model from a system description in a programming—language—like notation; it then uses an exhaustive state-space search algorithm to determine whether the model satisfies specifications in CTL temporal logic. If the model is found to *not* satisfy some CTL formula, SMV will produce an execution trace that proves the formula false.

McMillan's SMV is not the only way to do symbolic model checking. Another well-known system is SPIN from Bell Labs [14]. SPIN uses a linear temporal logic called LTL as its specification notation, whereas SMV uses the branching-time temporal logic CTL. We find the SMV specification notation a bit more natural to use, but for all practical purposes the two systems are equivalent.

4. ENCODING CobWeb STRUCTURES AS SMV MODEL CHECKER INPUT

We are able to apply Clarke's results in model checking to our hypermedia and collaboration problems because of an encoding scheme found by Navon [7]. The CPN in a CobWeb protocol has to be expressed as SMV input in order for a model checker to work on it. Readers who are not concerned with these technical details may wish to skip to the next section where we present an example of modeling HTML frames with a CobWeb protocol.

Figure 2 shows SMV input defining a small finite state machine, one with 2 states. The states are named, and their interconnections are defined in the case statement. One approach to representing the CPN in a CobWeb protocol in SMV is to convert the net into an equivalent finite state machine for direct encoding. It has been known for four decades that converting an *uncolored* P/T net into an equivalent finite state machine is a procedure that requires O(2^n) time and space, where n is the size of the net. Adding color to nets compounds the problem. Converting a colored net to a normal net requires O(2^n) steps where n is the number of colors in the net. Thus we have a doubly exponential problem. Even though the SMV model checker operates in time that is linear in the size of the model + size of the formula, getting the model in the first place is exponentially expensive if we were to try to build a state machine the traditional way and express it in SMV.

```
MODULE main
VAR
   request : boolean;
   state   : {ready, busy};
ASSIGN
   init(state) := ready;
   next(state) := case
      state = ready & request  : busy;
      1                        : {ready, busy};
   esac;
SPEC
   AG(request -> AF state = busy)
```

Figure 2: Simple SMV model

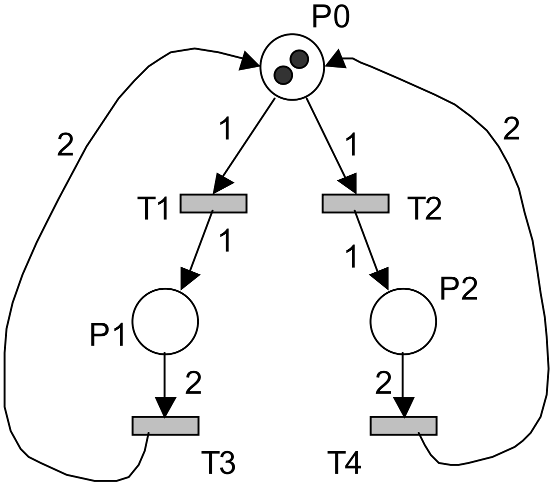


Figure 3: P/T net to be coded in SMV

Navon's encoding technique is a symbolic one, giving advantages similar to those the BDDs gave to model checking over explicit model representation. Since colors complicate the Cobweb structure considerably, for clarity we illustrate the encoding of a simple *uncolored* P/T net here and describe the alterations colors require after that.

In our SMV models of P/T nets we describe each transition as a module. Furthermore, since P/T net behavior is asynchronous, and the selection of the transition to fire is non-deterministic, we use SMV processes to describe this situation. Net places are modeled as an array of state variables. The value of each variable is a non-negative integer that represents the token count of that place in any given state.

For every module representing a particular transition we write SMV code that describe the portion of the transition relation associated to that specific part of the P/T net. This is done by writing a *next* sentence for every input and output place of the transition.

