PressBase: A Presentation Synchronization Database for Distributed Multimedia Systems

Ramazan Savaş Aygün *Member, IEEE,* and Aditya S. Patil

Abstract

Multimedia presentations are the basic objects of multimedia databases. Since a multimedia presentation is not an instant display of a query result, the control knowledge (or synchronization requirements) has to be incorporated into the database and necessary precautions have to be taken for a lengthy presentation. Active databases provide a mechanism for incorporation of control knowledge by using Event-Condition-Action (ECA) rules. In this paper, we describe how multimedia synchronization can be handled within a database using ECA rules. We present a prototype presentation synchronization database, named as *PressBase*, for distributed multimedia systems. We have adopted one of the synchronization models, *SynchRuler* [5], and then incorporated into a relational database system.

Index Terms

Multimedia synchronization, database triggers

I. Introduction

There has been tremendous effort on the management of multimedia information systems and databases for more than a decade. Most of the focus on multimedia database research has been on extracting, indexing, and querying high-dimensional features. The major difference between multimedia databases and image, audio, video, and text databases is the organization of different types of media in a single object named as a *multimedia presentation*. The synchronization requirements of a presentation are usually ignored or handled separately from the database. It has been the responsibility of the display tool to ensure

R. Aygün is with Computer Science Department, Technology Hall N360, University of Alabama in Huntsville, Hunstville, AL 35899. Phone: 256-824-6455, Fax: 256-824-6239, Email: raygun@cs.uah.edu

A. Patil was with University of Alabama in Huntsville. He is currently at ADS Environmental Services, 4940 Research Drive, Huntsville, AL 35805. Phone: 256-468-6815, Fax: N/A, Email: aditya.patil@adsenv.com

application specific quality of service. The new upcoming standards like MPEG-21 [13] must guarantee the delivery of corresponding content of a digital item.

A. PressBase

Active databases store the data as well as the control knowledge by using the event-condition-action (ECA) paradigm. For flexible presentations where delays or loss of data may occur over networks, multimedia presentation specification based on ECA rules provide consistent presentations. We call databases that handle the synchronization requirements of multimedia presentations as *active multimedia databases (AMD)*. Our focus in this paper is on how multimedia synchronization requirements can be expressed and applied in a relational database management system (DBMS). We have adopted *SynchRuler* synchronization model [5]. SynchRuler uses ECA rules for multimedia synchronization and is built on a Receiver-Controller-Actor (RCA) scheme. Receivers are objects to receive events, controllers are objects to check conditions, and actors are objects to execute actions.

In this paper, we provide a prototype presentation synchronization database for distributed multimedia systems named as *PressBase*. *PressBase* can be incorporated into systems that would like to enforce synchronization within a database and separate the synchronization module from the player module. To the best of our knowledge, *PressBase* is the first system that manages the synchronization within a database. This paper implicitly presents a mapping from SMIL expressions to SQL triggers. Our contributions can be listed as follows:

- the separation of the synchronization module from the player module,
- the conceptual modeling of synchronization requirements using enhanced entity-relationship (ER) model,
- the mapping of synchronization requirements to relational data model
- synchronization using database triggers, and
- the management of user interactions with DBMS support.

B. Related Work

The active database technology supports the definition, management, and execution of Event-Condition-Action (ECA) rules [10]. There have been many prototypes and projects that have appeared based on this technology like Starburst [18], POSTGRES [16], and Sentinel [8]. The architecture of active database systems is explained in [7]. Allen [1] proposed 13 basic temporal relationships for time intervals. Hamakawa et al. [11] has an object composition and a playback model where the constraints can be

defined only as pair-wise. FLIPS [14] is an event-based model that has barriers and enablers to satisfy the synchronization requirements at the beginning and the end of the streams. PREMO [12] presents an event-based model that also manages time. A hierarchical synchronization model that has events and constraints is given in [9]. SMIL [15] is a mark-up language for publishing synchronized multimedia presentations via the Internet. NSync [6] is a toolkit that manages synchronous and asynchronous interactions by synchronization expressions having syntax *When* $\{expression\}$ $\{action\}$. Multimedia synchronization using ECA rules is covered in our work [2], [3], [4].

This paper is organized as follows: The next section describes the *SynchRuler* and the architecture of *PressBase*. The database design of *PressBase* is given in Section III. Section IV explains the synchronization management. Section V discusses the performance analysis and experiments. The last section concludes our paper.

II. From SynchRuler to PressBase

A. The ECA Rule Model and Management in "SynchRuler"

In *SynchRuler*, ECA rules are called as *synchronization rules*. 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*. 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.

1) Events, Conditions, and Actions for a Presentation: The event expression enables the composition of events. Composite events can be created by boolean operators $\&\&$ and $||$. The hierarchy of multimedia events is given in [3]. Allen [1] specifies 13 temporal relationships. Relationships *meets, starts* and *equals* require the *InitPoint* event for a stream. Relationships *finishes* and *equals* require the *EndPoint* event for a stream. Relationships *overlaps* and *during* require *realization* event to start (end) another stream in the middle of a stream. The relationships *before* and *after* require temporal events since the gap between two streams can only be determined by time.

