ELSEVIER

Information and Software Technology 39 (1997) 561–578

INFORMATION AND SOFTWARE TECHNOLOGY

Distributed multimedia synchronization specifications using M²EST

Chung-Ming Huang[a,*], Ye-In Chang[b], Chih-Hao Lin[a], Jhy-Shiou Chen[a]

[a]*Laboratory of Multimedia Networking (LMN), Institute of Information Engineering, National Cheng Kung University, Tainan, Taiwan, R.O.C.*
[b]*Dept. of Applied Mathematics, National Sun Yat-Sen University, Kaohsiung, Taiwan, R.O.C.*

Received 6 November 1996; accepted 24 March 1997

Abstract

In order to properly schedule related multimedia objects, synchronization relationships of multimedia objects should be precisely specified and dispatched. Each multimedia presentation schedule contains two parts: (1) the state-transition control part, which specifies intra-medium and inter-media synchronization information, and (2) the data variables part, which specifies the dynamic aspects of the state-transition control for dealing with jitter and skew. In this paper, we propose a specification language for specifying multimedia synchronization. The language is called M²EST, which represents the MultiMedia Extended State Transition. M²EST can handle both the state-transition control part and the data variables part in multimedia presentation scheduling. Using M²EST, the temporal behavior of each medium stream is handled by an actor extended finite state machine (EFSM). The temporal relationships among media streams are handled by a synchronizer EFSM. Synchronizer and actors perform multimedia presentations cooperatively. The corresponding synchronization schemes, including both intra-stream and inter-stream synchronization schemes, which rectify the random networks delays caused on distributed presentation environment, can also be specified using M²EST. © 1997 Elsevier Science B.V.

Keywords: Multimedia systems; Distributed multimedia synchronization; Formal specification languages

1. Introduction

With advances in all areas of computer science, it is time to integrate advanced techniques to have new computing environments. 'Multimedia systems' is one of the typical computing platforms that results from the new consideration [1,2]. Multimedia systems, which combine text, graphics, image, audio, video, and/or animation, enhance the representation of information. A multimedia data object always consists of different information media (1) that are input from different media bases, which may be located remotely, and (2) that should be displayed concurrently on different locations, which may be on different devices. Thus, displaying multimedia data objects should have precise coordination among these media in order to have smooth and correct presentations. In other words, multimedia synchronization, i.e. the temporal relationships among the media to be presented, is the key issue for having smooth multimedia presentations [3,4].

Currently available media can be classified into two types: one is the static type and the other one is the continuous type. Static media, e.g. text, still images, and graphics, have no temporal property themselves. The temporal properties of static media are synthetically defined by applications. Continuous media, i.e. audio, video, and animation, essentially consist of sequences of data units. There are embedded temporal properties in continuous media. In distributed environments, since it is very hard to have constant delays from the commencement to the end of a multimedia presentation, jitter and skew phenomena always exist during presentation [5]. As a result, there are two types of multimedia synchronization, i.e. inter-media synchronization and intra-medium synchronization. Inter-media synchronization maintains the requirements of the temporal relationships between two or more media, such as lip-synchronization. Intra-medium synchronization controls the displaying schedule (rate) of one medium's composed data units. If network delays are not constant, some schemes, e.g. blocking and restricted blocking schemes [6], can be used to resolve the synchronization problems.

Currently, a lot of models/languages have been proposed to formally describe temporal relationships in multimedia presentations [7–10]. We roughly classify these models/languages into two types. The first type belongs to state-transition specifications, e.g. the object composition Petri net (OCPN) model [6,10], which is based on the Petri net model [11,12] and the data-flow-graph model [7]. The state-transition specifications use the concept of 'state'

* Corresponding author. E-mail: huangcm@locust.iie.ncku.edu.tw

PII S0950-5849(97)00012-8

and 'transition' to describe temporal relationships of multimedia objects. The state-transition specifications can represent the control part precisely, but the data part cannot be represented clearly. The other type is programming-language-based specifications [8,9]. Using the programming-language-based specifications, all of the temporal relationships are described by a set of procedure-based statements, which are executed sequentially or in parallel. The programming-language-based specifications can specify the dynamic behaviors of synchronization controls in detail. However, the state-transition control part cannot be precisely identified because programming-language-based specifications lack a general syntax to describe the state-transition control part.

Based on our proposed Extended Finite State Machine (EFSM) based multimedia synchronization model [13], this paper proposes a hybrid formal script language that provides mechanisms for specifying both the state-transition control part and the dynamic behaviors part in multimedia synchronization. The hybrid specification language is called M^{2}EST, which represents the MultiMedia Extended State Transition. The use and proposal of M^{2}EST is inspired by Estelle [14,15], which is an ISO Formal Description Technique for formally specifying communication protocols. That is, M^{2}EST is a modification/extension of Estelle. In M^{2}EST, (1) the control part is realized by a set of EFSMs, and each EFSM contains a number of state-transitions, and (2) the dynamic behaviors part is realized by using some programming-language-based, i.e. Pascal-like, statements. Temporal relationships among presented data objects can be precisely specified using M^{2}EST. The state-transition part in an M^{2}EST specification describes synchronization controls; the programming-language part in an M^{2}EST specification operates on related context variables to specify the dynamic aspects of the state-transition part for dealing with jitter and skew.

Using M^{2}EST, each individual medium stream is represented as an EFSM, which is called an actor EFSM, and the synchronization among media streams is also handled by an EFSM, which is called a synchronizer EFSM. Each actor EFSM controls intra-medium synchronization. The synchronizer EFSM controls inter-media synchronization. The synchronizer holds information about temporal relationships among all streams, and each actor denotes one stream and controls the data flow of the associated stream. The communication between synchronizer and actors is message-passing through defined channels. Since M^{2}EST contains the capability of programming-language-based descriptions, the synchronization schemes can also be formally specified using M^{2}EST.

In a distributed multimedia synchronization environment, an M^{2}EST specification uses the client–server model to describe the presentation of the multimedia objects. When a remote client selects one scenario for the presentation, the server site transfers media objects to the client site according to the scenario. At the server site, the server sends multimedia objects according to the scenario and control messages. At the client site, the client receives multimedia objects and presents these objects according to the scenario and control messages. In the current M^{2}EST system, the synchronization environment is on the top of the transport layer, which is currently TCP/IP over asynchronous transfer mode (ATM).

The rest of this paper is organized as follow. Section 2 briefly introduces the language constructs of M^{2}EST and the related knowledge for applying M^{2}EST to multimedia synchronization. Section 3 presents the distributed synchronization control architecture, and the M^{2}EST-based specifications for distributed multimedia synchronization. Section 4 introduces synchronization schemes for dealing with random network delays. Section 5 gives the development issues of the M^{2}EST system. Section 6 contains some discussion and concluding remarks.