We present now the complete SMV code of the simple P/T net of Figure 3 to illustrate the methodology. Notice that in the code of each module there is a *Define* clause (similar to a macro) that makes the code more readable. The code in SMV includes 5 modules: one for each transition, and one main module in which the processes and state variables corresponding to the 3 places are declared. The initial state of the net (2 tokens in place P0), and the CTL assertion to be checked is also written in the main module. The SMV input then is:

```
MODULE transition1(p)
DEFINE may_fire := p[0] >= 1;
ASSIGN
next(p[0]) :=case
    may_fire  : (p[0] – 1);
    1             : p[0];      esac;
next(p[1]) := case
    may_fire  : (p[1] + 1)mod 10;
    1             : p[1];      esac;
```

```
MODULE transition2(p)
DEFINE may_fire := p[0] >= 1;
ASSIGN
next(p[0]) :=  case
     may_fire : (p[0] – 1);
    1             :  p[0];     esac;
next(p[2]) := case
     may_fire  : (p[2] + 1)mod 10;
    1              : p[2];     esac;
```

```
MODULE transition3(p)
DEFINE may_fire := p[1] >= 2;
ASSIGN
next(p[0]) := case
    may_fire  : (p[0] * 2)mod 10;
    1             : p[0];     esac;
next(p[1]) := case
    may_fire  : (p[1] - 2);
    1             : p[1];     esac;
```

```
MODULE transition4(p)
DEFINE may_fire := p[2] >= 2;
ASSIGN
next(p[0]) := case
    may_fire  : (p[0] * 2)mod 10;
    1             : p[0];     esac;
next(p[2]) := case
    may_fire  : (p[2] - 2);
    1             : p[2];     esac;
```

```
MODULE main
VAR
     t1: process transition1(p);
     t2: process transition2(p);
     t3: process transition3(p);
     t4: process transition4(p);
      p: array 0..2 of 0..10;
ASSIGN
      init(p[0]) := 2;
      init(p[1]) := 0;
      init(p[2]) := 0;
SPEC
     AG(t1.may_fire|t2.may_fire|t3.may_fire|t4.may_fire)
```

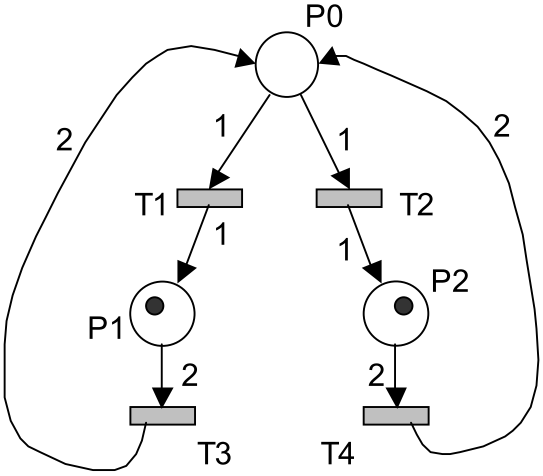


Figure 4: No transition can be fired

The property we want to prove (or disprove) in this case is specified in CTL logic in the SPEC section in the main module. It says that if we start from the given initial conditions it is always true that one of the four transitions can be fired. It turns out that this is *not* true, a fact that is correctly detected by the SMV interpreter in less than a second. The counter example found by the model checker is illustrated in Figure 4. The output of SMV showing this situation follows:

```
procrustes (806):   smv  pnet1.smv
   -- specification AG(t1.may_fire|t2.may_fire... is false
   -- as demonstrated by the following execution sequence
state 1.1:
t1.may_fire = 1
t2.may_fire = 1
t3.may_fire = 0
t4.may_fire = 0
p[0] = 2
p[1] = 0
p[2] = 0

state 1.2:
[executing process t2]
p[0] = 1
p[2] = 1

state 1.3:
[executing process t1]
t1.may_fire = 0
t2.may_fire = 0
t3.may_fire = 0
t4.may_fire = 0
p[0] = 0
p[1] = 1
p[2] = 1

resources used:
   user time: 0.383333 s, system time: 0.133333 s
BDD nodes allocated: 10015
Bytes allocates:1048576
BDD nodes representing transition relation: 285+1
```

Adding colors to P/T nets in SMV

Addition of colors to P/T nets requires a considerable amount more coding in the SMV input language. A state variable with one integer value representing the tokens in a place is not enough since we need to specify the distribution of the tokens into the different colors. A second important difference is that the firing rule is much more involved because of the use of variables in the arcs. The SMV code is, consequently, more extensive and tedious to write but this can be easily automated. In spite of the differences most of the strategy remains the same: we use separate processes for each transition and each of them is described in a separate module. The state variables associated to places are now bi-dimensional arrays (place and color).