The condition expression determines the set of conditions to be validated when an event expression is satisfied. A condition indicates the status of the presentation and its media objects.

The action expression is the list of the actions to be executed when a condition is satisfied. An action indicates what to execute when conditions are satisfied. *Start*ing and *end*ing a stream are sample actions. If an action has started and not finished yet, that action is considered as an alive action.

```
<seq id=''MAIN''>
    <par id=''PAR''>
        <video id="V1" src="perspective1.mpg"/ dur=''5min''>
        <video id="V2" src="perspective2.mpg" dur=''5min''/>
    </par>
    <video id="V3" src="intersection.mpg" begin=''1min'' dur="10min"/>
</seq>
```

Fig. 1. The SMIL expression.

(F1)	**on** `user(START)`	**if**	`direction=FORWARD`	**do**	`start(MAIN)`
(F2)	**on** `MAIN(INITPOINT)`	**if**	`direction=FORWARD`	**do**	`start(PAR)`
(F3)	**on** `PAR(INITPOINT)`	**if**	`direction=FORWARD`	**do**	`start(V1) start(V2)`
(F4)	**on** `(V1(ENDPOINT) && V2(ENDPOINT))`	**if**	`direction=FORWARD`	**do**	`end(PAR)`
(F5)	**on** `PAR(ENDPOINT)`	**if**	`direction=FORWARD`	**do**	`start(V3,1min)`
(F6)	**on** `V3(ENDPOINT)`	**if**	`direction=FORWARD`	**do**	`end(MAIN)`

Fig. 2. Synchronization rules.

2) Multimedia Presentations: The basic component of a multimedia presentation is a stream. In *SynchRuler*, a multimedia presentation may have a *container* that allows grouping of containers and streams. Let us consider a simplified training example that is composed of two parts. In the 1^{st} part, the trainer shows two videos (V1 and V2) of a traffic intersection scene from two perspectives, simultaneously. In the 2^{nd} part (1 minute after the 1^{st} part), the trainer shows the continuation of a scene from one of the perspectives (V3). If the presentation is grouped according to the SMIL expression in Fig. 1, there are two containers in the presentation: the parallel presentation (PAR) of V1 and V2, and the sequential presentation (MAIN) of PAR and V3. The SynchRuler yields the synchronization rules in Fig. 2.

3) Rule Processing: The rule manager 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

R1	USER(Start)
R2	MAIN(InitPoint)
R3	PAR(InitPoint)
R4	V1(EndPoint)
R5	V2(EndPoint)
R6	PAR(EndPoint)
R7	V3(EndPoint)

TABLE I

RECEIVERS.

C1	R1 && F
C2	R2 && F
C3	R3 && F
C4	R4 && R5 && F
C5	R6 && F
C6	R7 && F

TABLE II

CONTROLLERS.

A1	start(MAIN)
A2	start(PAR)
A3	start(V1)
A4	start(V2)
A5	end(PAR)
A6	start(V3,1min)
A7	end(MAIN)

TABLE III

ACTORS.

are satisfied. When receiver R receives event e, it sends information of the receipt of e to all its controllers. Controller C has two components to verify, composite events ee and conditions ce about the presentation. When the controller C is notified, it checks whether the event composition condition ee and conditions ce are satisfied. If they are both satisfied, the controller notifies its actors. Once actor A is informed, it waits for time t (≥ 0) and then starts its action.

According to the synchronization rules given in Fig. 2, there are 7 receivers for events specified in the event expressions (Table I), 6 controllers for synchronization rules (Table II), and 7 actors for actions (Table III). Each stream or a container is at least expected to signal a starting event (INITPOINT) and ending event (ENDPOINT). When the user clicks start button, receiver $R1$ receives this event. It informs controller $C1$. If the condition of $C1$ (direction=FORWARD) is satisfied, it notifies the actor $A1$. Once $A1$ is notified, it starts the container MAIN. When receiver $R4$ receives the EndPoint event for $V1$, it notifies the controller $C4$. $C4$ will ask $R4$ and $R5$ whether they received events. If they both received events, it checks the direction condition and notifies actor $A5$.

B. PressBase Architecture

The whole system is comprised of two main components: the front-end and back-end. The front-end manages the interactions with the user whereas the back-end manages the synchronization in the database. The front-end is responsible for carrying out the presentation to the user. The user may interact with the system with VCR-based user interactions like STOP, PLAY, FORWARD, and SKIP functions. The front-end generates the events and informs the database system. It sets and maintains the timer for the actions that are supposed to begin execution after a specified time interval. After an event, the front-end also polls the database to check the status of actions that are ready to execute and then executes them.

