# 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 tools 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 *irregular* user interactions which change the course of the presentation like backward and skip are not considered in the specification. Using PROMELA/ SPIN, it is possible to verify the temporal relationships between streams using our model allowing irregular user interactions. Since the model considers the delay of data, the author is assured that the requirements are really satisfied.

## 1 Introduction

There have been models proposed for the management of multimedia presentations. The synchronization specification languages like SMIL [10] have been introduced to properly specify the synchronization requirements. Multimedia query languages have been developed to check the relationships defined in the specification [5]. These tools check the correctness of the specification. However, the synchronization tools do not satisfy all the requirements in the specification or put further limitations. Moreover, the specification does not include user interactions. The previous query-based verification techniques cannot verify whether the system remains in a consistent state after a 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 is hard to model. The Petri-Nets are also used to verify the specification of multimedia presentations [9]. But Petri-Net modeling requires complex Petri-Net modeling for each interaction possible. Authors usually do not have much information about Petri-Nets.

⋆ This research is supported by NSF grant IIS-9733730 and NSF grant IIS-9905603.

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 report on the complexity introduced by user interactions. The experiments are conducted for parallel, sequential, and synchronized presentations.

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

## 2 Modeling of a Multimedia Presentation

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) paradigm. Our synchronization model has receivers, controllers and actors to handle events, condition expression and actor expression, respectively. Timelines are kept for receivers, controllers, actors and actions to keep track when events are signaled, when conditions are satisfied, and when actions start and end. This synchronization model is favored over the others since it allows interactions that change the course of the presentation.

### 2.1 Presentation

The presentation can be in *idle, initial, play, forward, backward, paused,* and *end* states (Figure 1 (a)). The presentation is initially in the *idle* state. The user interface is based on the model presented at [3]. 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.

### 2.2 Containers and Streams

A container or a stream may enter 4 states. A container 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 has some streams that are being played. When all the streams it contains reach their end or when the

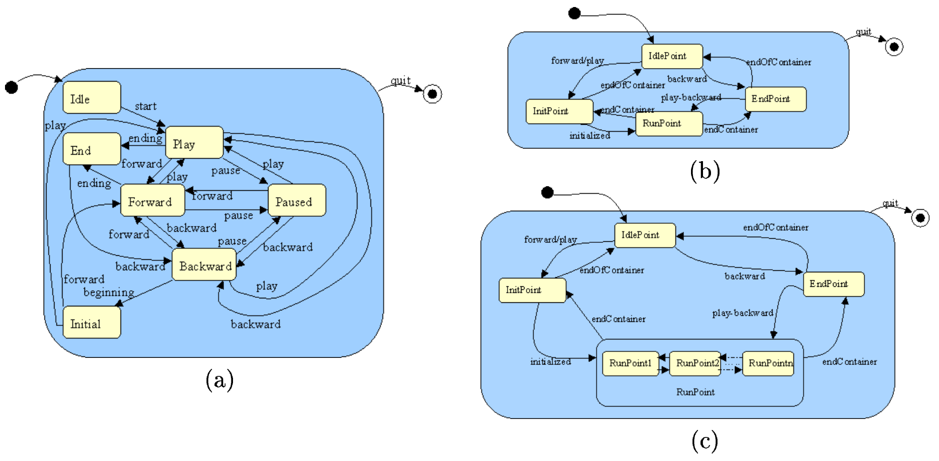

**Fig. 1.** (a) Presentation states (b) container states (c) stream states

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 1 (b)). 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 *RunPoint* state (Figure 1 (c)). The following is a portion of a PROMELA code for playing a stream:

```
proctype playStream (byte stream) {
#if (FC==3 || FC==4 || FC==5 || FC==6)
progressIdleStreams:
#endif
do
#if FC!=4
   :: atomic{ (eventHandled && getState() == RUN) &&
        (getStream(stream) == INIT_POINT) ->
        setStream(stream, RUN_POINT);
        if
        :: (stream==A1)->timeIndex=1;
        :: else -> skip;
        fi; }
   :: atomic{ (eventHandled && getState() == RUN) &&
        (getStream(stream) == RUN_POINT) ->
        setStream(stream, END_POINT);}
   :: atomic{ (eventHandled && getState() == RUN) &&
        (getStream(stream) == END_POINT) ->
        to_end: setStream(IDLE_POINT);
                signalEvent(stream,END_POINT) }
#endif
:
   :: atomic{ (eventHandled && getState() == QUIT) ->
46          to_playStream_quit: goto playStream_quit;}
47     :: else -> skip;
48     od;
49     playStream_quit: skip;}
```

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 17 correspond to forward 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 checking and updating the stream state have to be performed in a single step since the stream state may also be updated by the system after an user interaction.