We illustrate the methodology in the same way we did with ordinary P/T net by example. Figure 5 shows a CPN with just one transition with one input place and two output places. The transition of Figure 5 will fire with two tokens of the same color in place P0 no matter what color they are. Since the transition may fire in different ways we get a family of *may_fire* conditions and a family of *next* state descriptions, one for each of them.

Without loss of generality assume we have a maximum of just 3 colors: black = 0, red = 1, blue = 2. Then we get the following *may_fire* conditions:

a) may_fire0 := p[0][0] >= 2
b) may_fire1 := p[0][1] >= 2
c) may_fire2 := p[0][2] >= 2

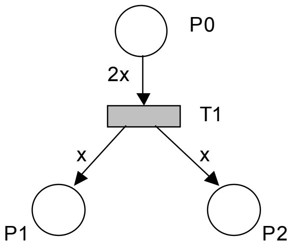


Figure 5: Simple CPN

And depending on which is selected to fire, the next state will be described by one of:

a) p[0][0] := p[0][0] - 2 ; p[1][0] := p[1][0] + 1 ; p[2][0] := p[2][0] + 1 ;
b) p[0][1] := p[0][1] - 2 ; p[1][1] := p[1][1] + 1 ; p[2][1] := p[2][1] + 1 ;
c) p[0][2] := p[0][2] - 2 ; p[1][2] := p[1][2] + 1 ; p[2][2] := p[2][2] + 1 ;

Using the module facilities of SMV we can build one module for each transition in the net in the following way:

- Using the *define* sentence, write as many *may-fire* conditions as necessary depending on the number of ways the transition can be fired. The number of sentences will depend on the number of allowed roles (colors) and also on the arc expressions.
- For each of the possible ways the transition may be fired and using the *case* sentence, write *next* instructions corresponding to the next state of all the input and output places that participate in the transition.

Applying this to the example of Figure 5 produces the following SMV module:

```
MODULE transition (p)
DEFINE may_fire_1 := p[0][0] >= 2;
DEFINE may_fire_2 := p[0][1] >= 2;
DEFINE may_fire_3 := p[0][2] >= 2;
ASSIGN
 next(p[0][0]) := case may_fire_1: p[0][0] - 2 ; esac;
 next(p[0][1]) := case may_fire_2: p[0][1] - 2 ; esac;
 next(p[0][2]) := case may_fire_3: p[0][2] - 2 ; esac;
 next(p[1][0]) := case may_fire_1: p[1][0] + 1 ; esac;
 next(p[1][1]) := case may_fire_2: p[1][1] + 1 ; esac;
 next(p[1][2]) := case may_fire_3: p[1][2] + 1 ; esac;
 next(p[2][0]) := case may_fire_1: p[2][0] + 1 ; esac;
 next(p[2][1]) := case may_fire_2: p[2][1] + 1 ; esac;
 next(p[2][2]) := case may_fire_3: p[2][2] + 1 ; esac;
```

The points to take away from our illustrations are that

- SMV input is very much like a programming language;
- the modules describing CPNs are systematically constructed; therefore

- SMV input can be derived automatically from CPN descriptions

The last point is important, as we will show in the following example that these complex SMV descriptions of CobWeb protocols can be directly derived from HTML frames source text. A user of our methods need never write them or even see them to benefit from the verification tools.