In *PressBase*, the multimedia presentations with its components are treated as database objects even during streaming and presenting data. The states of multimedia objects are updated during presentation. Although an event may be discarded, its status is maintained by its receiver.

III. Creation of the Synchronization Database

The representation of a synchronization model in a database system is challenging due to its complexity. So far the synchronization of multimedia presentations was only achieved (with some limitations) due to the flexibility and the power of multi-threaded procedural languages. We were challenged with the limitations of database systems.

A. Conceptual Model

A single entity type, *ContainerStream*, is created for containers and streams. ContainerStream is comprised of *containerStreamName*, *containerStatus*, *source file* and *isStream* attributes. ContainerStream-Name is the unique identifier for a container or a stream. ContainerStatus indicates the status of the container: 0 (container has not started), 1 (container started and has signaled event INITPOINT), and 2 (container has ended and signaled event ENDPOINT). Source file is the path of the file in case of streams and null for containers. IsStream indicates whether containerStreamName is a stream or a container.

An event entity has an *event type* (e.g., InitPoint, EndPoint) and an *event data*. The source of an event is the user or a container/stream. The ER diagram for container/stream and event is depicted in Fig. 3. For each event there is a receiver. The receiver keeps the status of its event whether it is signaled or not. A receiver is comprised of *receiverId, receiverTime,* and *status*. *ReceiverId* is the unique identifier for the receiver (e.g. R1, R2). *ReceiverTime* is the time from the start of the presentation till the receipt of the associated event. The ER diagram for event and receiver is depicted in Fig. 4.

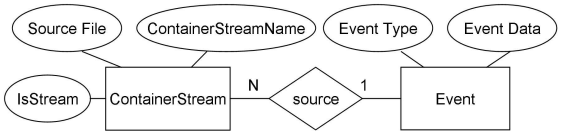

Fig. 3. The ER diagram for container, stream, and event.

One major difference between *SynchRuler* and *PressBase* is the interaction between the receivers and controllers. In *PressBase*, since the receivers do not notify the controllers, the controllers do not check whether its conditions are satisfied or not. When a receiver receives its event, it updates its status in the

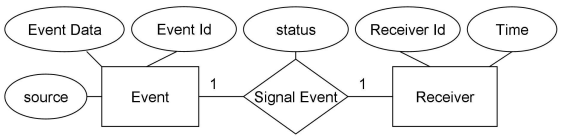

Fig. 4. The ER diagram for event and receiver.

composite conditions of controllers (Fig. 5). When the condition of a controller is satisfied, it notifies its controller.

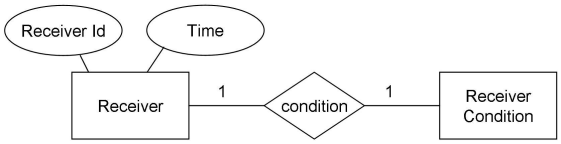

Fig. 5. The ER diagram for receiver as a condition.

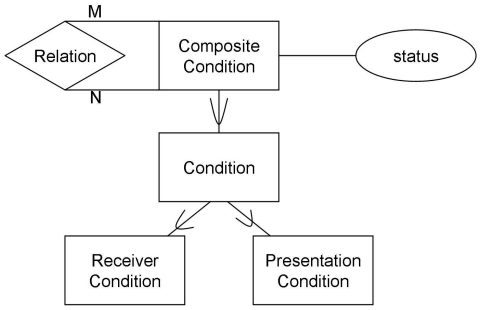

Fig. 6. The Enhanced ER diagram for composite condition.

Each receiver condition and presentation condition is considered as a subclass of the composite condition. This alleviates the utilization of receiver status and presentation conditions in the composite conditions. The entity composite condition is comprised of the *conditionName, subConditionName1, subConditionName2, compositionType,* and *status* (Fig. 6). The *conditionName* is the unique identifier of the composite condition. The *subConditionName1* and *subConditionName2* are the two components of the composite condition. The status indicates the evaluation result of the (composite) condition. The relation indicates the relation between the two conditions (AND, OR).

When a composite condition is satisfied, it notifies the controller having the controllerId as its identifier. The controller entity has *controllerId* and *controllerTime* attributes (Fig. 7). *ControllerId* is the unique

identifier of the controller. *ControllerTime* is the time when the conditions of the controller are satisfied and the *statusId* indicates whether the conditions related to the controller are satisfied or not. The *statusId* actually indicates whether the controller's composite conditions have evaluated to true or false at an instant.

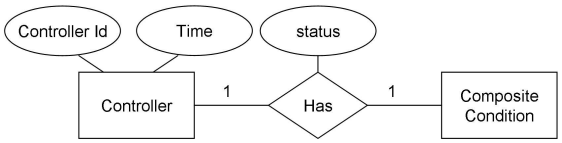

Fig. 7. The ER diagram for receiver and controller.

