Modeling and Verification of Interactive Flexible Multimedia Presentations Using PROMELA/SPIN *

Ramazan Savaş Aygün and Aidong Zhang

Department of Computer Science and Engineering,
State University of New York at Buffalo,
Buffalo NY 14260-2000, USA,
{aygun,azhang}@cse.buffalo.edu

Abstract. The modeling and verification of flexible and interactive multimedia presentations are important for consistent presentations over networks. There has been querying languages proposed whether the specification of a multimedia presentation satisfy inter-stream relationships. Since these tools are based on the interval-based relationships, they cannot guarantee the verification in real-life presentations. Moreover, the user interactions which change the course of the presentation like backward and skip are not considered in the presentation. Although there have been conceptual models proposed using Petri-Nets, it is very difficult for an ordinary author design Petri-Nets and verify the requirements. Using PROMELA/SPIN, it is possible to verify the temporal relationships between streams using our model including user interactions that change the course of the presentation. Since the model considers the delay of data, the author is assured that the requirements are really satisfied.

1 Introduction

A multimedia presentation is a presentation of multimedia streams in a synchronized fashion. There have been models proposed for the management of multimedia presentations. The synchronization specification languages like SMIL [11] have been introduced to properly specify the synchronization requirements. Multimedia query languages are developed to check the relationships defined in the specification [5]. These tools check the correctness of the specification. However, the synchronization tools may have some limitations and may not satisfy all the requirements. Moreover, the specification does not include user interactions. The previous query-based verification techniques cannot verify whether the system is really in a consistent state after an user interaction.

There are also verification tools to check the integrity of multimedia presentations [7]. The user interactions are limited and interactions like backward and skip are ignored. This kind of interactions are hard to model. The Petri-Nets are

* This research is supported by NSF grant IIS-9733730.

also used to verify the specification of multimedia presentations [9]. But Petri-Net modeling requires immense Petri-Net modeling for each interaction possible. Authors usually do not have much information about Petri-Nets.

PROMELA/SPIN is a powerful tool for modeling and verification of software systems [6]. Since PROMELA/SPIN traces all possible executions among parallel running processes, it provides a way of managing delay in the presentation of streams. In this paper, we discuss the properties that should be satisfied for a multimedia presentation. We analyze the complexity introduced by user interactions. The interactions which change the course of the presentation like backward and skip are also investigated. The experiments are conducted for parallel, sequential, and synchronized presentations.

This paper is organized as follows. The synchronization model is explained in Section 2. The conversion to PROMELA and abstractions are discussed in Section 3. Section 4 explains the properties that should be satisfied for a multimedia presentation. Section 5 discusses the experiments. The last section concludes our paper.

2 Multimedia Presentations

The synchronization model is based on synchronization rules [2]. Synchronization rules form the basis of the management of relationships among the multimedia streams. Each synchronization rule is based on the Event-Condition-Action (ECA) rule.

Definition A *synchronization rule* is composed of an event expression, condition expression, and action expression which can be formulated as:

on *event expression* **if** *condition expression* **do** *action expression.*

Event expression and condition expression are obtained by composing events and conditions using boolean operators (&& and ||), respectively. Action expression is a list of actions. A synchronization rule can be read as: When the *event expression* is satisfied if the *condition expression* is valid, then the actions in the *action expression* are executed.

Definition. An *event* is represented with $source(event\_type[, event\_data])$ where *source* points the source of the event, *event_type* represents the type of the event and *event_data* contains information about the event. Event source can be the user or a stream. Optional event data contains information like a realization point. The goal in inter-stream synchronization is to determine when to start and to end streams. The start and end of streams depend on multimedia events. The user has to specify information related with the stream events. Allen [1] specifies 13 temporal relationships. Relationships *meets, starts*, and *equals* require the *Init_Point* event for a stream. Relationships *finishes* and *equals* require the *End_Point* event for a stream. Relationships *overlaps* and *during* require *realization* event to start (end) another stream in the mid of a stream. The relationships *before* and *after* require temporal events since the gap between two streams can only be determined by time. Temporal events may be *absolute* with respect to a specific point in a presentation (e.g. the beginning of a presentation).

Temporal events may also be *relative* with respect to another event. Users can also cause events such as start, pause, resume, forward, backward, and skip.

Definition. A *condition* in a synchronization rule is a 3 tuple $C = condition\ (t_1, \theta, t_2)$ where θ is a relation from the set $\{=, \neq, <, \leq, >, \geq\}$ and t_i is either a state variable that determines the state of a stream, or presentation, or a constant. A condition indicates the status of the presentation and its media objects. The most important condition is whether the direction of the presentation is forward. The receipt of the events matter when the direction is forward or backward. Other types of conditions include the states of media objects.

Definition. An *action* is represented with $action\_type(stream[, action\_data], sleeping\_time)$ where *action_type* needs to be executed for *stream* using *action_data* as parameters after waiting for *sleeping_time*. Action_data can be the parameter for speeding, skipping, etc. An action indicates what to execute when conditions are satisfied. *Start*ing and *end*ing a stream, and *displaying* or *hiding* images, slides, and text are sample actions. For backward presentation, *backward*ing is used to backward and *backend* is used to end in the backward direction. There are two kinds of actions: *Immediate* Action and *Deferred* Action. *Immediate* action is an action that should be applied as soon as the conditions are satisfied. *Deferred* action is associated with some specific time. The deferred action can only start after this *sleeping_time* has been elapsed. If an action has started and had not finished yet, that action is considered as an alive action.

2.1 Elements of a Multimedia Presentation

The basic component of a multimedia presentation is a stream. In our model, a multimedia presentation may have a *container* consisting of containers or other streams. This allows grouping of streams and creation of subpresentations. The containers have *init* and *end* points. This means that the container initiates its presentation and the container ends either when one or more of its components end or is terminated by other containers or streams.