### 2.3 Receivers, Controllers and Actors

A receiver is set when it receives its event. 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 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 or "something good will eventually happen". 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 should have at least the following fairness constraint: "Eventually the user clicks START button and no user interaction is allowed after that ($\Diamond$ $(userStart \rightarrow \Box noInteraction)$)". If a property is stated as undesirable, the

| | **Properties** | **LTL Formulas** |
|---|---|---|
| 1 | Clicking button for START enables buttons for PAUSE, FORWARD, and BACKWARD, and it changes the simulations state to RUN. | $\Box\ (actionStartClicked \rightarrow\ \Diamond\ actionToRun)$ |
| 2 | Clicking button for PAUSE enables buttons for PLAY, FORWARD, and BACKWARD and it changes the presentation's state to PAUSED. | $\Box\ (actionPauseClicked \rightarrow\ \Diamond\ actionToPaused)$ |
| 3 | The presentation will eventually end. | $\Box\ (stateRun \rightarrow !\ \Diamond\ stateEnd)$ |
| 4 | A stream may start if it is active. | $\Diamond\ (streamRunPoint\ \mathbf{U}\ streamInitPoint)$ |
| 5 | A stream may terminate if it is idle. | $\Diamond\ (streamIdlePoint\ \mathbf{U}\ streamEndPoint)$ |
| 6 | Stream A is before stream B. | $(Q\ \mathbf{U}\ ((R\ \wedge\ M)\ \mathbf{U}\ K))$ |
| 7 | Stream A starts with stream B. | $\Diamond(P\ \wedge\ K)$ |
| 8 | Stream A ends with stream B. | $\Diamond(Q\ \wedge\ L)$ |
| 9 | Stream A is equal to stream B. | $\Diamond(P\ \wedge\ K\ \wedge\ \Diamond\ (Q\ \wedge\ L))$ |
| 10 | Stream B is not during stream A. | $!(\Diamond(P\ \wedge\ \Diamond K)\ \vee\ \Diamond\ (Q\ \wedge\ \Diamond L))$ |
| 11 | Stream B does not overlap stream A. | $!(\Diamond(Q\ \wedge\ \Diamond K)\ \vee\ \Diamond\ (L\ \wedge\ \Diamond Q))$ |
| 12 | Stream A is played. | $\Diamond(P\ \wedge\ \Diamond\ Q)$ |
| 13 | (a) The *state* is reachable in forward presentation. (b) It is possible to reach the *state* in the backward presentation. | (a) $!\Diamond state$ (b) $\Box ! state$ |
| 14 | (a) The *state* is reachable in forward presentation. (b) It is possible to reach the *state* after user interactions. | (a) $!\Diamond state$ (b) $\Box ! state$ |

**Table 1.** Properties and LTL formulas

system should not allow it. The properties and their corresponding LTL formulas are given in Table 1. Properties 1 and 2 are sample properties about state transitions that are allowed by buttons. For LTL formulas 1 and 2, *actionButtonClicked* corresponds to successful clicking *Button* when the button is enabled. *actionToState* corresponds to state transition to *State* after the *action*. Some of the specification patterns are presented in [4, 8]. These specification patterns can be used in the verification. The number of properties about the user interface and state transitions can be increased. Property 3 corresponds to a liveness property that should be checked whether the presentation reaches to its end once it starts. Properties 4 and 5 are undesirable properties that allow streams to restart and terminate when they are in *RunPoint* (active) and *InitPoint* states, respectively.

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. 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.

Properties 6 to 11 are related with verification of Allen's [1] temporal relationships. Properties 10 and 11 are undesirable. For LTL formulas 6 to 11, $P = streamA\_InitState$, $Q = streamA\_EndState$, $R = streamA\_IdleState$, $K = streamB\_InitState$, $L = streamB\_EndState$, and $M = streamB\_IdleState$.

Property 12 checks whether stream A is eventually played. 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. So, Property 13 is stated as two fold. Part (a) is an undesirable property. If the part (a) is wrong, then Part (b) 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 Property 13 to Property 14.