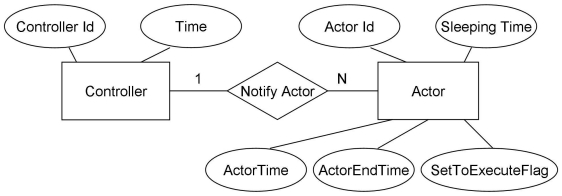

Fig. 8. The ER diagram for controller and actor.

When the status of a controller becomes true, it notifies its actors. Each controller may have more than one actor but each actor is only notified by a single unique controller (Fig. 8). An actor is comprised of *actorId, sleepingTime, actorTime,* and *setToExecuteFlag*. *ActorId* is the unique identifier of the actor. The *sleepingTime* indicates the time for which the actor waits before executing the action. The *actorTime* indicates the time from the start of the presentation until the actor is activated. The *setToExecuteFlag* indicates the status of the actor and it may be 0 (actor not started), 1 (actor has begun the timer; i.e. sleeping time started), 2 (actor is sleeping for sleeping time), and 3 (actor is done; i.e. triggers the corresponding action).

When the sleeping time of an actor expires, the actor executes its action. The action entity is comprised of *actionId, actionData, actionType, actionTime,* and *setToExecute* attributes. The action has also a destination (container/stream) where the action has to be applied. The relationship between an action and container/stream is displayed in Fig. 9. *ActionId* is the unique identifier of the action to be executed by the corresponding actor. The *actionType* determines the type of the action (e.g., start, end). The *actionData* keeps the parameter of the action. The *actionTime* is the time when the action executes. The *setToExecute* indicates the the status of the action and it may be 0 (action not started), 1 (action ready

to start; i.e. sleeping time expired), 2 (action is executing; i.e. alive), and 3 (action has ended). The ER diagram for actor and action entities is depicted in Fig. 10.

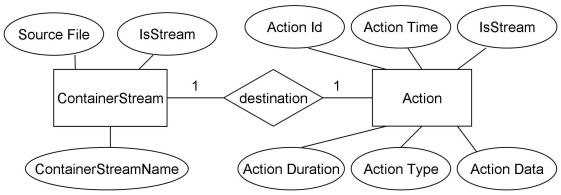

Fig. 9. The ER diagram for container, stream, and action.

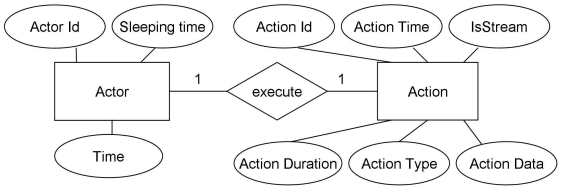

Fig. 10. The ER diagram for actor and action.

We also modeled user interactions through UserSelectionAndPresentationDirection entity. It is comprised of *userSelection*, *userSelectionStatus*, and *userData*. UserSelection is the interaction name (SKIP, FORWARD, BACKWARD, PAUSE). UserSelectionStatus indicates the interaction mode: 1 (interaction) and 0 (no interaction). UserData indicates the data for interaction types: a) for SKIP, data indicates the time to which the presentation to be skipped, b) for FORWARD/BACKWARD, data indicates the speed of the presentation, and c) for PAUSE, data does not apply.

B. Mapping to Relational Schema

The receiver and event are combined under a single relation since the relation is 1:1 and requires total participation for both sides. The controller, composite condition, actor, action, userSelectionAndPresentationDirection, and containerStream entities are mapped to Controller, CompositeCondition, Actor, Action, UserSelectandPresDirection, and ContainerStream relations, respectively. The relational schema is as follows:

RECEIVER`(receiverId varchar(50),`

`receiverTime int, controllerId varchar(50),`

```
source varchar(50), eventType varchar(50),
eventData int, eventStatus int)
COMPOSITECONDITION(compositeConditionName
varchar(50), subConditionName1 varchar(50),
subCondition1Status int, relation int,
subConditionName2 varchar(50),
subCondition2Status int, status int)
CONTROLLER(controllerId varchar(50),
controllerTime int, statusId int)
ACTOR(actorId varchar(50), sleepingTime int,
actorTime int, setToExecuteFlag int,
controllerId varchar(50), actorEndTime int)
ACTION(actionId varchar(50),
actorId varchar(50), streamName varchar(50),
actionData varchar(50), actionTime int,
setToExecute int, isStream int,
actionDuration int, actionType varchar(50))
USERSELECTANDPRESDIRECTION(userSelection
varchar(50), userSelectionStatus int,
userData int)
CONTAINERSTREAM(containerStatus int,
ContainerStreamName varchar(50),
sourceFile varchar(50), isStream int)
```