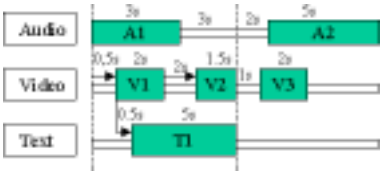


Fig. 1. Sample Presentation.

A sample presentation is depicted in Figure 1. There are 6 stream elements: A1, A2, V1, V2, V3, and T1. A1 and A2 are audio elements. V1, V2, and V3 are video elements and T1 is a text element. There are 4 containers in the presentation: sequential presentation of V1 and V2 (SEQ1), parallel presentation

of A1, T1, and SEQ1 (PAR1), parallel presentation of A2 and V3 (PAR2) and sequential presentation of PAR1 and PAR2 (MAIN).

2.2 Receivers, Controllers and Actors

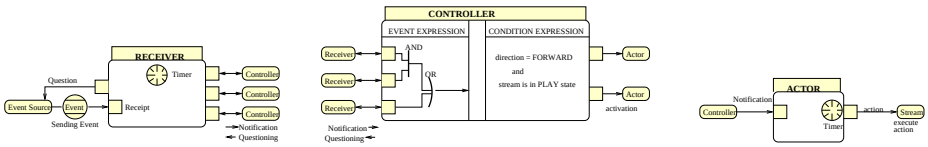


Fig. 2. (a) A receiver object (b) A controller object (c) An actor object.

The synchronization model is composed of three layers, the receiver layer, the controller layer, and the actor layer. Receivers are objects to receive events. Controllers check composite events and conditions about the presentation such as the direction. Actors execute the actions once their conditions are satisfied.

Definition 1. A receiver is a pair $R = (e, C)$ where e is the event that will be received and C is a set of controller objects.

Receiver R can question the event source through its event e. When e is signaled, receiver R will receive e. When receiver R receives event e, it sends information of the receipt of e to all its controllers in C. A receiver object is depicted in Figure 2(a). There is a receiver for each single event. The receivers can be set and reset by the system anytime.

Definition 2. A controller is a 4-tuple $C = (R, ee, ce, A)$ where R is a set of receivers; ee is an event expression; ce is a condition expression; and A is a set of actors.

Controller C has two components to verify, composite events ee and conditions ce about the presentation. When the controller C is notified, it first checks whether the event composition condition ee is satisfied by questioning the receiver of the event. Once the event composition condition ee is satisfied, it verifies the conditions ce about the states of media objects or the presentation. After the conditions ce are satisfied, the controller notifies its actors in A. A controller object is depicted in Figure 2(c). Controllers can be set or reset by the system anytime.

Definition 3. An actor is a pair $A = (a, t)$ where a is an action that will be executed after time t passes.

Once actor A is informed, it checks whether it has some sleeping time t to wait for. If t is greater than 0, actor A sleeps for t and then starts action a. If t is 0, action a is an immediate action. If $t > 0$, action a is a deferred action. An actor object is depicted in Figure 2(b).

13 receivers, 12 controllers, and 15 actors will be generated for the presentation given in Figure 1. These are listed in Figure 3.

2.3 Timeline

If multimedia presentations are declared in terms of constraints, synchronization expressions or rules, the relationships among streams will not be explicit because they only keep the relationships that are temporally adjacent or overlapping. The status of the presentation must be known at any instant. In our work, the timeline object keeps track of all temporal relationships among streams in the presentation.

Definition 4. A timeline object is a 4-tuple $T = (receiverT, controllerT, actorT, actionT)$, where $receiverT$, $controllerT$, $actorT$, and $actionT$ are time-trackers for receivers, controllers, actors, and actions, respectively.

The time-trackers $receiverT$, $controllerT$, $actorT$, and $actionT$ keep the expected times of the receipt of events by receivers, the expected times of the satisfaction of the controllers, the expected times of the activation of the actors and the expected times of the start of the actions, respectively. Since skip and backward operations are allowed, alive actions, received or not-received events, sleeping actors and satisfied controllers must be known for any point in the presentation. The generation of the timeline is explained in [10].

The timeline for receivers, controllers, and actors for the presentation shown in Figure 1 is depicted in Figure 3. On top of the figure the receivers, the controllers, and the actors for the presentation are listed. The four time-trackers are shown at the bottom side. The receivers and controllers are ordered according to their expected satisfaction time. Only actors which have a sleeping time greater than 0 are displayed. The name of the actor shows its activation (sleeping time) and the underlined actor shows the ending of sleeping time. The actions are also displayed in the same way. The name of the container or the stream shows its starting time and if it is underlined it shows the ending time. At a time instant, if a stream or a container has the same starting time as its container, the main container is shown in the timeline.

3 Modeling of a Multimedia Presentation

3.1 Presentation

The presentation can be in idle, initial, play, forward, backward, paused, and end states (Figure 4). The presentation is initially in the idle state. When the user clicks START button, the presentation enters play state. The presentation enters end state when the presentation ends in the forward presentation. The presentation enters the initial state when it reaches its beginning in the backward presentation. The user may quit the presentation at any state. Skip can be performed in play, forward, backward, initial, and end states. If the skip is clicked in play, forward, and backward states, it will return to the same state unless skip to initial or end state is not performed. If the presentation state is in end or initial states, skip interaction will put into the previous state before reaching these states (Figure 4 (b)). The presentation changes states as the user clicks on the button. The most important state variable of the presentation is the direction.