2. The language construct of M^{2}EST

An M^{2}EST specification consists of a set of hierarchical cooperating entities. Each entity is described as a module. Fig. 1 shows the abstract format of an M^{2}EST-based multimedia scheduling protocol specification. The actual behavior of each module is described by a set of submodules, or an Extended Finite State Machine (EFSM) at the innermost level. Each module interacts with other modules by exchanging messages through channels. All of the modules are executed in parallel. A channel is defined as a bidirectional first-in-first-out (FIFO) pipe. A channel transmits messages between two connected interaction points (ips) that are allocated in modules. Modules A and B can exchange messages through a channel that links two interaction points x and y, in which x and y are in A and B respectively. Each EFSM consists of a set of transitions. Each transition consists of a *from* state, a *to* state, a *when* clause (input event), a *provided* clause (the predicate), a *priority* clause, a *delay* clause, and a list of Pascal-like statements that describe actions. A *when* clause represents an input event. A *provided* clause consists of a set of Boolean expressions. A *delay* clause is represented as delay $[t_{min},t_{max}]$. A *priority* clause is represented as priority n, where n represents the priority level. If there is no priority clause, the priority is middle. The action part, which is delimited by keywords *begin* and *end*, can contain some output events and a number of Pascal-like statements that operate on context variables. Each EFSM can be represented as a nine-tuple $(\Sigma,S,s_0,V,E,T,P,A,\delta)$, where

- Σ is the set of messages that can be sent or received,
- S is the set of the states,
- s_0 is the initial state,
- V is the set of context variables,
- E is the set of predicates that operate on context variables (*provided* clauses),

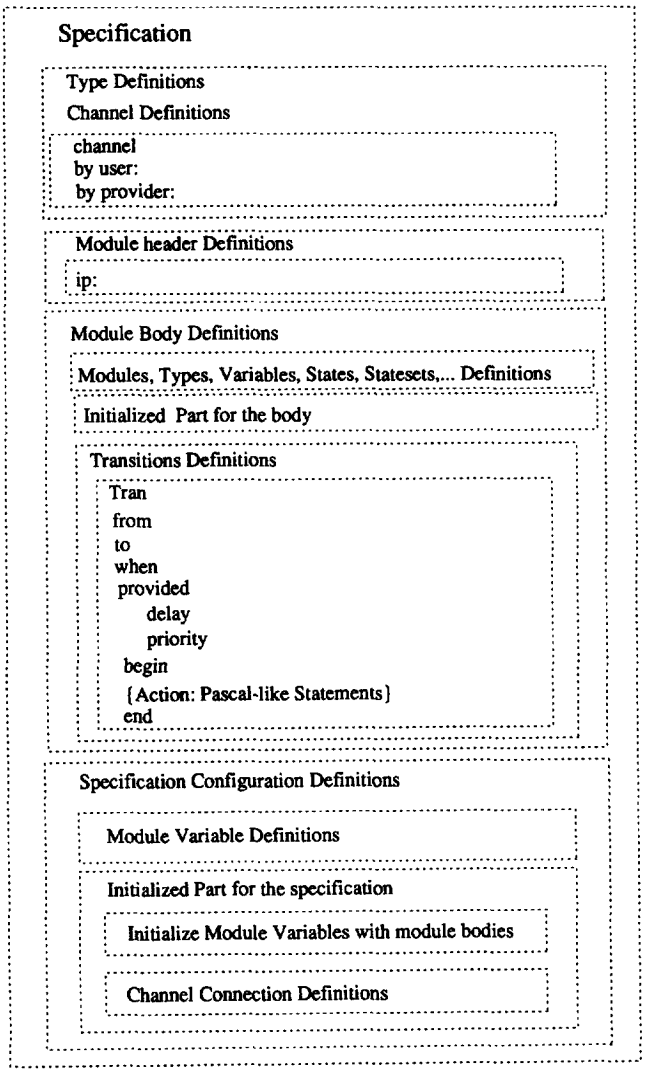


Fig. 1. The abstract format of an M²EST-based specification.

- T is the set of time intervals (*delay* clauses),
- P is the set of *priority* clauses,
- A is the set of actions that operate on context variables,
- δ is the set of transition functions, where each transition function can be formally represented as follow: $S \times \Sigma \times E(V) \times \mathrm{P} \times T \rightarrow \Sigma \times A(V) \times S$

A transition can be executed when the condition part is true, i.e. (1) the input event is available, (2) the predicate is true, (3) its priority is the highest among executable transitions. Let the delay clause be $[t_{\min}, t_{\max}]$. When the conditions are satisfied, the transition can be executed after $t + t_{\min}$ and should be executed before $t + t_{\max}$, where t is the time the associated EFSM enters into the transition's head state. If $t_{\min}$ is equal to $t_{\max}$, it means that the corresponding transition should be fired at $t + t_{\min}$.

Each module has a number of interaction points for communicating with other modules. The links of interaction points are classified into two types:

- attached: two interaction points are *attached* when the associated two modules have parent/children relationships;
- connected: two interaction points are *connected* when the associated two modules have sibling relationships.

An M²EST specification and each module inside the specification mainly contain three definitions: type and channel definitions, module header definitions, and module body definitions. Type definitions define some declared structure types, variables' types, constant definitions, etc. Channel definitions define channel types, and the corresponding input/output messages. Each channel is associated with two roles, e.g. user role and provider role. An interaction point ip that is declared with channel type C's user (provider) role can output the messages that are associated with C's user (provider) role, and input the messages that are associated with C's provider (user) role. Some examples are as follows:

```
channel VCSAatS (User, Provider);      module SSYN;
by User: Transmit; DisResp;            ip
by Provider: Over(Vend: integer);      SSIP1: VCSAatS(User)
ConInd;                                BUFFER [2];
                                       SSIP2: ACSAatS(User)
                                       BUFFER [2];
                                       SSIP3: ICSAatS(User) BUFFER [2];
```

where interaction point SSIP1 in module SSYN uses channel type VCSAatS with role User, i.e. SSIP1 can output messages belonging to the User role, e.g. message Transmit in the above example, and can input messages belonging to the Provider role, e.g. message Over in the above example. Each interaction point is associated with an input buffer, which is denoted by BUFFER[n] and n is the buffer size. Depending on the delay, jitter, bandwidth, synchronization schemes, etc., different buffer sizes can be assigned.

A module header definition can specify attributes, interaction points, channel definition, the types of exchanged messages, etc. The attributes of modules are assigned as follows:

- The outermost *specification* module, i.e. the whole specification, is attributed as CENTRALIZED for the centralized environment case, or as DISTRIBUTED for the distributed environment case.
- When the outermost module is attributed as DISTRIBUTED, modules in the second hierarchy are attributed as SOURCE for the server site, in which media bases are located, or as DESTINATION for the client site, in which media information is received for displaying.
- The innermost module is non-attributed.

A module body definition includes declaration part, initialization part, and transition part. The declaration part declares (sub-)modules, module variables with their associated module types, interaction points, structure types, variables, functions/procedures, states, etc. The

initialization part specifies the initial configuration, i.e. the initial state, the initial values of variables, the assignment of module variables with the corresponding module bodies, the connection information of *attached* and/or *connected* interaction points etc., of the module (EFSM) that is associated with the module body. The transition part specifies the transitions of the module associated with the module body.

For the outermost module, i.e. the specification module, the specification configuration, which is denoted by the 'Specification Configuration Definitions' in Fig. 1, mainly contains two parts: module variables definition and the initialized part for the specification module. Module variables definition declares module variables with their associated module types. The initialized part for the module initializes module variables with the associated module bodies and sets up the channel connections, i.e. the two interaction points that are attached or connected.

In distributed environments, two sets of EFSMs are required for M^{2}EST-based formal synchronization specifications, one for the source site and the other for the destination site. The corresponding M^{2}EST specification is attributed as DISTRIBUTED at the outermost level; two modules, in which the one attributed as DESTINATION represents the destination site's module and the other one attributed as SOURCE represents the source site's module, are specified at the second level; synchronizer EFSMs and actor EFSMs are specified at the third level. Fig. 2, in which MMSynchronization, Client, and Server are the identifiers of these modules, shows the abstract M^{2}EST-based specification architecture that is used in distributed environments. In distributed environments, each actor at the destination site should be able to communicate with its peered actor at the source site via networks. With reference to Fig. 2 for the following explanation, the communication is achieved by the following configuration: (i) an interaction point of each actor is attached to its parent module, i.e. DA*i* (SA*i*) is attached to DSP*i* (SSP*i*) at the destination (source) sites; (ii) an interaction point DSP*i* at the destination site is connected to the peered interaction point SSP*i* at the source site; (iii) an interaction point DAIP*i* (SAIP*i*) of each actor is connected to the interaction point DSIP*i* (SSIP*i*) of the synchronizer at the destination (source) site, where $i = 1,...,n$. At the source site, actor EFSMs collect media units from source devices when the related media streams are ready, and the synchronizer EFSM issues the signal for the commencement of transmission. Actor and synchronizer EFSMs at the destination site cooperatively coordinate the presentation.

The complete formal syntactic definition of M^{2}EST is explained in reference [16].