IV. Managing Synchronization

A. Synchronization among Streams and Containers

The synchronization is managed by the triggers in the DBMS. We have used Microsoft SQL Server 2000 as our database system. Triggers are implemented using Transact Structured Query Language (T-SQL) [17]. When a receiver receives its event, it has to update its values in the CompositeCondition relation. Whenever an event is raised in the system, the eventStatus attribute of the particular Receiver that receives the event is set to 1 (TRUE). This triggers off the trigUpdateReceiver trigger that scans the CompositeCondition relation. Let R be the receiver that received its event. R may exist as a

subCondition1 or a subCondition2. Therefore, the status in both columns needs to be updated. Since R exists as subCondition, the change may affect its composite conditions. The composite conditions need to be updated based on its relation (AND, OR). As a result, there are 5 components of the trigger for each update. Since composite conditions may also participate as a subCondition, both columns for subCondition1 and subCondition2 that have the updated composite condition must be updated. The updates will continue until no change in the status of conditions. The trigger code for the receiver condition update in CompositeCondition relation is as follows:

```
CREATE TRIGGER trigUpdateCompositeCondition
 ON RECEIVER FOR UPDATE AS
  Update CompositeCondition
   set subCondition1Status = 1
   where subConditionName1 IN
   (SELECT ReceiverId from Inserted
   where eventStatus = 1)
  Update CompositeCondition
   set subCondition2Status = 1
   where subConditionName2 IN
   (SELECT ReceiverId from Inserted
   where eventStatus = 1)
 Update CompositeCondition
   set subCondition1Status = 1
   where subCondition1Status = 0
   and subConditionName1 IN
   (SELECT CompositeConditionName
   from CompositeCondition where status=1)
 Update CompositeCondition
   set subCondition2Status = 1
   where subCondition2Status = 0 and
   subConditionName2 IN
   (SELECT CompositeConditionName
   from CompositeCondition where status=1)
```

```
Update CompositeCondition set status = 1
  where ((subCondition1Status =1 and
  subCondition2Status =1 and relation = 0)
  or ((subCondition1Status =1 or
  subCondition2Status =1) and relation = 1))
```

When the composite condition becomes true, its controller status must be updated. The update of the CompositeCondition relation triggers off the trigUpdateController trigger. It finds all the composite conditions in the CompositeCondition relation that have been evaluated to TRUE and sets the corresponding entries in the Controller relation to TRUE that were previously FALSE. The trigger for the composite condition is as follows:

```
CREATE TRIGGER trigUpdateController
ON CompositeCondition FOR UPDATE AS
 update Controller set statusId = 1
  where statusId = 0 and ControllerId IN
  (select CompositeConditionName from Inserted
  where status = 1)
```

When the status of a controller becomes true, it has to notify its actors. The update of the controller triggers off the trigUpdateActor that sets the setToExecuteFlag of the tuples of the Actor relation to 1 (actor has begun) if previously set to 0, corresponding to the ControllerId of the updated tuple(s). The trigger for the controller is as follows:

```
CREATE TRIGGER trigUpdateActor
ON Controller FOR UPDATE AS
 update Actor set setToExecuteFlag = 1
  where setToExecuteFlag = 0 and
  ControllerId IN (select ControllerId from
  Inserted where statusId = 1)
```

When the sleeping time of an actor expires, the front-end sets the status of setToExecuteFlag of actor to 3 (sleeping time has expired). This indicates that the action needs to be triggered. The actor has to update the status of its action so that the front-end can realize the actions to execute. The update of

the Actor relation sets off the trigUpdateAction trigger that sets the setToExecute attribute of tuples of Action relation to 1 if previously set to 0. When the setToExecute flag is set to 1, the stream is ready to execute. The trigger is as follows:

```
CREATE TRIGGER trigUpdateAction
ON Actor FOR UPDATE AS
 update Action set setToExecute = 1
  where setToExecute = 0 and actorId IN
  (select actorId from Inserted
  where setToExecuteFlag = 3)
```

B. User Interactions

Our system is also able to handle VCR-based user interactions like, PLAY, PAUSE, FORWARD, SKIP, and BACKWARD. The user interactions like skip or changing direction (backwarding when playing forward or vice versa) need to be handled carefully. PressBase allows to skip forward and backward. In other words, the user may skip to a point before or after the current point. The skip operation also allows to skip to any point in the presentation whether it may correspond to skipping a portion of a stream or not. For example, if the user is watching 9^{th} minute of the presentation in Fig. 1 (this corresponds to 3^{rd} minute of V_3), the user may skip to 2^{nd} minute of the presentation (this corresponds to 2^{nd} minutes of V_1 and V_2). When skip-forward is performed, some events may be skipped that may cause ignorance of future streams that depend on the receipt of the skipped events. This problem is solved by using the time information associated with receivers, controllers, actors, and actions. The actions that will be active at the skip point are started from their corresponding points. The actors whose *sleeping time*s have not expired are allowed to sleep for the remaining time. To skip to 13^{th} minute of a forward presentation, the following query is generated:

```
update UserSelectAndPresDirection
 set userSelectionStatus = 1, userData = 13
 where userSelection = 'SKIP'
```