## 4 Experiments

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. 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. A tool is developed for automatic generation of PROMELA code for this kind of presentations. For each interaction, there is a fairness constraint and these are hard-coded in the model (e.g. `FC==3` in line 2 of PROMELA code in Section 2.2).

We conducted tests for each type of interaction using only buttons. The complexity of verification in terms of states and depth are given for no interaction and all interactions in Table 3. The elapsed time of verification of properties is given in Figure 3.

## 5 Evaluation

The previous work on checking the integrity of multimedia presentations deal with presentations that are presented in nominal conditions (i.e., no delay). 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 detection of non-progress cycles when all the user interactions are allowed yields

| Type | No of streams | Depth | States | Transitions | Memory |
|---|---|---|---|---|---|
| single | 1 | 67 | 177 | 306 | 1.5 |
| seq | 2 | 99 | 432 | 865 | 1.5 |
| seq | 3 | 143 | 1021 | 2321 | 1.6 |
| seq | 4 | 209 | 2347 | 5868 | 2.0 |
| par | 2 | 101 | 488 | 1021 | 1.5 |
| par | 3 | 139 | 1699 | 4642 | 1.8 |
| sync | 2 | 73 | 185 | 334 | 1.5 |
| sync | 3 | 78 | 201 | 398 | 1.5 |
| sync | 4 | 83 | 233 | 542 | 1.5 |

**Table 2.** No interaction.

| Depth | States | Transitions | Memory |
|---|---|---|---|
| 745 | 24586 | 46364 | 4.8 |
| 1954 | 103197 | 200756 | 18.5 |
| 5717 | 424039 | 846276 | 85 |
| 18676 | 2.21 K | 4.50 K | 500 |
| 1509 | 184092 | 351747 | 31 |
| 3571 | 1.75 K | 3.42 K | 318 |
| 823 | 30299 | 57461 | 6.1 |
| 869 | 38179 | 74420 | 8.3 |
| 871 | 53939 | 109319 | 12 |

**Table 3.** All interactions allowed

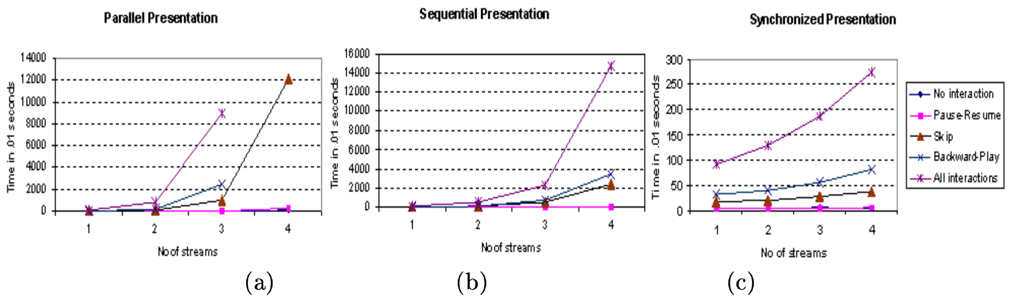

**Fig. 2.** Elapsed time for verification of properties on (a) parallel (b) sequential (c)synchronized presentations.

a general status of the presentation model. 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.

Multimedia presentations which provide interactions that change the course of the presentation like skip and backward restricts using PROMELA structures like message channels. If processes are blocked and an interaction (interrupt) requires these processes 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 [11] 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. 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.

## 6 Conclusion

In this paper, we presented how interactive multimedia presentations can be modeled using PROMELA and verified by SPIN. A subset of these properties is given in Section 3. Since the PROMELA code can be generated automatically, it allows automatic verification of the properties. However, the time complexity of parallel presentations is exponential. This makes the verification difficult when the number of streams increases and requires further optimization in the model. SPIN's tracing of all possible states provides a way of modeling of delay for multimedia presentations. The synchronization model will be incorporated into the NetMedia [12] system, a middleware design strategy for streaming multimedia presentations in distributed environments. It is necessary to assure that the system will present a consistent presentation after the user interactions.

## 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. SMIL. http://www.w3.org/AudioVideo.
11. S. Tripakis and C. Courcoubetis. Extending promela and spin for real time. In *Proceedings of TACAS, LNCS 1055*, 1996.
12. A. Zhang, Y. Song, and M. Mielke. *NetMedia*: Streaming Multimedia Presentations in Distributed Environments. *IEEE Multimedia*, 9(1):56–73, 2002.