3. M^{2}EST-based formal specifications of distributed multimedia synchronization

Essentially, there are four synchronization control architectures in distributed environments. Fig. 3 shows the abstract configurations of these four architectures. The first type, which is depicted in Fig. 3(a), has no synchronization controller at both the destination site and the source site. The second (third) type has a client (server) synchronization controller at the destination (source) site. In this case, there is some coordination during media displaying (transmission) at the destination (source) site. But there is no coordination at the source (destination) site. The fourth type has a server and a client synchronization controller at the source and destination sites respectively. Coordination can be invoked during media displaying/transmission at the destination/source sites respectively. Depending on (1) the available computing and communication environment, e.g. whether the global clock exists or not, (2) the available equipment, i.e. low-end and cheaper or high-end and more expensive, set-top boxes or computers, (3) the required qualities of presentations etc., different multimedia applications can adopt one of the four synchronization control architectures to achieve their own expected

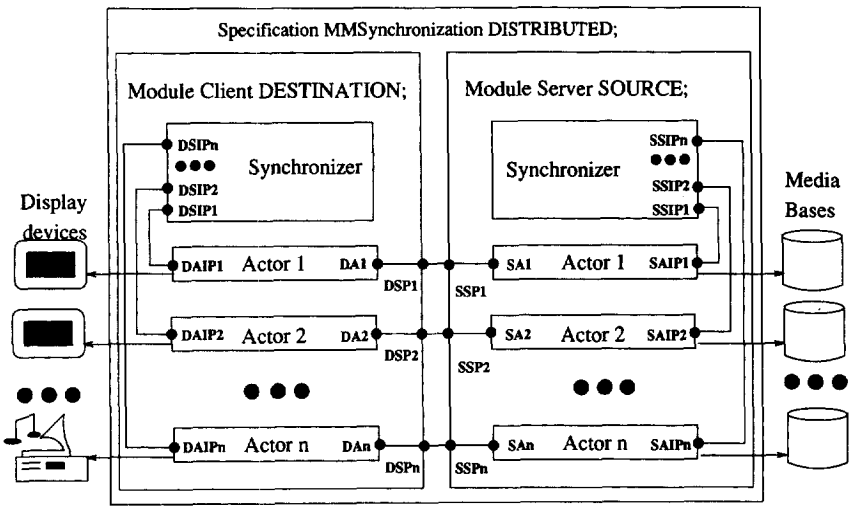


Fig. 2. The abstract M^{2}EST-based specification architecture in distributed environments.

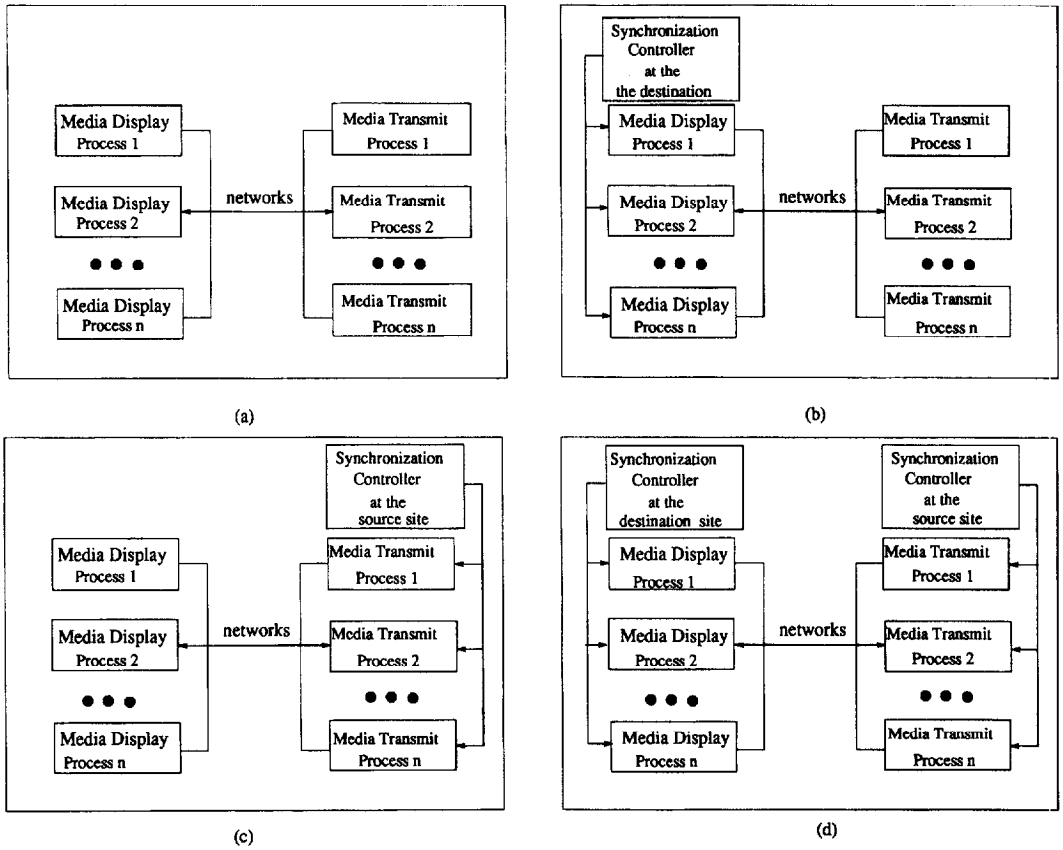


Fig. 3. The abstract configurations of the four synchronization control architectures.

presentations.[1] For convenience, we use the fourth architecture for explanation.

Depending on the underlined networking environments and protocols, various computing and networking resources are required to achieve the expected presentation qualities. Let the underlined networks be ATM-based. In order to maintain the continuity, multimedia applications can tolerate errors that result from packet corruption or loss without retransmission or correction [1]. ATM-based networks have low cell lost/garbled rates, and it is expected that there is no bursty cell loss. When a (some) cell(s) is (are) lost/garbled, the corresponding place(s) at the medium unit, e.g. a video frame, which consists of a lot of cells, can be filled with some default or derived information. Thus, we assume that all of the media units will not be lost, but may be presented with some minor erroneous information.[2]

Fig. 4(a) shows a display-time bar chart of a multimedia presentation in a distributed environment. There are four presentation stages in the video stream and the audio stream respectively, and two presentation stages in the image stream. The interaction sequence of the presentation is depicted in Fig. 4(b). The corresponding M^{2}EST-based formal specifications of actors and Synchronizers of the presentation are depicted (1) in Fig. 5, where a circle represents a state and an arrow represents a transition, and (2) in the appendices, where the statement *when ip.mess* (*output ip.mess*) represents *mess* is input (output) from (to) interaction point *ip* of the EFSM.[3]

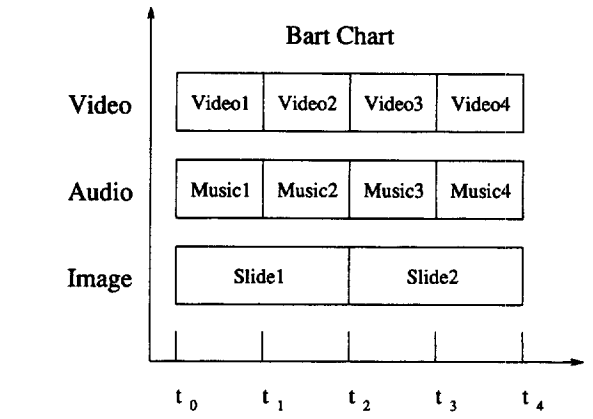
Since two sets of actors communicate with each other via networks, some transitions for connection setup and for disconnection are needed in each actor EFSM and each synchronizer EFSM. With reference to Fig. 5 and Appendices A, B, C and D, which contain the contents of transitions, and the corresponding interaction sequences depicted in Fig. 4(b), (1) transitions T_1, T_2, T_3, and T_4 of the synchronizer at the destination site, (2) transitions T_1 and T_2 of the actors at the destination site, (3) transitions T_1, T_2, and T_3 of the synchronizer at the source site, and (4) transition T_1 of the actors at the source site are used to have the connection setup.