The receivers whose events had to be signaled before 13^{th} minute must be set to true. The other receivers will be set to false. The trigger code to maintain Skip for Receiver relation is as follows:

```
CREATE TRIGGER updateReceiver
```

```
ON USERSELECTANDPRESDIRECT FOR UPDATE AS
 DECLARE @skipTime INTEGER
 DECLARE @getInteraction VARCHAR(50)
 select @skipTime = (select userData
  from Inserted where userSelectionStatus = 1
  and (userSelection = 'SKIP' ))
 select @getInteraction =
  (select userSelection from Inserted)
 IF @getInteraction = 'SKIP' and @skipTime>0
  BEGIN
   update Receiver set eventStatus = 1
    where receiverTime ≤ @skipTime
   update Receiver set eventStatus = 0
    where receiverTime > @skipTime
  END
```

The receiver condition in composite condition relation can only be set by trigUpdateCompositeCondition trigger. After skip operation, some of the receivers are reset and this may require the resetting of a composite condition. Therefore, the original trigUpdateCompositeCondition is modified in a way to include the cases where the receiver status becomes false. Due to space limitations, we are not going to repeat the trigUpdateCompositeCondition for resetting (it is almost the same as the initial code where 1s are replaced with 0s). The trigger for the controllers include the updates of the status based on the updates from the composite condition states. The trigger for controllers is the same as the trigger for receivers since the controllers are updated according to their time values.

The triggers for actors and actions are updated based on their time values. However, actors and actions correspond to an interval rather than an instant. After a skip, an actor might need to start sleeping or be in the middle of its sleeping time. An actor just needs to sleep for the remaining time. The similar case also applies for actions. The trigger code is as follows:

```
IF @isSkip = 1 --if operation is skip
 BEGIN
  update Actor set setToExecuteFlag = 3
   where actorEndTime ≤ @skipTime
```

```
  update Actor set setToExecuteFlag = 1
   where actorTime = @skipTime
  update Actor set setToExecuteFlag = 0
   where actorTime > @skipTime
  update Actor set setToExecuteFlag = 2
   where actorTime < @skipTime
    and actorEndTime > @skipTime
  END
```

In a similar fashion, the trigger code for updating action after a skip interaction is as follows:

```
IF @isSkip = 1 --if operation selected is skip
 BEGIN
  update Action set setToExecute = 0
   where actionTime > @skipTime
  update Action set setToExecute = 3
   where (actionTime + actionDuration)
    ≤ @skipTime and isStream=1
  update Action set setToExecute = 1
   where actionTime ≤ @skipTime and isStream=0
  update Action set setToExecute = 1
   where actionTime = @skipTime and isStream=1
  update Action set setToExecute = 2
   where actionTime < @skipTime
   and (actionTime + actionDuration)
    > @skipTime and isStream=1
 END
```

Only streams that must be alive at the skip time are allowed to be alive. Under normal execution, the corresponding container status must be set to 1 (started but not done) when its action is ready to start. The skip operation supports skipping a certain interval of a stream as well as skipping the presentation of several streams. The actionTime is the time when the action should be executed in a nominal presentation (when there is no delay and no interaction). The certain interval of a stream to be skipped is determined

by $skipTime - actionTime$. To minimize the repetition we have not included the cases when there is no skip. The trigger for containerStream is as follows:

```
CREATE TRIGGER trigUpdateContainerStream
ON ACTION FOR UPDATE AS
IF @isSkip=1
BEGIN
 update ContainerStream set containerStatus=1
  where containerStreamName IN
  (select streamName from Inserted
  where setToExecute = 1 or
  setToExecute = 2 ) and isStream=1
 update ContainerStream set containerStatus=2
  where containerStreamName IN
  (select streamName from Inserted
  where setToExecute = 3 and isStream=1)
 update ContainerStream set containerStatus=0
  where containerStreamName IN
  (select streamName from Inserted
  where setToExecute = 0)
 update ContainerStream set containerStatus=1
  where containerStreamName IN
  (select streamName from Inserted
  where setToExecute = 1
  and actionType='START') and isStream=0
 update ContainerStream set containerStatus=2
  where containerStreamName IN
  (select streamName from Inserted
  where setToExecute = 1
  and actionType='END') and isStream=0
END
```

V. Experiments and Performance Analysis

Let $|R|$, $|C|$, and $|A|$ be the number of receivers, the number of controllers, and the number of actors respectively. The number of actors is equal to the number of actions. Each controller has a composite condition expression (*cce*), and a *cce* may be represented as a binary tree of conditions where leaf nodes are conditions (receivers or the direction) and internal nodes are binary operators (like AND, OR). In our experiments, each *cce* has one direction condition and has at least one receiver. This guarantees that each *cce* has at least three nodes. Let L_{C_i} be the number of leaf nodes containing receivers in the *cce* of controller C_i. Let $L_C = \Sigma_{i=1}^{n} L_{C_i}$ denote the total number of leaf nodes containing receivers in *cce*s of all controllers. In SynchRuler, the number of controller notifications is $|R|$. When the controller is notified, it checks the states of the receivers. The number of checkings is L_C. However, the number of rule firings is $|C|$. This corresponds to the fact that $L_C - |C|$ checkings do not cause any rule firing.