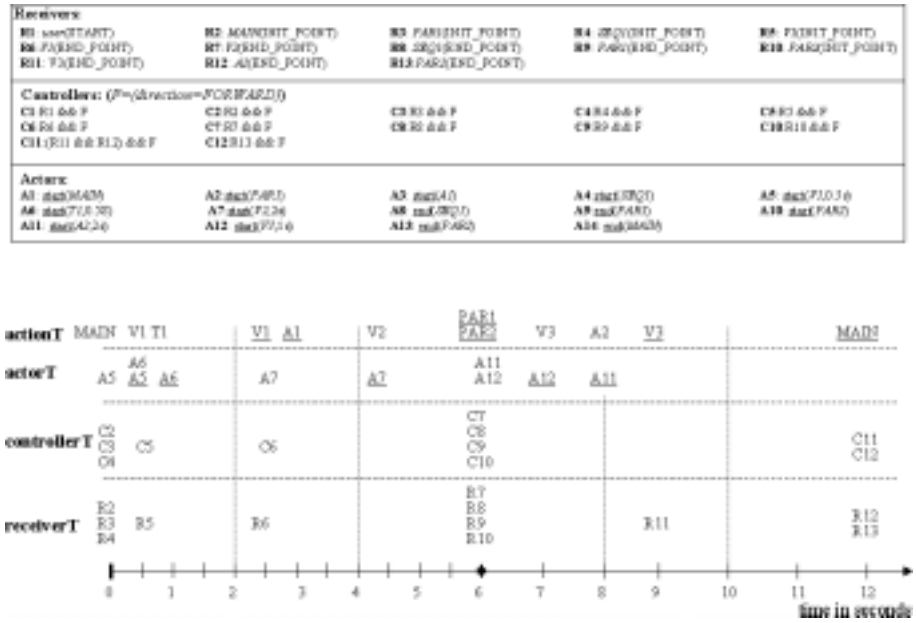


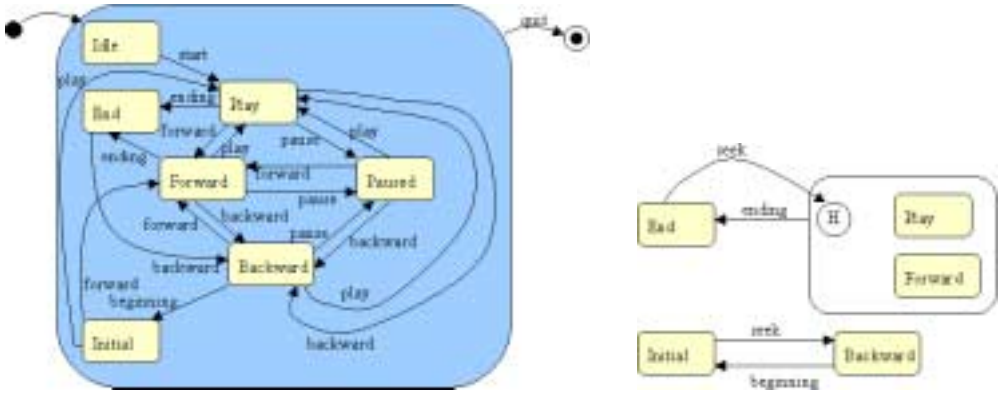
Fig. 3. The timeline.

3.2 Containers and Streams

A container may enter 4 states. It is in IdlePoint state initially. Once started, a container is at InitPoint state in which it starts the containers and streams that it contains. After the InitPoint state, a container enters its RunPoint state. In RunPoint state, a container knows that it has some streams that are being played. When all the streams it contains reach their end or when the container is notified to end, it stops execution of the streams and signals its end and then enter idle state again. In the backward presentation, the reverse path is followed (Figure 5 (a)).

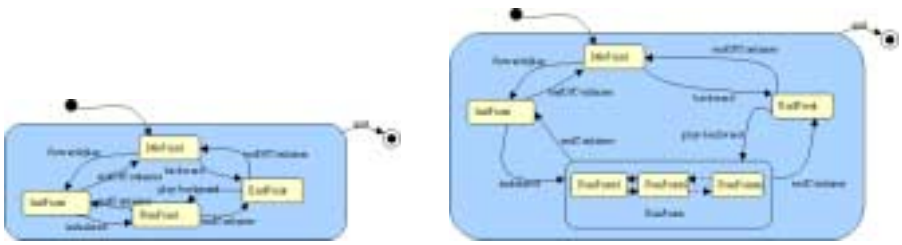
A stream is similar to the container. Since the number of states grow exponentially, some abstractions has to be made on modeling a stream. Since we are interested in interstream relationships, the points which affect the interstream relationships will be considered. From the modeling point of view, if the displaying or playing specific segment of a stream does not affect the playout of the presentation, there is no need to handle each segment of the stream.

If a stream does not signal any event except its start and end, the stream enters the same 4 states as a container. If a stream has to signal an event, a new state is added to RunPoint state per event. So after the stream signals its event, it will be still in the RunPoint state (playing mode) (Figure 5 (b)). Since the realization for the backward presentations will also be considered, there will be another event (also state) for the backward presentation. In this sense, each realization event will add two states. One will be used for the forward



(a) (b)

Fig. 4. The presentation states, (a) general state transitions (b) state transition for skip



(a) (b)

Fig. 5. (a) container states (b) stream states

presentation and the other will be used for the backward presentation. The following is a PROMELA code for playing a stream.

```
1   proctype playStream (byte stream) {
2   #if (FC==3 || FC==4 || FC==5 || FC==6)
3   progressIdleStreams:
4   #endif
5   do
6   #if FC!=4
7      :: atomic{ (eventHandled && getState() == RUN) &&
8           (getStream(stream) == INIT_POINT) ->
9           printf("Starting stream %d",stream);
10          setStream(stream, RUN_POINT);
11          if
12          :: (stream==A1)->timeIndex=1;
13          :: else -> skip;
14          fi; }
15     :: atomic{ (eventHandled && getState() == RUN) &&
16          (getStream(stream) == RUN_POINT) ->
```

```
17          printf("Playing stream %d",stream);
18          setStream(stream, END_POINT);}
19     :: atomic{ (eventHandled && getState() == RUN) &&
20          (getStream(stream) == END_POINT) ->
21          to_end: printf("Ending stream %d",stream);
22                  setStream(IDLE_POINT);
23                  signalEvent(stream,END_POINT) }
24  #endif
25  #if (FC!=3 && FC!=5 && FC!=6)
26     :: atomic{ (eventHandled && getState() == BACKWARD) &&
27          (getStream(stream) == INIT_POINT) ->
28          to_init: printf("Ending backwarding stream %d",stream);
29                  setStream(IDLE_POINT);
30                  signalEvent(stream,INIT_POINT);}
31     :: atomic{ (eventHandled && getState() == BACKWARD) &&
32          (getStream(stream) == RUN_POINT) ->
33          printf("Playing stream %d backward",stream);
34          setStream(stream, INIT_POINT);}
35     :: atomic{(eventHandled && getState() == BACKWARD) &&
36          (getStream(stream) == END_POINT) ->
37          to_backward: printf("Backwarding stream %d",stream);
38                  if
39                  :: (stream==A1)->timeIndex=1;
40                  :: else -> skip;
41                  fi;
42                  signalEvent(stream,END_POINT);
43                  setStream(stream,RUN_POINT); }
44  #endif
45     :: atomic{ (eventHandled && getState() == QUIT) ->
46          to_playStream_quit: goto playStream_quit;}
47     :: else -> skip;
48     od;
49     playStream_quit: skip;
50  }
```