Two sets of EFSMs that are geographically separated cooperatively achieve distributed multimedia presentations

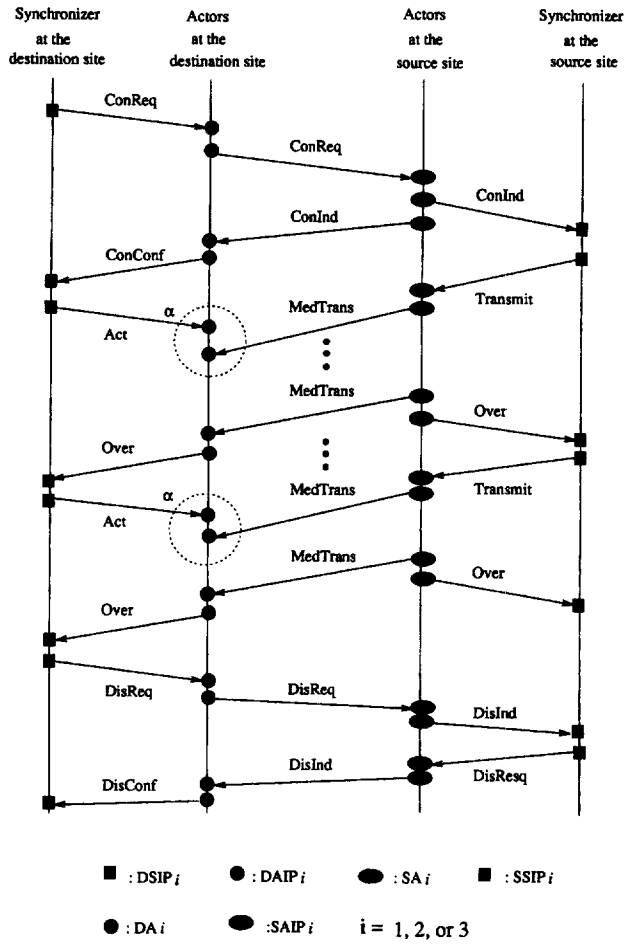
[1] For applications with multiple destination (source) sites, e.g. *n*-party video conferencing, each destination (source) site has one set of EFSMs. With *n* destination sites, media information is broadcast to these *n* destination sites. With *n* source sites, *n*-set of media information sent from *n*-set of EFSMs at these *n* source sites are received by the associated set of EFSMs at the destination site.

[2] This scheme is suitable for uncompressed data or compressed data whose adopted compression techniques are intra-medium-unit-based, e.g. JPEG. For compressed data whose adopted compression techniques are inter-media-unit-based, e.g. MPEG, different schemes should be used because a faulty cell in a medium unit will affect not only the current medium unit but also the following *k* media units.

[3] For simplicity, T represents TRUE and F represents FALSE in this paper.

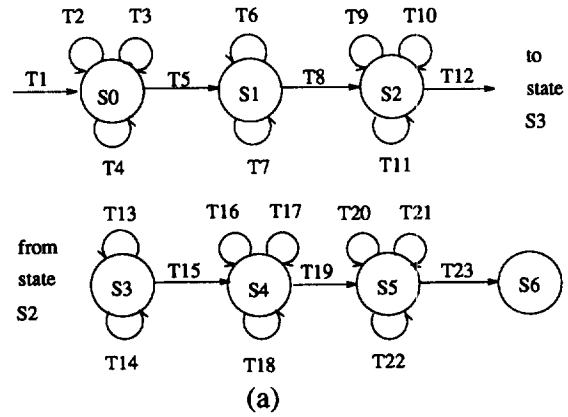


(a)

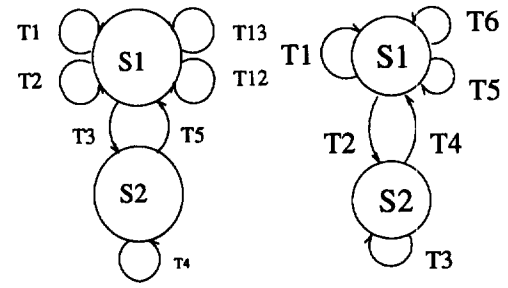


(b)

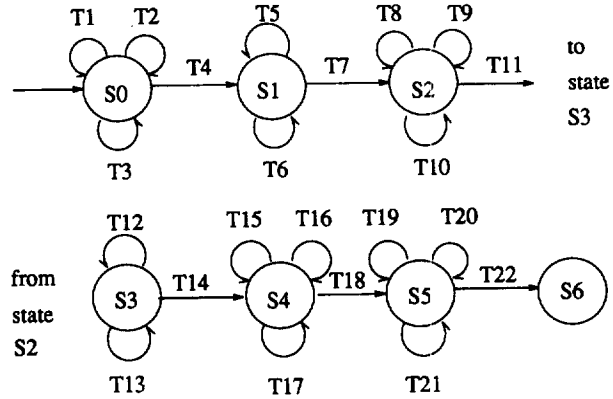
Fig. 4. (a) The display-time bar chart of a multimedia presentation; (b) the corresponding interaction sequence.



(a)



(b) (c)



(d)

Fig. 5. (a) The synchronizer EFSM at the destination site; (b) the actor EFSM at the destination site; (c) the actor EFSM at the source site; and (d) the synchronizer EFSM at the source site.

after all EFSMs have executed the initialization transitions for connection setup. The synchronizer at the source site issues the transmission signals in transitions T_4, T_7, T_{11}, and T_{14}, i.e. sending 'Transmit' messages to actors, which are depicted in Fig. 5(d), and waits for the 'Over' messages, which are sent from actors, in the corresponding transitions. Each actor at the source site starts to transmit media units when the 'Transmit' message is received. For pre-orchestrated presentations, transmission schedules are always pre-decided according to the associated networks protocols and the required presentation qualities. The time, i.e. $T(stage,counter) - T_s$, specified in delay clauses of transition T_3 which are depicted in Appendix C, is used to control the transmission time of each medium unit. An

example of a transmission schedule is as follows: (1) the delay time for the very first medium unit of each stage, i.e. $T(i,0) - T_s$, is 0; (2) the delay time is $(1/\theta) - T_s$, for the other media units, i.e. $T[i,j], j \neq 0$, where θ is the presentation rate and T is the computing overhead, including the overhead of transmitting a medium unit and the overhead of other computation in the action part of T_3. The delay clauses may not be required for live presentations, e.g. video conferencing, because actors immediately transmit media units that are generated from the live media generators.

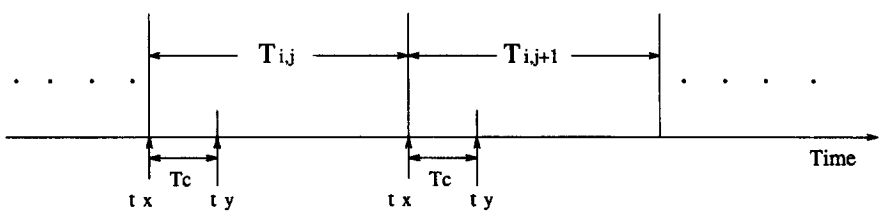
The synchronizer at the destination site notifies actors to receive media units and to display these media units by sending 'Act' messages in transitions T_5, T_8, T_{12}, and T_{15}, which are depicted in Fig. 5(a). An actor sends an 'Over' message when a presentation stage is finished. In Appendix A and Appendix D, variables *VIDstage*, *AUDStage*, and *IMAstage* denote the current presentation stage of video, audio, and image media respectively; variables *Vend*, *Aend*, and *Iend* denote currently finished presentation stage of video, audio, and image media respectively, variables *VIDplay*, *AUDplay*, and *IMAplay* denote the presentation status of video, audio, and image media respectively. When a video, audio, or image presentation stage is finished, *VIDplay*, *AUDplay*, or *IMAplay* becomes TRUE.