5. AN HTML FRAMES EXAMPLE

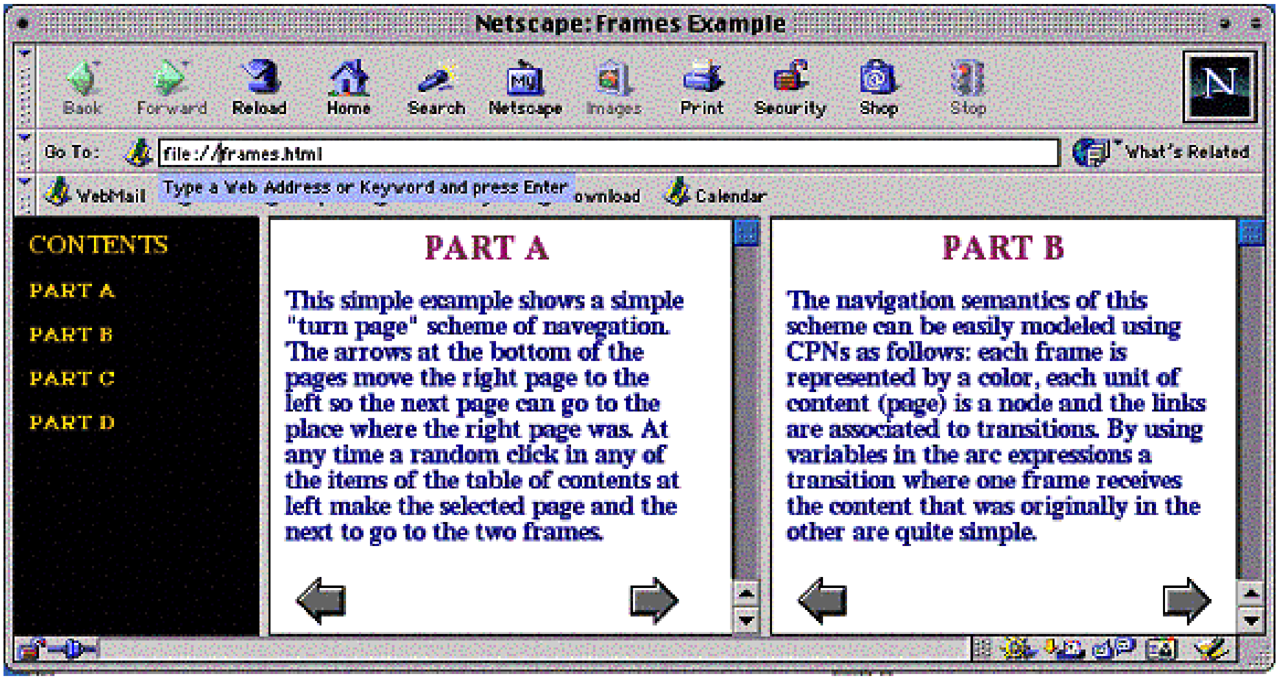
We present now an example to show how CobWeb and model checking solves our problem of verifying frames behavior. Frames were introduced with HTML 4.0 and have been controversial because they add complexity to the navigation semantics, as well as difficulty to Web authoring. The use of frames requires each link to include not only the associated URI, but also the target frame (sometimes implicitly specified) where the contents should be rendered. The problem is that sometimes, after a series of clicks, the user may end up with an unwanted configuration of pages in the different frames if they are not carefully authored.

We can use CobWeb to model frames by mapping various parts of the HTML source text to the components of a CobWeb protocol. We use places in the CPN to represent the workpieces, the content Web pages; transitions will represent the event triggers, the links. We use the colors in the CPN to represent the individual frames involved; for instance a red token in place X means that the contents associated with X are visible in the frame associated with color red. After we build the CPN model we can use our model checker to see if the design behaves as wanted.

Figure 6 shows a nontrivial but still relatively simple Web page with frames. The idea is to use the frames to model a "turn page" kind of navigation through some bibliographic material. There are only three frames: a narrow left frame used to display the table of contents or list of available Web pages (frame A) and two frames of equal size to show the contents of the items (frames B and C).

Navigation is very simple. There are left and right arrows at the bottom of each of the two content frames. A click in any link of the table of contents loads that page into frame A and the next one into frame B. A click in the left arrow of frame A will load what was in frame B into A and the page that follows the one that was in frame B goes to frame B (the one that was in frame A disappears). A click in the right arrow of frame B will load the page that was in A into B and the page previous to that into A (the one that was in B disappears). Both right arrow in left frame and left arrow in right frame are inactive. Finally, we consider the workpieces organized as a circular list of topics so the first page comes after the last one.