The #if directives are used for hard-coded fairness constraints. There are 3 states for forward and backward presentations. The cases at lines 7, 14 and 15 correspond to forward presentation. The cases at lines 26, 31 and 35 correspond to the backward presentation. The case at line 45 is required to quit the process. The else statement at line 49 corresponds to IdlePoint state. Streams signal events as they reach the beginning and end (lines 23 and 30). The variable *eventHandled* is used to check whether the system enters a consistent state after an user interaction. The *atomic* command enables execution of statements in a single step thus reduces the complexity and increases safety. The checking and updating the stream state has to be performed in a single step since the stream state may also be updated by the system after an user interaction. Labels like *to_init, to_end,* and *playStream_quit* are added to create LTL formulas related with these points of the presentation.

3.3 Receivers, Controllers and Actors

A receiver is set when it receives its event. As long as there is no user interaction, a receiver will stay at this state for the rest of the presentation. Thus a controller which requires a receipt of this event can be satisfied later. When a controller is satisfied, it activates its actors. And to disable the reactivation of the actors, the controller is reset. An actor is either in idle or running state to start its action after sleeping. Once it wakes up, it starts its action and enters the idle state. The following is a code for receiver definition (lines 51-52), controller satisfaction (lines 54-59) and actor activation (lines 61-64). The expression "receivedReceiver(receiver_Main_INIT)" (line 52) corresponds to the receipt of the event when the main container starts. The expression "setActorState(...,RUN_POINT)" activates the actors (line 58-59). The expression "activateActor(actor_Main_START)" (line 63) elapses the time and the action follows (line 64).

```
51  #define Controller_Main_START_Condition
52     (receivedReceiver(receiver_Main_INIT) && (direction==FORWARD))
53
54  :: atomic{(eventHandled
55     && !(satisfiedController(controller_Main_START))
56     && Controller_Main_START_Condition) ->
57          setController(controller_Main_START);
58          setActorState(actor_A1_START,RUN_POINT);
59          setActorState(actor_A2_START,RUN_POINT)}
60
61  :: atomic{(eventHandled
62     && getActorState(actor_Main_START) == RUN_POINT) ->
63          activateActor(actor_Main_START);
64          setContainerState(Main,INIT_POINT);}
```

3.4 User and User Interface

The user interface provides 7 buttons: START, PLAY, PAUSE, FORWARD, BACKWARD, SKIP, and QUIT. Each button may enter two states in the model. A button is either in enabled or disabled state. As the presentation changes states, the states of the buttons may change. Figure 4 shows the possible state transitions with enabled user interactions. The user interface is based on the model presented at [3].

For example, if a skip is requested to the mid (6 seconds) of the presentation that is shown in Figure 1, the timeline will be followed in Figure 3. Receivers R2, R3, R4, R5, R6, R7, R8, R9, and R10 are assumed to receive their events. Receivers R11, R12, and R13 are assumed that they did not receive their events. Controllers C2, C3, C4, C5, C6, C7, C8, C9, and C10 are assumed to be satisfied. C11 and C12 are assumed not to be satisfied. A satisfied controller cannot notify its controllers. It is assumed that it already notified its actors. At the middle point, there is no sleeping actor. The actors A11 and A12 are activated. So, all

the actors should be set to their original time. MAIN container should be set to running point. V1, T1, V2, and A1 should be idle. PAR1 should be idle too. PAR2 should be set to its INIT_POINT so that it can start the streams that it contains. V3 and A2 should also be set to IDLE_POINT.

There are infinite number of skips that can be performed by the user. The timeline shown in Figure 3 is divided into pieces where the streams perform similar behavior in each piece. There are 21 pieces which are determined by starting, ending actors, and actions. So, it is only possible to perform 21 skips.

On the other hand, the backward modifies the direction of the presentation. The synchronization model needs to synchronize after changing direction since streams may proceed at different speeds. To synchronize, the time at that instant should be known. We define a time index which is initially 0 and can be the number of pieces at most. Some specific streams are allowed to increase or decrease after the time index and the current time index can be determined (lines 12 and 19). The necessary actors, actions, receivers, and controllers are set and reset after changing the direction.

4 Specification

Two basic properties that should be checked are *safety properties* and *liveness properties*. *Safety properties* assert that the system may not enter undesired state or "something bad will not happen". *Liveness properties* on the other hand assure that system executes as expected me or "something good will eventually happen". In LTL, it is possible to set fairness constraints. Fairness constraint is satisfied infinitely often in fair paths of executions. Fairness constraints are necessary to prove some properties of the system. For example, to prove that "stream A is played before stream B", no skip operation should be allowed. Skip operation may skip to any segment of the presentation and thus violating the above expression. To prove the properties of the system, we have 2 fairness constraints:

Fairness Constraint 1 *The user is only allowed to click START button and clicks START button and no user Interaction is allowed after that. This constraint is expressed as:* $\Box\ (userStart \rightarrow \Diamond noInteraction)$

Fairness Constraint 2 *The user always clicks enabled button. This is expressed as* $\Box\ (userClickButton \rightarrow buttonEnabled)$

If a property is stated as undesirable, the system should not allow it. We first start with the properties about transitions that are allowed by buttons.