In Appendix B and Appendix C, variable *stage* denotes the *stage*th presentation stage; variable *Length*[*stage*] represents the number of media units to be presented at the *stage*th presentation stage; variable *counter* is the counting number of the just presented medium unit; variables *VIDframe*, *AUDseg*, and *IMAdata* contain the media units that are stored in video, audio, and image media buffers respectively. For static media, the displaying duration of the corresponding static data unit is contained in the parameter 'duration(stage, counter)', which is in the delay clause of transition T_4 that is depicted in Appendix B. In the delay clause, T_c represents the computing overhead, which is the overhead of executing the action part of T_4. Since the display duration of the last piece of image in each stage is contained in transition T_5's delay clause, 'duration(stage,0) $- T_c$' is equal to 0. In the case of continuous media, i.e. video and audio, a medium unit is the minimum data component to be displayable. The corresponding video, audio, and image media displaying system routines are contained in procedures *PlayVideo*, *PlayAudio*, and *PlayImage* respectively. Parameters *position*, *width* and *height* in *PlayVideo*, *PlayAudio*, and *PlayImage* denote the coordinate of the upper-left corner, the width, and the height of the displaying window respectively. In video and audio actors, 'interval[i,j] $- T_c$' in the delay clause of transition T_4 is (1) equal to 0, if $j = 0$, (2) equal to $(1/\theta) - T_c$, if $j \neq 0$, where $1/\theta$ is a video frame's or an audio segment's presentation time, e.g. 1/30 second, and T_c is the computing overhead. The computing overhead is the overhead of executing the action part of T_4. Fig. 6 depicts the abstract time configuration of executing media displaying transitions. For the same reason as that in the image actor, i.e. the presentation time of the last frame (segment) in the video (audio) actor is contained in transition T_5's delay clause, 'interval[i,0] $- T_c$' is equal to 0. With reference to Fig. 4(b), racing conditions may occur at the α point, i.e. an actor may receive message 'MedTrans' earlier. When racing occurs, message 'MedTrans' will be stored at buffers until message 'Act' is received and the corresponding actor EFSM chances its state to S2, according to the presentation schedule depicted in Fig. 5.

When the presentation is over, network connections should be released. The disconnection event is triggered by the synchronizer at the destination site. When the final presentation stage is finished, transition T_{19} in Fig. 5(a) is executed. Transitions T_{19}, T_{20}, T_{21}, T_{22} and T_{23} of the synchronizer at the destination site, transitions T_{12} and T_{13} of the actors at the destination site, transitions T_{18}, T_{19}, T_{20}, T_{21}, and T_{22} of the synchronizer at the source site, and transitions T_5 and T_6 of the actors at the source site are used to have network disconnection.

4. Specifications of synchronization schemes

In distributed environments, multiple data streams are



T i,j - For static media: the displaying time length of "duration(i, j)".
- For continuous media: the displaying time length of "interval(i, j)".

Tc - The overhead of executing the action part of T4.

t x - The time when the EFSM starts to execute the media displaying routines.

t y - The time when the EFSM enters into the tail state of T4 and starts to select a transition to be executed.

Fig. 6. The time configuration of executing media displaying transitions.

retrieved from the remote source site and displayed at the destination site. Since the computing platform is in distributed environments, the random network delays impose some jitter delays in each individual stream, or impose some skews among streams at the destination site. Thus, synchronizer and actors at the destination site must take some commencement and synchronization schemes to compensate or smooth abnormal situations.

The frequency of invoking re-synchronization processing is application-dependent. Considering the following CNN report presentation: at stage I, the video displaying window displays a reporter, who is giving some news about some pandas in mainland China. At stage $I + 1$, the video displaying window is switched to display some pandas, which is accompanied by the reporter's oral explanation. In this situation, stage I should invoke the re-synchronization process more often than stage $I + 1$. The main reason is that human beings need more accurate synchronization for the lip presentation, i.e. lip-synchronization. Thus, there are two synchronization types, one is coarse-grain and the other one is fine-grain, in which the re-synchronization process is invoked very frequently. Coarse-grain synchronization is suitable for synchronizing presentation stages. When the related media need tight synchronization, e.g. lip-synchronization between audio and video, fine-grain synchronization should be invoked.

4.1. Commencement control

To have smooth commencement of a multimedia presentation, some commencement schemes can be adopted. A commencement scheme is as follows: the commencement can be invoked when each medium stream has buffered at lease one medium unit. To adopt the commencement scheme, the action part of each transition T_2 of the actor EFSMs at the destination site, which is depicted in Appendix B, should be modified: the statement 'bufferfilled(1)' is added before the statement of 'Output DAIPi.ConConf;' where $i = 1$, 2, or 3. Procedure 'bufferfilled(1)', where '1' identifies the number of media units to be filled as 1, tests whether 1 medium unit is received or not. When 1 medium unit is received, 'bufferfilled(1)' returns.

4.2. Coarse-grain synchronization schemes

Intra-medium synchronization controls the displaying schedule of one medium's composed data units. There are three intra-medium synchronization schemes: blocking, restricted blocking, and non-blocking.

- The blocking scheme: if an expected medium unit does not arrive on time, the actor suspends its presentation until the medium unit arrives.
- The restricted blocking scheme: if an expected medium unit X does not arrive on time, the actor blocks for a predefined interval δ. If medium unit X arrives during the interval δ, the actor still displays X; otherwise, if X does not arrive during the interval δ, the actor skips the demanded medium unit X but displays the most recently received medium unit.
- The non-blocking scheme: If an expected medium unit does not arrive on time, the actor immediately redisplays the most recently received medium unit. The re-display procedure is repeatedly executed until the expected medium unit arrives.

The actor EFSMs that are depicted in Fig. 5(b) in fact adopt the blocking scheme. During presentation, each actor is suspended at state S2 till the corresponding medium unit arrives. When the expected medium unit arrives, which is contained in message ' MedTrans', transition T_4 is executed. The restricted blocking and non-blocking schemes can be formally specified using M^{2}EST by augmenting some transitions and/or states into original EFSMs. For simplicity, we use the restricted blocking scheme as an example.

To keep the continuity property, the blocking scheme is not suitable for video media. The restricted blocking scheme can be adopted. Based on the video Actor EFSM depicted in Fig. 5(b), the restricted blocking scheme is achieved by adding transitions T_6 and T_7, and modifying transitions T_3 and T_4:

```
{T3}
from S1
to S2
when DAIP1.Act

begin
counter: = 0;
replay: = 0;
end

{T6}
from S2
to S2
priority low
provided counter < > 0
and counter < Length[stage]
delay [interval(stage,counter) − Tc +
tR,interval(stage,counter) − Tc + tR]
begin
PlayVideo(oldframe);
counter: = counter + 1;
replay: = replay + 1;
end
```

```
{T4}
from S2
to S2
delay [interval(stage, counter) −
Tc,interval(stage,counter) − Tc]
when DA1.MedTrans(VIDframe)
begin
PlayVideo(VIDframe);
oldframe: = VIDframe;
counter: = counter + 1;
end

{T7}
from S1
to S1
provided replay > 0
priority high

begin
adjustbufferpointer(replay)
end
```

Based on the added/modified transitions, the restricted blocking scheme is achieved as follows: if the video actor stays at state S2 for t_R time units more, the video actor gives up waiting for the demanded medium unit but displays the most recently received medium unit. Transition T_6 is in charge of the re-displaying action. Variable *counter*, which denotes the number of displaying frames, is increased by 1 when transition T_6 is executed once. Variable *replay* denotes the number of re-displaying in a stage, i.e. the number of executing transition T_6 in a stage. The priority

clause 'priority low' in T_6 is used to resolve the collision condition: when the expected medium unit arrives at the due time, i.e. both transitions T_4 and T_6 become executable, T_4 is still the selected transition to be executed. Transition T_4 is modified to record the latest received medium unit, i.e. 'oldframe: = VIDframe'. Transition T_3 is modified to set the initial value of replay as 0. Transition T_7 is used to skip the lately received media units in order to play the right media units in the next presentation stage, i.e. adjust the bufferpointer using procedure 'adjustbufferpointer()' to the right medium unit.