The CPN model of this frame structure is shown in Fig. 7. Our example has only three frames so we need three colors. Let them be green for frame A, blue for frame B and red for frame C. To make things simple, let's assume that the material includes only 4 pages (as shown in Fig. 6). The net place at the top of the figure corresponds to the table of contents page whereas the other 4 places are the pages with the actual contents. The state of the net in Fig. 7 corresponds to the same moment in browsing shown in Fig. 6: Part A is on frame blue, Part B is on frame red and Table is on frame green. From T we can fire any of the four transitions at any time because they only require a green token that is always in T. Firing will put blue and red tokens in two consecutive places. Note that the page that is in frame B has only one transition available, the one that needs a blue token. After firing, the blue token goes to the next page and a red token to the next. In the same analogous way the page that sits in frame C can fire only the transition that requires a red token and will put a blue in the place that was red (page moves to the left) and a red into the next one.



CONTENTS

PART A

PART B

PART C

PART D

PART A

This simple example shows a simple "turn page" scheme of navegation. The arrows at the bottom of the pages move the right page to the left so the next page can go to the place where the right page was. At any time a random click in any of the items of the table of contents at left make the selected page and the next to go to the two frames.

PART B

The navigation semantics of this scheme can be easily modeled using CPNs as follows: each frame is represented by a color, each unit of content (page) is a node and the links are associated to transitions. By using variables in the arc expressions a transition where one frame receives the content that was originally in the other are quite simple.