Property 1. Clicking button for START enables buttons for PAUSE, FORWARD, and BACKWARD, and it changes the simulations state to RUN. (requires fairness constraint 2)

Property 2. Clicking button for PAUSE enables buttons for PLAY, FORWARD, and BACKWARD and it changes the presentation's state to PAUSED. (requires fairness constraint 2)

Property 3. The buttons for BACKWARD and SKIP are enabled and the buttons for START, PLAY, PAUSE, and FORWARD are disabled after the presentation reaches its end. (requires fairness constraint 2)

Property 4. The user interface should ignore if the user clicks a disabled button. (requires fairness constraint 2)

Property 5. The button for PAUSE is enabled only when the presentation is in RUN, FORWARD, or BACKWARD states. (requires fairness constraint 2)

Property 6. The button for SKIP is enabled when the presentation is in RUN, FORWARD, BACKWARD, INITIAL or END states. (requires fairness constraint 2)

Property 7. The buttons for PLAY, FORWARD, and BACKWARD are enabled when the presentation is in PAUSED state. (requires fairness constraint 2)

Property 8. The button for START is enabled only when the presentation is in IDLE state. (requires fairness constraint 2)

Property 9. The button for PAUSE is enabled outside RUN, FORWARD, and BACKWARD. (undesirable, requires fairness constraint 2)

Property 10. Buttons for START, PAUSE, PLAY, FORWARD, and BACKWARD are in enabled condition at any particular time. (undesirable, requires fairness constraint 2)

Some refinements are needed to convert the properties to LTL formulas. In the following formulas, *actionButtonClicked* corresponds to successful clicking *Button* when the button is enabled. *actionToState* corresponds to state transition to *State* after the *action*. ButtonEnabled corresponds to *Button* is enabled. *UserButton* corresponds to clicking of *Button* by the user. Some of the specification patterns are presented in [4, 8]. These specification patterns can be used in the verification. For each property, the following LTL formulas are generated.

LTL 1 $\Box\ (actionStartClicked \rightarrow\ \Diamond\ actionToRun)$

LTL 2 $\Box\ (actionPauseClicked \rightarrow\ \Diamond\ actionToPaused)$.

LTL 3 $\Box\ (backwardEnabled\ \wedge\ skipEnabled \wedge !startEnabled \wedge !playEnabled \wedge !pauseEnabled \wedge !forwardEnabled \rightarrow stateEnd)$

LTL 4

1. $\Box\ ((userStart \wedge !startEnabled) \rightarrow !eventHandled\ \mathbf{U}\ userInterfaceIgnore)$
2. $\Box\ ((userPause \wedge !pauseEnabled) \rightarrow !eventHandled\ \mathbf{U}\ userInterfaceIgnore)$
3. $\Box\ ((userPlay \wedge !playEnabled) \rightarrow !eventHandled\ \mathbf{U}\ userInterfaceIgnore)$
4. $\Box\ ((userForward \wedge !forwardEnabled) \rightarrow !eventHandled\ \mathbf{U}\ userInterfaceIgnore)$
5. $\Box\ ((userBackward \wedge !backwardEnabled) \rightarrow !eventHandled\ \mathbf{U}\ userInterfaceIgnore)$
6. $\Box\ ((userSkip \wedge !skipEnabled) \rightarrow !eventHandled\ \mathbf{U}\ userInterfaceIgnore)$

7. $\Box\ ((userQuit \wedge !quitEnabled) \rightarrow !eventHandled\ \mathbf{U}\ userInterfaceIgnore)$

LTL 5 $\Box\ ((stateInitial \vee stateEnd \vee statePaused \vee stateIdle) \rightarrow !pauseEnabled)$

LTL 6 $\Box\ ((stateRun \vee stateForward \vee stateBackward \vee stateEnd) \rightarrow skipEnabled)$

LTL 7 $\Box\ (statePaused \rightarrow (playEnabled \wedge forwardEnabled \wedge backwardEnabled)$

LTL 8 $\Box\ ((stateRun \vee stateEnd \vee statePaused \vee stateForward \vee stateBackward \vee stateInitial) \rightarrow !startEnabled)$

LTL 9 $\Box\ ((stateInitial \vee stateEnd) \rightarrow !pauseEnabled)$

LTL 10 $\Box\ (startEnabled \wedge pauseEnabled \wedge playEnabled \wedge forwardEnabled \wedge backwardEnabled)$

A liveness property that should be checked whether the presentation reaches to its end once it starts.

Property 11. The presentation will eventually end. (requires fairness constraints 1 and 2)

LTL 11 $\Box\ (stateRun \rightarrow !\ \Diamond\ stateEnd)$

There are also some properties that should be satisfied for streams. If a stream is in RunPoint state, the stream cannot be started by other streams. This is assumed to be a wrong attempt. So, a warning should be signaled to the author. In the same way, a stream cannot be terminated if it is already idle. The properties are as follows:

Property 12. A stream can be started if it is already active. (undesirable, requires fairness constraints 1 and 2)

Property 13. A stream can be terminated if it is already idle. (undesirable, requires fairness constraints 1 and 2)

The LTL formulas will be:

LTL 12 $\Diamond\ (streamRunPoint\ \mathbf{U}\ streamInitPoint)$

LTL 13 $\Diamond\ (streamIdlePoint\ \mathbf{U}\ streamEndPoint)$

In [7], some properties between two consecutive user interactions based on time are verified. In a distributed system, these constraints cannot be satisfied due to delay of data. For example, pause operation for a stream may be performed within t seconds after the start of the presentation where $0 < t < d$ and d is the duration of the stream. In our model, the user cannot change the state of a stream directly but he/she can change the state of the presentation thus changing the state of a stream indirectly. Since there are relationships among

streams and containers, these can start and end each other. In our case, time is associated with actors. Since there is no delay in passing of time, the actor elapses its time right away once it is activated.