The purpose of inter-media synchronization is to maintain the temporal relationships among streams. There are three inter-media synchronization schemes: parallel-first, restricted parallel-first, and parallel-last.

1. When the parallel-first scheme is adopted, all of the related streams should keep pace with the first terminated stream to conform synchronization. In other words, the slower streams drop some media units to keep pace with the fastest one.
2. The restricted parallel-first scheme is a modified parallel-first scheme, in which some delay tolerance is allowed. Let the delay tolerance be t_1. When the restricted parallel-first scheme is adopted, the synchronizer terminates all actors' execution at time $t + t_1$, where t is the time the synchronizer receives the first 'Over' message.
3. When the parallel-last scheme is adopted, the synchronizer issues the signal for next presentation stage after having received all actors' 'Over' messages. In other words, the faster streams wait for a while so that the slower streams can keep pace with the faster ones.

These inter-media synchronization schemes can also be specified using M^{2}EST. For simplicity, the parallel-first scheme is used as an example.

Based on the parallel-first scheme, the synchronizer terminates all actors' execution as soon as the first 'Over' message sent by an actor is received. Some transitions should be added/modified in the synchronizer EFSM, which is depicted in Fig. 5(a), to achieve the parallel-first scheme. For simplicity, the modified transition, i.e. T_8, and the added transitions, i.e. $T_{9'}$ and $T_{10'}$, at states S_1 and S_2 are used for explanation. Added/modified transitions in other states can be derived in the same way:

```
{T8}
from S1
to S2
provided (VIDplay = T)
or (AUDplay = T)
begin
if VIDplay = T
then output DSIP2.Stop;
else output DSIP1.Stop.,
endif
VIDstage: = VIDstage + 1;
output DSIP1.Act;
VIDplay: = F;
AUDstage: = AUDstage + 1;
output DSIP2.Act;
AUDplay: = F;
end
```

```
{T9'}
from S2
to S2
priority high
when DSIP1.Over(Vend)
provided Vend < > VIDstage
begin
end

{T10'}
from S2
to S2
priority high
when DSIP2.Over(Aend)
provided Aend < > AUDstage
begin
end
```

Transition T_8 is modified to perform the required actions: (1) the predicate is modified as follows: 'VIDplay = T or AUDplay = T'. Thus, when one of the transitions T_6 and T_7 at state S_1 has been executed, transition T_8's predicate becomes true and T_8 is executed. (2) The actions in T_8 include transmitting 'Stop' messages to other actors in order to force other actors terminating their current presentation stages and having the next presentation stages. Collision conditions may occur: it is possible that actor X finishes its current presentation stage during T_8's execution time, i.e. X sends message 'Over' before receiving message 'Stop'. Thus, transitions $T_{9'}$ and $T_{10'}$ are added to receive these 'Over' messages that are sent from the video actor and the audio actor respectively, after the execution of T_8.

When the parallel-first scheme is adopted, actors need to add some transitions to receive the 'Stop' messages transmitted by the modified synchronizer. The added transitions for the video actor EFSM are as follows:

```
{T8}
from S2
to S1
priority high
when DAIP1.Stop
begin
replay: = Length[stage] − counter + replay;
adjustbufferpointer(replay);
stage: = stage + 1
end
```

```
{T9}
from S1
to S1
when DAIP1.Stop
begin
end
```

The modified specification for other actors can be derived in the same way. Transitions T_8 and T_9 can receive the 'Stop' message sent from the Synchronizer, in which T_9 is used to receive the 'Stop' message when the collision condition occurs. The action part of T_8 adjusts the bufferpointer to the first medium unit of the next stage.

Let the sending rate and presentation rate be equal and the commencement scheme be the one that is depicted in Section 4.1. When the audio stream adopts the blocking scheme and the video stream adopts the restricted blocking scheme, the relationship among the buffer size, the maximum jitter J_{max}, and the sending and presentation rate θ, and the incurred intra-medium and inter-media asynchrony anomalies are as follows. In order to guarantee the quality of synchronization and avoid data loss, the transport protocol must allocate at least $1 + [J_{max}/\theta]$ buffer units to the video stream that adopts the restricted blocking scheme, and allocate at least $1 + [J_{max}/\theta]$ buffer units to the audio stream that adopts the blocking scheme. The incurred maximum intra-medium asynchrony anomalies for the video stream

that adopts the restricted blocking scheme and for the audio stream that adopts the blocking scheme are $\theta + {}_{V}J_{max}$ and ${}_{A}J_{max}$ respectively. The corresponding incurred maximum asynchrony anomaly is the maximum value of $\theta + {}_{V}J_{max}$ and ${}_{A}J_{max}$ [17].

4.3. Fine-grain synchronization schemes

Parallel-first and parallel-last schemes can be adopted for fine-grain synchronization. For simplicity, the parallel-first fine-grain synchronization control is used as an example.

The added/modified transitions, which are at state S_3, for the Synchronizer EFSM are as follows:

```
{T12}
from S2
to S3
provided
(VIDplay = T)
and (AUDplay T)
and (IMAplay T)
begin
VIDstage:
= VIDstage + 1;
output DSIP1.Act;

VIDplay: = F;
AUDstage:
= AUDstage + 1,
output DSIP2.Act;
AUDplay: = F;

IMAstage: = IMAstage
+ 1;
output DSIP3.Act;
IMAplay: = F;
VIDfine: = 1;
AUDfine: = 1;
end

{T13'}

from S3

to S3
when DSIP1.FineO-
ver(Vfineend)
begin
VIDfineact: = T;
end
```

```
{T13"}
from S3
to S3
when DSIP1.
Fineover(Vfineend)
provided (Vfineend <
> VIDfine)
begin
end

{T14'}
from S3

to S3
when DSIP2.Fineover
(Afineend)
begin

AUDfineact: = T;
end

{T14"}
from S3
to S3
when DSIP2.FineOver
(Afineend)
provided Afineend <
> AUDfine
begin
end
```

```
{T15'}
from S3
to S3
provided (VIDfineact =
T)
or (AUDfineact = T)

begin
if VIDfineact = T
then output DSIP2.
FineStop;
else output DSIP1.
FineStop;
VIDfine: = VIDfine + 1;
output DSIP1.FineAct;

VIDfineact: = F;
AUDfine: = AUDfine +
1;
output DSIP2.FineAct;

AUDfineact: = F;
end
```

The added/modified transitions at other states can be derived in the same way. Using the fine-grain parallel-first synchronization scheme, transition $T_{15'}$ of the synchronizer is executed when the synchronizer receives message 'FineOver' from one actor: transition $T_{15'}$ (1) sends message 'FineStop' to the other actor in order to terminate the current fine-grain synchronization cycle, and (2) sends message 'FineAct' to invoke the next fine-grain synchronization cycle. Transitions $T_{13''}$ and $T_{14''}$ are added to deal with the collision condition.