Figure 6: Example frames-based Web page

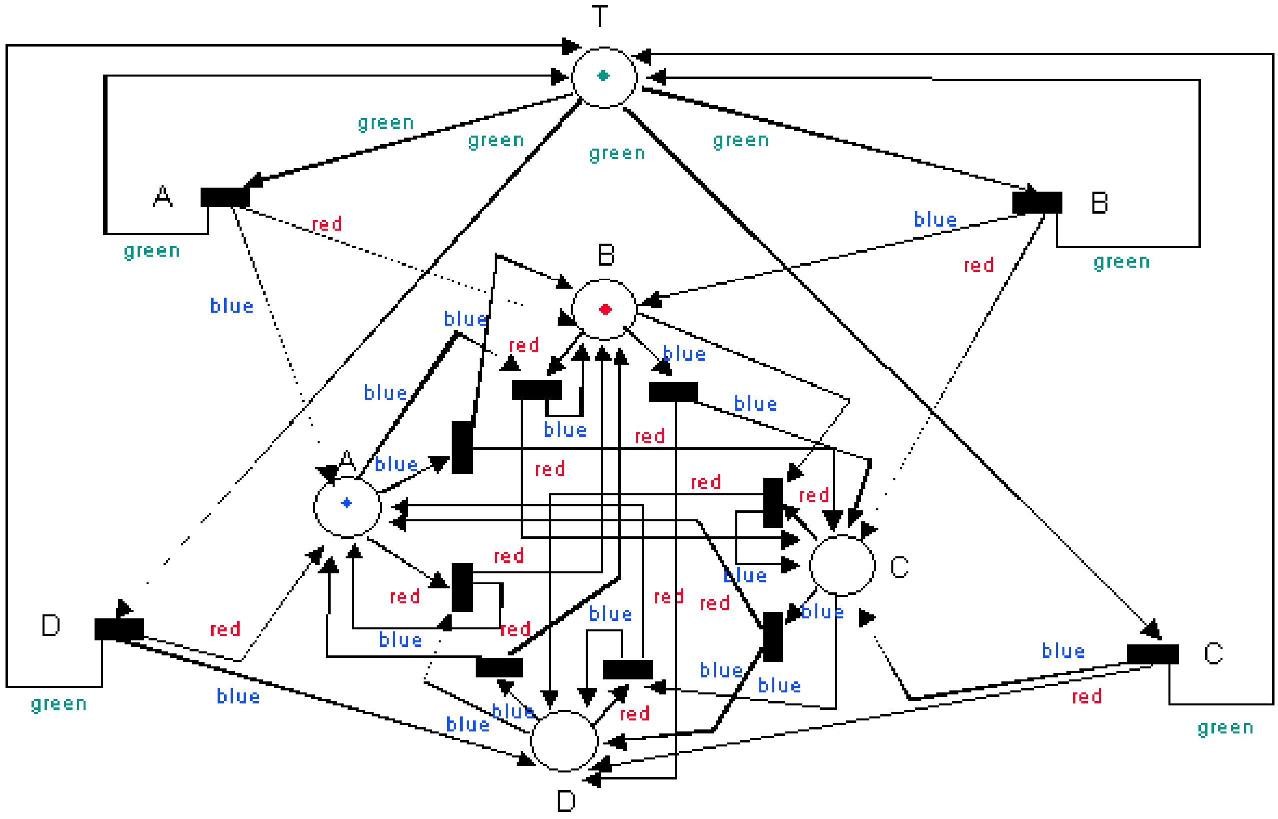


Figure 7: CPN model of frames in Fig. 6

Now we understand informally the navigation model and the behavior we wish to see when the frames are browsed. We can put this Cobweb protocol into the SMV symbolic model checker and ask if its structure behaves in ways we want it to:

- Is it possible that in some way one can end up with the table of contents in one of the two contents frames? This query translates to checking for a reachable state where there is a green token in place A, B, C or D.
- Is it possible that in some way one can end with a content page in the table of contents frame? We need to check for a reachable state with a red or blue token in the place associated with the table of contents.
- Is it possible to end up with two non-sequenced pages in the content pages? We need to check if it is always the case that we have the pair blue and red in A and B, B and C, C and D, or D and A.
- Is it true that the table of contents is always visible in the left frame? Translation: Is there a green token in that place at all times?

For this simple example with just 3 frames and 5 content pages it is relatively easy to find the affirmative answers in all cases by just reasoning with the graphic model. Conversion of the CPN in Figure 7 into SMV input and running the model checker on it gives the same results. Manual inspection to determine such properties is, of course, impossible for Web pages with many frames and hundred or (even dozens) of content pages.

Deriving CobWeb protocols from HTML source

We do not expect authors of frames-based HTML source text to generate the complex CPNs that the model checker works on. Instead these models can be derived from the HTML source directly. An author would never see a CobWeb model or SMV input. She would, however, have to produce CTL specifications to describe the browsing behavior the frames should exhibit. We address this issue at the end of the example.

There are a finite number of HTML source files containing the text of any frame set and its contents. These files are parsed, and from the HTML we extract all frame names, all target names in the anchor tags, all source URIs, and while parsing we build a representation of the interconnections among the files. Using these pieces we assign colors to frames, assign places to content URIs, and transitions to the link anchors from which the information is parsed. These components are then used to produce the SMV modules as described in the previous section. CTL specifications are added to the main module and SMV invoked on the input.

If the HTML source is incorrect (that is, if one of the CTL specifications is not true for the model), the SMV output will give an indication of which frames and which content pages are involved in the browsing sequence that causes the CTL formulae to fail. This gives the author some assistance in debugging the HTML sources.

Applying our solution in practice, then, requires these steps of an author:

- an author writes specifications of the desired behavior
- the author writes the HTML frames
- the author runs a tool to derive the CPN model of his frames as SMV input
- the author applies a model checker to the derived model to see if the HTML provides the frames behavior indicated by the specs
- the author repairs the HTML sources, if necessary, guided by the model checker counter examples

Writing CTL formulae

Writing specs of the desired frames behavior means creating the CTL temporal logic formulae that express the dynamics that should happen at browse time. While the syntax and semantics of CTL would have to be learned for an author to do this directly, there are several points we can note. First, the task of writing CTL assertions is no more intellectually challenging than writing SQL queries to manipulate a relational database. This is not to say it is trivial, but it is something that can be mastered by people with some technical training. Secondly, an approach can be taken that is similar to the manner in which non-technical people use relational databases like Microsoft Access (which hundreds of thousands do). SQL queries can be formed via filling out forms and the SQL syntax need never be seen or understood; the same can be done with CTL formulae. Finally, an even simpler, though more limited approach can be used. A large palette of common frames properties, pre-written as CTL formulae, can be provided in an authoring environment for complex frame-based Web documents. An author would simply select from the menu to have the model checker verify that his/her document exhibited the chosen behavior property.