Further checks can be performed based on the relationships among streams. Based on Allen's temporal relationships, the following properties may be checked:

Property 14. Stream A is before stream B. (requires fairness constraints 1 and 2)

Property 15. Stream A starts with stream B. (requires fairness constraints 1 and 2)

Property 16. Stream A ends with stream B. (requires fairness constraints 1 and 2)

Property 17. Stream A is equal to stream B. (requires fairness constraints 1 and 2)

Property 18. Stream B is not during stream A. (undesirable, requires fairness constraints 1 and 2)

Property 19. Stream B does not overlap stream A. (undesirable, requires fairness constraints 1 and 2)

Let $P = streamA\_InitState, Q = streamA\_EndState, R = streamA\_IdleState$, $K = streamB\_InitState$, $L = streamB\_EndState$, $M = streamB\_IdleState$. The LTL formulas will be as follows:

LTL 14 $(Q\ \mathbf{U}\ (R\ \wedge\ M)\ \mathbf{U}\ K)$

LTL 15 $\Diamond(P\ \wedge\ K)$

LTL 16 $\Diamond(Q\ \wedge\ L)$

LTL 17 $\Diamond(P\ \wedge\ K\ \wedge\ \Diamond\,(Q\ \wedge\ L))$

LTL 18 $!(\Diamond(P\ \wedge\ \Diamond K)\ \vee\ \Diamond\,(Q\ \wedge\ \Diamond L))$

LTL 19 $!(\Diamond(Q\ \wedge\ \Diamond K)\ \vee\ \Diamond\,(L\ \wedge\ \Diamond Q))$

One of the basic queries is whether all streams are played or not. If one of the streams is not played, this may be considered as an undesired behavior and the author may correct it.

Property 20. Stream A is played. (requires fairness constraints 1 and 2)

LTL 20 $\Diamond(P\ \wedge\ \Diamond\,Q)$

For a multimedia presentation, the states of streams that are possible to visit in the backward presentation should also be reachable in the forward presentation. We call this property as *backward consistency* of a presentation and term such a presentation as *backward consistent* presentation. If we show the existence of a state which is not reachable in forward presentation while it is reachable in backward presentation, it is not backward consistent.

There are a couple of ways writing the LTL formula to check the backward consistency of a presentation. In one way, the state which is reachable in the forward presentation is given (if exists) as a contradictory example. This complicates the verification since we also need to distinguish the states that are reachable in the forward presentation. Another problem is that the presentation may enter in a inconsistent state after backward operation and from that inconsistent state, the desired state may be reachable in the forward presentation. So, the property is stated as two fold.

Property 21.

1. The *state* is reachable in forward presentation (undesirable, requires fairness constraints 1 and 2)
2. It is possible to reach the *state* in the backward presentation. (requires fairness constraint 2)

Notice that first part requires the existence check. The corresponding LTL formulas will be as follows:

LTL 21

1. $!\Diamond state$
2. $\Box !state$

If the first part is wrong, then the second part is verified. The number of states that need to be checked is $\lfloor m^n \rfloor$ where m is the number of states that a stream may enter and n is the number of streams. Eventually, we need to convert the previous property into the following one:

Property 22.

1. The *state* is reachable in forward presentation (undesirable, requires fairness constraints 1 and 2)
2. It is possible to reach the *state* after user interactions. (requires fairness constraint 2)

The previous LTL formula, in fact, corresponds to this property.

5 Experiments and Analysis

We firstly developed a complex model to handle the user interactions. Since this user interface increases the number of initial states significantly, we removed

the user interface during verification. Only buttons change their states as part of the user interface. The forward (fast) interaction is not allowed to reduce the complexity of the model since we are not interested in the speed of the presentation. We are rather interested in the direction change. The backward and play interactions are enough to verify the model.

Different kinds of presentations have been used to check the correctness of the presentation model. We considered the number of streams and their organization. The streams are presented in a sequential order or in parallel. If the streams are presented in parallel, they may also be presented in a synchronized fashion.

The fairness constraints are hard-coded in the presentation. For each interaction, there is a fairness constraint and these are hard-coded in the model (lines 2,6,25). FC==3, FC==4, FC==5, FC==6 and FC==7 correspond to interactions where only start, only backward, pause-resume, skip, and backward-play are allowed, respectively.

We first investigated the complexity of the number of streams and the organization when no interaction (except to start the presentation) is allowed. The results are given in Table 1.

Presentation	No of Streams	Depth	States	Transitions	Memory	Time
single	1	67	177	306	1.5	0:00.03
sequential	2	99	432	865	1.5	0:00.04
sequential	3	143	1021	2321	1.6	0:00.08
sequential	4	209	2347	5868	2.0	0:00.25
parallel	2	101	488	1021	1.5	0:00.05
parallel	3	139	1699	4642	1.8	0:00.13
parallel	4	173	6678	26132	2.8	0:00.76
synchronized	2	73	185	334	1.5	0:00.03
synchronized	3	78	201	398	1.5	0:00.05
synchronized	4	83	233	542	1.5	0:00.04

Table 1. Experiments without interaction.

When a new stream is added into the sequential presentation, there will be phases where the new stream starts, plays, and ends. The ending of stream does not add any complexity since they will all be idle at the end of the presentation. Since each stream adds 3 more phases, the number of states is nearly tripled after each addition of a stream in a sequential presentation. The complexity of number of states is $O(m^n)$ where n is the number streams in the sequential presentation and m is one less than the number of states that a stream may enter (to exclude idle state). In our experiments, m is 3. The running time and the depth also increases in the same way.