Some transitions, i.e. T_{10}, T_{11}, T_{14}, and T_{15}, and a new state S_3 need to be added in the corresponding video actor EFSMs.

```
{T10}
from S2
to S3
priority low
provided (counter < > 0)
and (counter mod SynNo = 0)

begin
output DAIP1.FineOver(finestage);
finestage: = finestage + 1;
end

{T11}
from S3
to S2
when DAIP1.FineAct
begin
end
```

```
{T14}
from S2
to S3
when DAIP1.FineStop
begin
skip: = SynNo − (counter mod
SynNo);
counter: = counter + skip;
adjustbufferpointer(skip);
end

{T15}
from S3
to S3
when DAIP1.FineStop
begin
end
```

The corresponding audio actor part can be derived similarly. Let each fine-grain synchronization cycle be invoked after every SynNo media units has been displayed, which is controlled by the predicate 'counter **mod** SynNo = 0' in transition T_{10}. Transition T_{10} sends message 'FineOver' to notify the synchronizer that the current fine-grain synchronization cycle is finished. A new fine-grain synchronization cycle can be commenced when one actor has sent the message 'FineOver' to the synchronizer, and message 'FineAct' has been sent to each actor by the synchronizer in transition $T_{15'}$ of the synchronizer EFSM. When an actor receives the message 'FineAct', i.e. transition T_{11} in each actor, the actor starts to display newly arrived media units. When $j = k^*$SynNo, $k = 1,2,3,\ldots$, the computing overhead T_c in 'interval(i,j) − T_c', which is in the delay clause of transition T_4 that is depicted in Appendix B, is changed. The computing overhead T_c should include the execution time of transitions T_{10} and T_{11}. Transition T_{14} calculates the number of media units to be skipped, and adjusts the buffer pointer to the right medium unit for the next fine-grain synchronization cycle. Transition T_{15} is used to receive the 'FineStop' message when the collision condition occurs.

4.4. Other specifications

The transitions parts of the actor and synchronizer EFSMs are specified and explained in the previous subsections. The corresponding M^{2}EST specifications also include the following parts.

1. Channel definitions: channels between the synchronizer and actors at the source site, between the synchronizer and actors at the destination site, and channels between

actors at the source site and actors at the destination site.

2. Module header definitions for the synchronizer, i.e. DSYN, and for actors, i.e. DACT1, DACT2, and DACT3, at the destination site; module header definitions for the synchronizer, i.e. SSYN, and for actors, i.e. SACT1, SACT2, and SACT3, at the source site; and module header definitions for the destination and source models, i.e. client and server.
3. Module body definitions: each of these can contain the declaration part, the initialization part, and the transition part. There is one module body definition for each module, i.e. (1) SSYN_BODY, SACT1_BODY, SACT2_BODY, and SACT3_BODY for modules SSYN, SACT1, SACT2, and SACT3 respectively at the source site, (2) DSYN_BODY, DACT1_BODY, DACT2_ BODY, and DACT3_BODY for modules DSYN, DACT1, DACT2, and DACT3 respectively at the destination site, and (3) SERVER_BODY and CLIENT_BODY for the server module and the client module respectively. The declaration part declares some types, variables, etc. The declaration part of the SERVER_BODY (CLIENT_BODY) includes (1) the corresponding synchronizer and actors submodules (EFSMs) definitions at the source (destination) site, which have been described previously, and (2) the module variables definition of the SERVER_BODY (CLIENT_BODY), depicted in Table 1, column a (Table 1, column b). The initialization part sets up the initial status of the module body, including variables' initial values, the initial state, the channel links configuration, etc. The initialization part of the SERVER_BODY (CLIENT_BODY) is depicted in Table 1, column d (Table 1, column e). The associated transition part with different synchronization schemes of synchronizers and actors is presented in Appendices A, B, C and D and the previous subsections.
4. The whole specification module configuration definitions part for the specification module, i.e. the MMSynchronization module, defines module variables with the associated module definitions, sets up module variables with the associated module bodies, sets up the channel links configuration etc., as depicted in Table 1, column c.
5. Other type/constant definitions, function procedure definitions, etc.

5. Development of the M²EST system

Fig. 7 shows the abstract architecture of the M²EST-based multimedia synchronization specification and execution environment. There are four layers: (1) an M²EST editor in the application's user interface, (2) an M²EST compiler, (3) the embedded computer environment, (4) the underlined computer networks. Users of the associated multimedia applications can use the provided M²EST editor to specify the presentation schedule, the duration for each presentation stage, the transmission and presentation rates, the intra-medium and inter-media synchronization points, the coarse-grain and fine-grain synchronization schemes, and the associated parameters, e.g. the restricted blocking time interval etc.

The M²EST compiler is a suite that consists of an M²EST translator, which translates M²EST specifications to C codes, a linker, a loader, etc. The M²EST compiler mainly consists of a token scanner, a syntax parser and some code generator routines. The token scanner and the syntax parser are generated by using UNIX standard utilities LEX and YACC, respectively. LEX generates a token scanner by using a lexical specification as an input. YACC receives the M²EST grammar specification, which is the M²EST BNF. For each parsed grammar, there is a C-code generator that generates the corresponding C-code.

The M²EST compiler translates M²EST specifications to the target executable programming language, i.e. the C

Table 1
Some of the other specifications

(a)	(b)	(c)	(d)	(e)
modvar	**modvar**	**modvar**	**initialize**	**initialize**
SouS: SSYN;	DesS: DSYN;	D: client;	**begin**	**begin**
SouA1: SACT1;	DesA1: DACT1;	S: server;	**init** SouS **with** SSYNC_BODY;	**init** DesS **with** DSYNC_BODY;
SouA2: SACT2;	DesA2: DACT2;	**initialize**	**init** SouA1 **with** SACTI_BODY;	**init** DesA1 **with** DACT1_BODY;
SouA3: SACT3;	DesA3: DACT3;	**begin**	**init** SouA2 **with** SACT2_BODY;	**init** DesA2 **with** DACT2_BODY;
		init D **with** CLIENT_BODY;	**init** SouA3 **with** SACT3_BODY;	**init** DesA3 **with** DACT3_BODY;
		init S **with** SERVER_BODY;	**connect** SouS.SSIP1 **to** SouA1.SAIP1;	**connect** DesS.DSIP1 **to** DesA1.DAIP1;
		connect D.DSP1 **to** S.SSP1;	**connect** SouS.SSIP2 **to** SouA2.SAIP2;	**connect** DesS.DSIP2 **to** DesA2.DAIP2;
		connect D.DSP2 **to** S.SSP2;	**connect** SouS.SSIP3 **to** SouA3.SAIP3;	**connect** DesS.DSIP3 **to** DesA3.DAIP3;
		connect D.DSP3 **to** S.SSP3;	**attach** SSP1 **to** SouA1.SA1;	**attach** DSP1 **to** DesA1.DA1;
		end	**attach** SSP2 **to** SouA2.SA2;	**attach** DSP2 **to** DesA2.DA2;
			attach SSP3 **to** SouA3.SA3;	**attach** DSP3 **to** DesA3.DA3;
			end	**end**

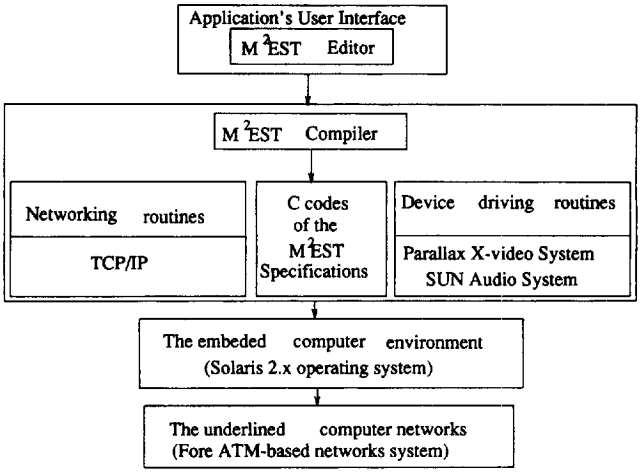


Fig. 7. The abstract architecture of M²EST-based multimedia synchronization specification and execution environment.

language currently, then links and loads the associated devices, networking etc., system calls to have the complete executable code. The M²EST translator generates (1) one program in the centralized case, or (2) two programs, one for the destination site and the other one for the source site, in the distributed case.

There are mainly three components in the generated executable C code. (1) The transition dispatcher: these select a transition to be executed according to the current state of the EFSM, the current values of variables, the head message of the input buffer, etc. The M²EST compiler generates one dispatcher for each EFSM module according to the EFSM table that is built after parsing an M²EST specification. (2) The transition objects: the M²EST compiler compiles each transition in an EFSM module to a condition object and an action object. The dispatcher can execute the selected action object according to the condition objects. (3) The networking interface processor: the processor contains the corresponding networking routines offered by the operating system and the networking interface hardware/software system.