The trigger to update the composite condition is fired at least $|R|$ times. It may increase when a receiver participates in more than one controller. The number of triggers is equivalent to the number of internal nodes in the composite condition tree. Since the composite condition expression is a binary tree, the total number of internal nodes is $N_{C_i} = L_{C_i} - 1$ for *cce* of controller C_i. Let $N_C = \Sigma_{i=1}^{n} N_{C_i}$ denote the total number of internal nodes in *cce*s of all controllers. When there is an update on the status of a composite condition, it may also trigger a new recursive trigger. However, this only happens as the number of internal nodes, N_C, increases. If there are $|C|$ controllers, $N_C - |C|$ triggers are unnecessary. Each update on the status of a composite condition entry will cause a trigger to update the status of controllers. This trigger is again called N_C times. The updates on the controllers will cause a trigger to update the actor. Since there are $|C|$ controllers, the number of firings is $|C|$. The updates on the actors will cause $|A|$ firings on the action. Let $T_{pb} = |R| + 2 * N_C + |C| + |A|$ denote the number of triggers for a presentation in PressBase and $T_{sr} = |R| + L_C + |C| + |A|$ denote the number of notifications and checkings in *SynchRuler*.

The reason that PressBase has $2 * N_C$ is the treatment of composite condition expression as a separate table. In SynchRuler, composite condition expression is part of its controller. For the presentation in Fig. 2, $|R| = 7$, $|C| = 6$, $|A| = 7$, $N_C = 7$, and $L_C = 7$. Only the *cce* of controller $C4$ has two internal nodes. This means that there are two unnecessary triggers $(2 * (N_C - |C|))$ for this example and $T_{pb} = 7 + 2 * 7 + 6 + 7 = 34$. Besides the number of triggers, the duration of a presentation is also important. If the duration of a presentation is t seconds, the number of triggers per second is T_{pb}/t. If the duration of a presentation is 1 minute, there is one trigger every two seconds for the presentation.

The front-end is implemented using Microsoft Visual C#. We measured the performance of our system. We have considered four factors: the jitter between two streams, the time to notify the database, the time for the database to respond, and the time for restart after skip. When two streams must start at the same time, the difference in their start time is reported in Fig. 11. The average delay is 31 milliseconds. The average time to start a stream when an event causes rule firing is 219ms since the signaling of the event. The experiments about the time to notify the database is given in Fig. 12. The average time to notify the the database is 52ms. The experiments to measure the performance of stored procedures and triggers are given in Fig. 13. The average time to reflect the stored procedures and triggers are 9ms. We also measured the time to start streams after a skip operation (Fig. 14). The average time to start the presentation after a skip is 587ms. This is reasonable time due to closing active streams and allocating the new streams. We believe that our system performance is a bit degraded since we kept a log to record the timings of events, conditions, and actions. We have performed additional experiments to determine whether there are any relationships with the spikes in the figures and the number of rule firings in the database. As a result of our experiments, we have not found any correlation between them. However, whenever there is a spike in Fig. 12, it causes a spike in Fig. 14. This is due to the fact that informing the database is also a part of the skip operation.

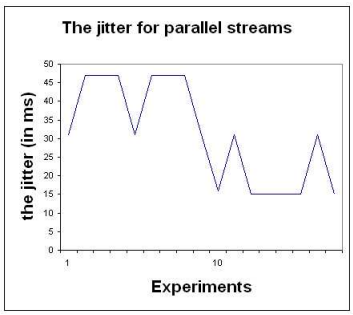

Fig. 11. The jitter on the start of parallel streams.

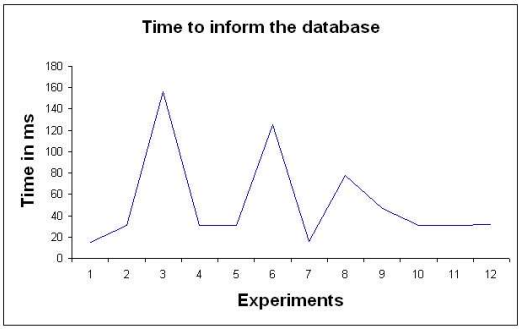

Fig. 12. The experiments on time to inform the database on an event.

VI. Conclusion

In this paper, we presented how the synchronization can be managed within a database. The major breakthrough is the separation of the synchronization module from the player module. We have also given a detailed explanation how user interactions like skip are handled by the database. We have listed our code so that our code can be utilized, tested, and compared with new synchronization models that are

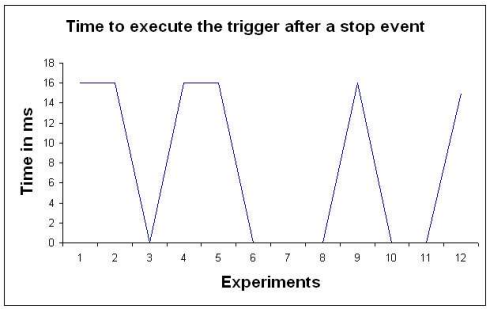

Fig. 13. The experiments on time to execute a trigger after an event.