For a parallel presentation, there are more combinations of playing streams. In the parallel presentations, the streams may be interleaved. The number of possible interleavings for n streams which have m states is

$$I(n,m) = \frac{(2n)!}{(m!)^n}. \quad (1)$$

This explains the steep increase in running time, memory, states, transitions, and depth. Nevertheless, the running time is still within a second for 4 streams. The verification can be performed for a presentation having a fair number of parallel presentations. On the other hand, a synchronized parallel presentation's complexity is $O(n)$ for depth, transitions and running time but $O(2^n)$ for the states.

To evaluate the effect of user interactions, we tested user interactions separately. The experiments with pause-resume interactions are given in Table 2. The pause resume interactions increase the complexity in linear time. Therefore, the presentations having pause and resume interactions do not add more complexity and this is an expected result. But these interactions increased the initial number of states, depth and complexity.

Presentation	No of Streams	Depth	States	Transitions	Memory	Time
sequential	1	94	279	487	1.5	0:00.04
sequential	2	168	718	1435	1.6	0:00.06
sequential	3	304	1814	2246	4060	0:00.12
sequential	4	559	4564	11341	2.5	0:00.36
parallel	2	178	829	1810	1.6	0:00.07
parallel	3	258	3519	11174	2.1	0:00.32
parallel	4	423	22913	101443	6.1	0:02.76
synchronized	2	106	287	517	1.5	0:00.03
synchronized	3	111	303	591	1.5	0:00.04
synchronized	4	122	335	749	1.5	0:00.06

Table 2. Experiments with Pause-Resume

The experiments with skip interaction are given in Table 3. The time complexity of synchronized presentations is $O(2^n)$. On the other hand, the complexity of states for parallel presentations increased from $O(4^n)$ to (10^n). The complexity of states for sequential presentations increased from $O(3^n)$ to $O(4^n)$.

The experiments with backward interaction are given in Table 4. The play interaction is allowed along with the backward interaction. The time complexity of synchronized presentations is $O(2^n)$. On the other hand, the complexity of states for parallel presentations increased from $O(4^n)$ to (10^n). The complexity of states for sequential presentations increased from $O(3^n)$ to $O(4^n)$. These results show that the complexity of backward is similar to the skip. Since the direction of the presentation may change in the backward presentation, the number of initial states doubled and this caused severe exponential increase in the running time.

To realize the effects of interactions, experiments where all interactions are allowed are conducted. The complexity for sequential, parallel, and synchronized

Presentation	No of Streams	Depth	States	Transitions	Memory	Time
sequential	1	156	4476	7219	2.0	0:00.17
sequential	2	380	20622	34946	4.8	0:00.81
sequential	3	1085	102309	179545	21.7	0:04.80
sequential	4	2436	438514	784559	100	0:24.01
parallel	2	347	26914	44746	5.6	0:00.98
parallel	3	677	233890	402438	43.8	0:09.96
parallel	4	1356	2.34 K	4.15 K	473	2:00.59
synchronized	2	166	5020	8179	2.2	0:00.20
synchronized	3	176	6108	10171	2.5	0:00.27
synchronized	4	186	8284	14299	3.0	0:00.38

Table 3. Experiments with Skip

Presentation	No of Streams	Depth	States	Transitions	Memory	Time
sequential	1	216	8358	14898	2.6	0:00.31
sequential	2	561	34201	62691	7.1	0:01.45
sequential	3	1437	140408	260996	29	0:07.11
sequential	4	3157	596432	1.12 K	136	0:34.40
parallel	2	359	55780	101419	10	0:02.24
parallel	3	948	528264	978800	97	0:24.63
parallel	4	2031				
synchronized	2	228	9548	17254	2.9	0:00.39
synchronized	3	239	11912	22098	3.5	0:00.55
synchronized	4	250	16640	32154	4.7	0:00.81

Table 4. Experiments with Backward

presentations are similar to the backward and the skip (Table 5). The effects of interactions on types of presentations are depicted in Figure 6.

6 Evaluation

6.1 Contribution of PROMELA/SPIN

The previous work on checking the integrity of multimedia presentations deal with presentations that are presented in nominal conditions (i.e., no delay). SPIN verifier takes into account each possible state that the processes and elements of a presentation may enter. Since the processes may iterate at different states as long as they are enabled, this introduces processes proceeding at different speeds. From the perspective of a multimedia presentation, this may correspond to delay of data in the network. The SPIN verifier checks the properties of a presentation also at the worst case. The unexpected false presentations are reported by contradictory examples.

SPIN enables verification of LTL formulas. LTL formulas requires tracing all the execution paths. For example, it may be possible that two streams may start

Presentation	No of Streams	Depth	States	Transitions	Memory	Time
sequential	1	745	24586	46364	4.8	0:00.92
sequential	2	1954	103197	200756	18.5	0:04.71
sequential	3	5717	424039	846276	85	0:23.13
sequential	4	18676	2.21 K	4.50 K	500	2:27.68
parallel	2	1509	184092	351747	31	0:07.87
parallel	3	3571	1.75 K	3.42 K	318	1:30.09
synchronized	2	823	30299	57461	6.1	0:01.30
synchronized	3	869	38179	74420	8.3	0:01.86
synchronized	4	871	53939	109319	12	0:02.75

Table 5. Experiments with all interactions allowed

at the same time. What we are really interested is whether these two streams will eventually start at the same time in all occasions. The never-claims expressions provide the contradictory examples.

The detection of non-progress cycles when all the user interactions are allowed yields a general status of the presentation model. In reality, it is not possible to perform all the interactions at all possible occasions. During the initial modeling phases of our model, SPIN verifier detected a case which naturally is less likely to occur. In this case, the user starts the presentation and then clicks the BACKWARD button just before the presentation proceeds. This leads to an unexpected state where the presentation enters an infinite loop.

After the user starts a presentation and just before the presentation proceeds if the user attempts to backward the presentation, the presentation then enters an unexpected state and stays in this state forever.

6.2 Limitations

Multimedia presentations which provide interactions that change the course of the presentation like skip and backward restricts using PROMELA structures like message channels. The communication among processes like actors and streams are first modeled using channels. If processes are blocked and an interaction (interrupt) requires these process to abort, significant coding is required to cope with the blocked processes.