6. CONCLUSIONS

We have presented a technique to allow authors of Web documents to create complex frame structures and verify that they behave "correctly" once they are authored. "Correctly" is defined by a collection of CTL temporal logic specifications that describe how content pages are to move throughout the frames in the frame set as browsing progresses.

We are still investigating the reverse manner of using the model checker with frames. It would be desirable to author frames by first writing the CTL specs that describe the behavior you want, and letting your favorite structured Web authoring environment derive a frame structure that makes all specifications true. We do not yet have an algorithm to do this. Even if one exists, it may not be fully satisfying, as it would require a considerable effort for an author to produce a *complete* set of CTL specs; i.e., one that captured all needed aspects of the frames behavior. The usage we have described does not require a complete set of CTL specs; they are given as *necessary* behaviors, but not as the *only* behaviors.

Though we have not developed it in detail, we think the same methods we have presented for HTML will work (with some syntax modifications) for XHTML documents.

ACKNOWLEDGEMENTS

This work was supported with funds from the National Science Foundation, grant # 9732577 to the University of North Carolina.

7. REFERENCES

[1] D. Stotts, R. Furuta, "Petri-Net-Based Hypertext: Document Structure with Browsing Semantics," ACM TOIS, vol. 7, pp. 3-29, 1989.

[2] D. Stotts, R. Furuta, C. Ruiz Cabarrus, "Hyperdocuments as Automata: Verification of Trace-based Browsing Properties by Model Checking, " ACM Trans. on Information Systems, vol. 16, no. 1, January 1998, pp. 1-30.

[3] R. Furuta, D. Stotts, "Interpreted Collaboration Protocols and their use in Groupware Prototyping, " Proc. of the 1994 ACM Conf. on Computer Supported Cooperative Work (CSCW '94), Research Triangle Park, NC, October 1994, pp. 121-131.

[4] B. Ladd, M. Capps, D. Stotts, R. Furuta, "Multi-head/Multi-tail Mosaic: Adding Parallel Automata Semantics to the Web, " World Wide Web Journal, O'Reilly and Assoc. Inc., vol. 1 (Proc. of the 4th Int'l WWW Conference, Boston, December 11-14, 1995), pp. 433-440.

[5] M. Capps, B. Ladd, D. Stotts, "Enhanced Graph Models in the Web: Multi-client, Multi-head, Multi-tail Browsing, " Computer Networks and ISDN Systems, vol. 28 (Proc. of the 5th WWW Conf., May 6-10, 1996, Paris), pp. 1105-1112.

[6] D. Stotts, J. Prins, L. Nyland, T. Fan, "CobWeb: Tailorable, Analyzable Rules for Collaborative Web Use," Technical Report CS-98-307, Dept. of Computer Science, University of North Carolina, Chapel Hill, 1998. http://www.cs.unc.edu/~stotts/papers/cobweb.ps

[7] J. Navon, "Specification and Semi-Automated Verification of Coordination Protocols for Collaborative Software Systems," Ph.D. Thesis, Dept. of Computer Science, Univ. of North Carolina at Chapel Hill, June 2001; advisor: D. Stotts.

[8] "Storyspace: A Hypertext Tool for Writers and Readers", http://www.eastgate.com/Storyspace.html

[9] K. Jensen, G. Rozenberg, "High-level Petri Nets", Springer-Verlag, Berlin, 1991.

[10] E. M. Clarke, E. A. Emerson, A. P. Sistla, "Automatic verification of finite-state concurrent systems using temporal logic specifications," ACM Trans. on Programming Languages and Systems, vol. 8, pp. 244-263, 1986.

[11] A. Pnueli, "A temporal logic of concurrent programs," Theoretical Computer Science, pp. 45-60, 1981.

[12] J. R. Burch, E. M. Clarke, K. L. McMillan, "Symbolic Model Checking: 10^20 States and beyond," Information and Computation, vol. 98, pp. 142-170, 1992.

[13] K. McMillan, "Symbolic Model Checking," Carnegie Mellon University, 1992.

[14] "On-the-fly, LTL Model Checking with SPIN", research report, Bell Labs Research, New Jersy, 2001, http://netlib.bell-labs.com/netlib/spin/whatispin.html