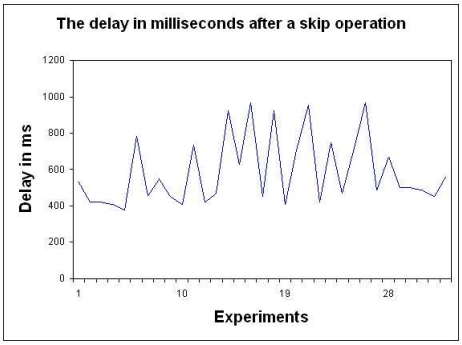

Fig. 14. The experiments on time to start streams after a skip.

managed by a DBMS. We are planning to perform a detailed investigation on the transaction management (especially when a multimedia presentation is accessed by various users), query processing, and rule management in *PressBase*. We lose efficiency due to complex structure of composite conditions. As a future work, we are planing to improve the structure of composite conditions in the relational database. According to our experience, the current results are promising for future multimedia systems that aim to incorporate synchronization in database systems.

REFERENCES

[1] J. Allen. Maintaining Knowledge about Temporal Intervals. *Communications of ACM*, 26(11):823–843, November 1983.

[2] R. 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.

[3] R. Aygun and A. Zhang. Middle-tier for multimedia synchronization. In *2001 ACM Multimedia Conference*, pages 471–474, Ottawa, Canada, October 2001.

[4] R. Aygun and A. Zhang. Management of backward-skip interactions using synchronization rules. In *The 6th World Conference on Integrated Design & Process Technology*, Pasadena, California, June 2002.

[5] R. Aygun and A. Zhang. SynchRuler: A flexible synchronization model using model checking. accepted by *IEEE Transations in Knowledge and Data Engineering*.

[6] B. Bailey, J. Konstan, R. Cooley, and M. Dejong. Nsync - a toolkit for building interactive multimedia presentations. In *Proceedings of ACM Multimedia*, pages 257–266, September 1998.

[7] A. Buchmann. Architecture of Active Database Systems. *Active Rules in Database Systems*, pages 29–48, 1999.

[8] S. Chakravarty, V. Krishnaprasad, Z. Tamizuddin, and R. Badani. Eca rule integration into an oodbms: Architecture and implementation. In *Proc. of the 11th Intl. Conf. on Data Engineering*, 1995.

[9] J. Courtiat, R. Oliverira, and L. Carmo. Toward a New Multimedia Synchronization Mechanism and its formal Specification. In *Proceedings of ACM Multimedia 94*, pages 133–140, San Francisco, California, October 1994.

[10] U. Dayal. Active database management systems. In *Proceedings of the 3rd Intl. Conf. on Data and Knowledge Bases*, 1988.

[11] R. Hamakawa and J. Rekimoto. Object composition and playback models for handling multimedia data. *Multimedia systems*, 2:26–35, 1994.

[12] I. Herman, N. Correira, D. A. Duce, D. J. Duke, G. J. Reynolds, and J. V. Loo. A standard model for multimedia synchronization: Premo synchronization objects. *Multimedia systems*, 6(2):88–101, 1998.

[13] MPEG-21. Iso/iec tr 21000-1:2001(e) part 1: Vision, technologies and strategy.

[14] J. Schnepf, J. Konstan, and D. Du. FLIPS: Flexible Interactice Presentation Synchronization. *IEEE Selected Areas of Communication*, 14(1):114–125, 1996.

[15] SMIL. http://www.w3.org/AudioVideo.

[16] M. Stonebraker and G. Kemnithz. The POSTGRES Next-Generation Database Management System. *Communications of ACM*, 34(10), October 1991.

[17] Transact-SQL. http://www.msdn.microsoft.com/SQL/sqlreldata/TSQL.

[18] J. Widom. *Active Database Systems: Triggers and Rules for Advanced Database Processing*. Morgan-Kaufmann, San Francisco, California, 1996.

Ramazan S. Aygün received the B.S. degree in computer engineering from Bilkent University, Ankara, Turkey in 1996, the M.S. degree from Middle East Technical University, Ankara in 1998, and the Ph.D. degree in computer science and engineering from State University of New York at Buffalo in 2003. He is currently an Assistant Professor in Computer Science Department, University of Alabama in Huntsville. His research interests include multimedia databases, multimedia synchronization, and video processing.

Aditya S. Patil received the B.E. degree in computer engineering from University of Pune, India in 2003 and the M.S. degree in computer science from University of Alabama in Huntsville in 2005. He is currently working as a software engineer at ADS Environmental Services in Huntsville, Alabama. His research interests include multimedia databases, networking and operating systems.