The PROMELA language does not provide time in the modeling. Thus it is not possible to incorporate time directly in the model. RT-SPIN enables the declaration of time constraints and checks acceptance cycles, non-progress cycles and some liveness properties. The first problem is some guards may be skipped due to lazy behavior of RT-SPIN. In our case, most of the time constraints are equality constraints. Also the interactions like pause, resume, skip, and backward requires the guard condition to be updated after these interactions even when waiting for the guard condition to be satisfied.

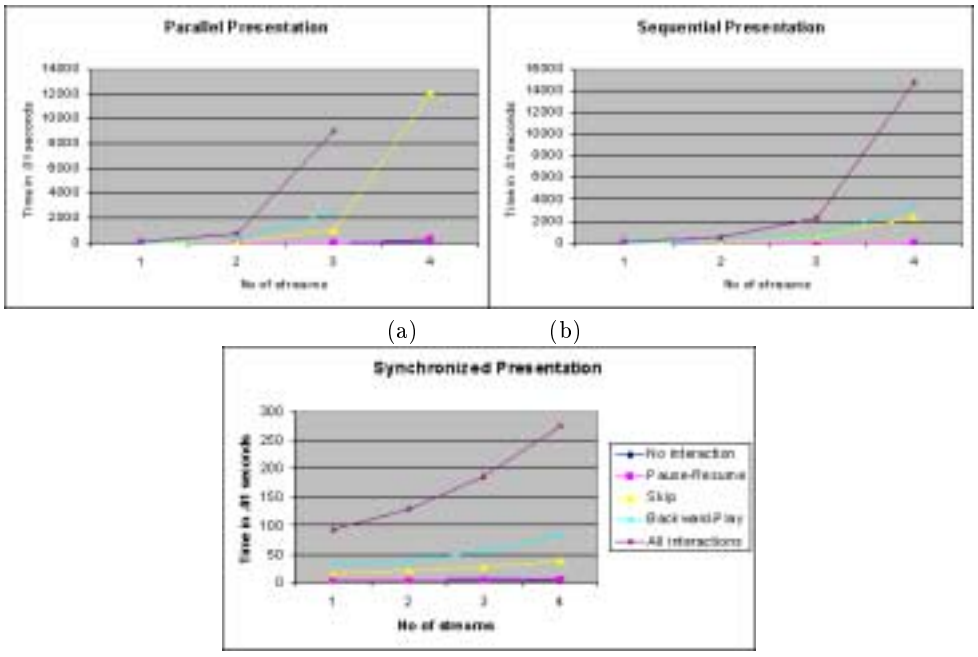


Fig. 6. Performance of user interactions on (a) parallel (b) sequential (c)synchronized presentations.

6.3 Experiences

We had problems in the evaluation and generation of never claims. If the verification reports unreached states before initial execution, a skip statement is included to the beginning of the never claim expression.

When there are still processes enabled, the spin verifier may yield acceptance cycles. If those processes were allowed to proceed, those cycles would be removed. Progress labels are inserted to break these cycles. The never claims are added with $np\_$ to check non progress cycles. We included the weak fairness constraint wherever necessary. We also had a case where never claim is in a cycle after all processes end.

By using LTL to never claim converter, it is not possible to check the ordering of states. Because the same never-claim expression is generated for all $!(y\ \mathbf{U}\ z)$, $!(x\ \mathbf{U}\ y\ \mathbf{U}\ z)$, and $!(w\ \mathbf{U}\ x\ \mathbf{U}\ y\ \mathbf{U}\ z)$.

7 Conclusion

The synchronization model will be incorporated into the NetMedia [12] system, a middleware design strategy for streaming multimedia presentations in distributed environments. NetMedia supports user interactions that change the

course of the presentation like backward and skip. It is necessary whether the system will present a consistent presentation after the user interactions. In this paper, we showed a way of verifying multimedia presentations that also include backward and skip. Firstly, the synchronization model is developed to respond these functionalities . Then the user interactions are allowed and the specification is verified. SPIN's tracing of all possible states provides a way of modeling of delay for multimedia presentations.

References

1. J. Allen. Maintaining Knowledge about Temporal Intervals. *Communications of ACM*, 26(11):823–843, November 1983.
2. R. S. Aygun and A. Zhang. Middle-tier for multimedia synchronization. In *2001 ACM Multimedia Conference*, pages 471,474, Ottawa, Canada, October 2001.
3. CMIS. http://www.cis.ksu.edu/ robby/classes/spring1999/842/index.html.
4. M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of 21st International Conference on Software Engineering*, May 1999.
5. S. Hibino and E. A. Rundensteiner. User interface evaluation of a direct manipulation temporal visual query language. In *ACM Multimedia'97 Conference Proceedings*, pages 99–107, Seattle, USA, November 1997.
6. G. J. Holzmann. The model checker spin. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.
7. I. Mirbel, B. Pernici, T. Sellis, S. Tserkezoglou, and M. Vazirgiannis. Checking temporal integrity of interactive multimedia documents. *VLDB Journal*, 9(2):111–130, 2000.
8. D. O. Paun and M. Chechik. Events in linear-time properties. In *Proceedings of 4th International symposium on Requirements Engineering*, June 1999.
9. B. Prabhakaran and S. Raghavan. Synchronization Models for Multimedia Presentation with User Participation. *Multimedia Systems*, 2(2), 1994.
10. R. S. Aygun and A. Zhang. Interactive multimedia presentation management in distributed multimedia systems. In *Proc. of Int.Conf. on Information Technology: Coding and Computing*, pages 275–279, Las Vegas, Nevada, April 2001.
11. SMIL. http://www.w3.org/AudioVideo.
12. A. Zhang, Y. Song, and M. Mielke. *NetMedia*: A Middleware Design Strategy for Streaming Multimedia Presentations in Distributed Environments. *IEEE Multimedia*. to appear.