The main components in code generator routines are depicted in Fig. 8. There are mainly three components: a declaration part processor, an EFSM translator, and an execution environment generator. The declaration part processor consists of class generator that decides which case, CENTRALIZED or DISTRIBUTED, is parsed, type/constant generator, channel generator, module generator, procedure/function generator, and state generator.

The channel generator constructs the channel information table. In an M²EST specification, abstract channels are transferred to two types of communication mechanisms. One is the global shared variables for intra-machine (intra-site) light-weight-process (LWP) communication channels and the other one is UNIX sockets for inter-machine (inter-site) communication channels. The channel generator mainly generates a channel information table that records the channel name, channel roles, and messages that the channel can exchange. The message exchange is achieved by an interaction point that represents one role of one channel. In processing the interaction point message exchange specification, the channel generator can find the channel name, the channel role, and messages that are allowed for the interaction point in the channel information table, and then check if the message exchange is allowed.

The EFSM translator translates transitions, generates the dispatcher C code, and adds the real-time control functions, e.g. functions to measure the computing overhead of T_c and T_s described in Section 3, to the C code. The EFSM translator consists of a transition dispatcher generator and a transition translator. The transition translator consists of a condition object generator, an action object generator, and a real-time control generator. These generators are described as follows:

- Condition object generator: a transition can consist of two components, the condition part and the action part. The condition part has the following clause: FROM clause, TO clause, PROVIDED clause, WHEN clause, PRIORITY clause, and DELAY clause. The TO clause and FROM clause are used to generate the EFSM table. The PROVIDED clause is translated to a subroutine to check the Boolean value of the Boolean expression. The WHEN clause is translated to a subroutine to check one specific message buffer. To process the PRIORITY clauses, each PRIORITY clause is translated to a corresponding priority number. A transition selection mechanism compares the priority numbers of executable transitions. When there are some executable transitions with the same highest priority number, the default transition is the firstly checked transition. For a DELAY clause, the compiler generates a timer for the transition.
- Action object generator: to process the action part of a transition, the main work is to translate the Pascal statements to the C code.
- Real-time control generator: the timer that is associated with a DELAY clause is added by the real-time control generator. The timer_create, the timer_settime, and the timer_delete functions are all generated by the real-time control generator which is supported by the real-time library. The timer_create function creates a local timer. Each timer is relative to a system clock as the timing base. The real-time control can send a signal, which is defined in the parameter of a timer_create function, to the processor when the timer expires. The timer_create CLOCK and event are translated to the clock 'CLOCK_REALTIME' and signal 'SIGALRM' respectively. The timer_settime function sets the time out interval for the timer. The timer used in the timer_settime function is created by the timer_create function, and the time interval defined in timer_settime is relative to the clock of the timer defined in timer_create. In addition to timer_create, timer_settime and timer_ delete functions, a real-time control also

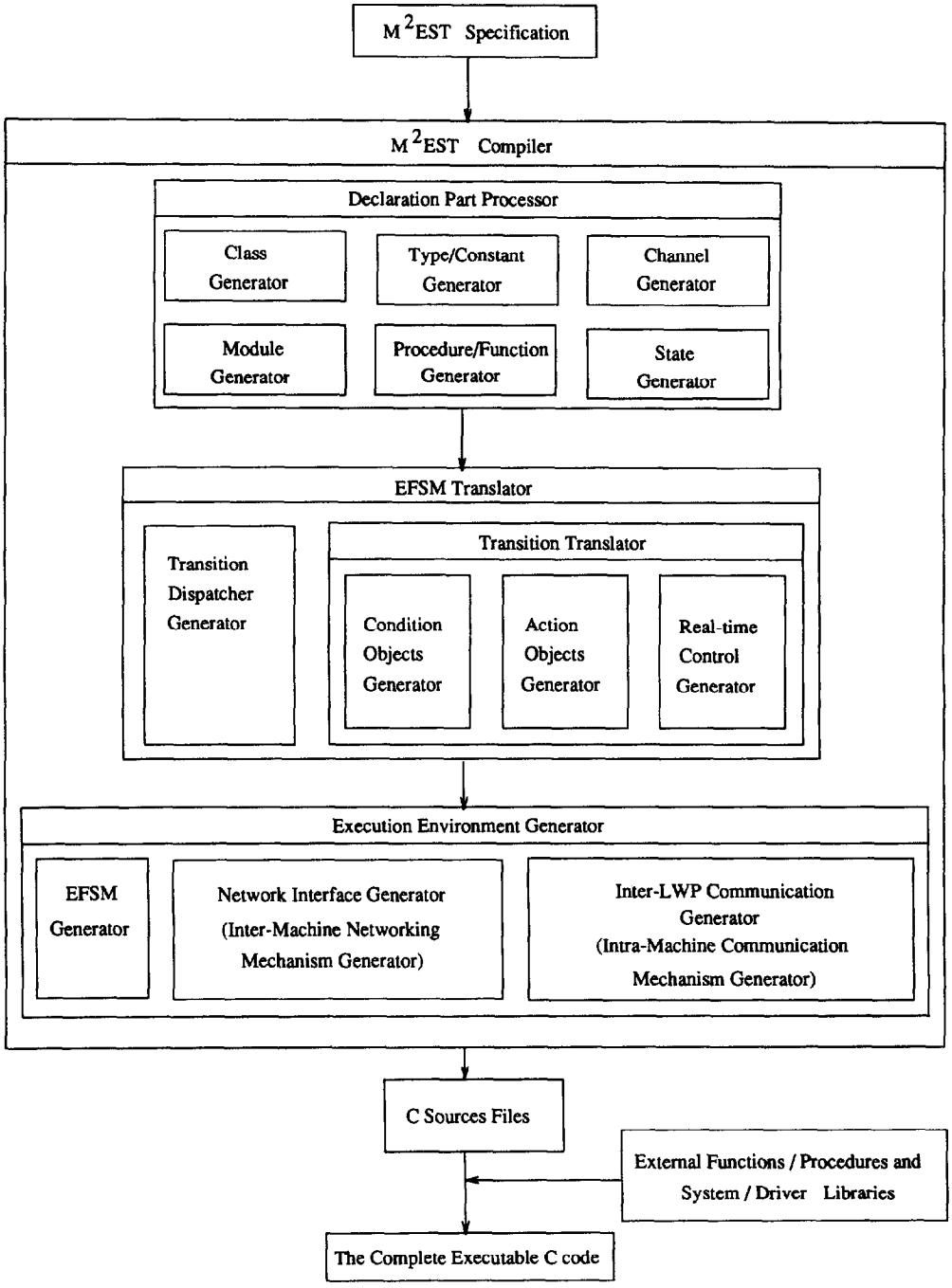


Fig. 8. The main components in the M²EST compiler.

includes a signal handler. The signal handler receives the signal sent from the timer_settime and then enables the delay transition. The timer_delete function removes a timer.

- Transition dispatcher generator: the transition dispatcher generator generates one dispatcher for each EFSM module according to the EFSM table. The EFSM table records the transitions that each state has, and the head state and the tail state of each transition. The transition dispatcher generator generates transition objects for each EFSM module. Each transition object includes a condition object and an action object. For each state of the module, there are always more than one transition object. The dispatcher executes these transition objects in parallel, that is, (1) checks the condition part of each transition object in parallel, then (2) selects one executable transition by a transition selection mechanism generated by the condition object generator.

The execution environment generator consists of an EFSM generator, a network interface generator and an

inter-LWP communication generator, as described below. The EFSM generator maps a module name to a template EFSM module. There are two communication types that are used in M²EST multimedia synchronization specifications, the inter-machine (inter-site) communication and the intra-machine (intra-site) communication. The network interface generator and the inter-LWP communication generator deal with the inter-machine communication and the intra-machine communication respectively.

- The EFSM generator: the EFSM generator generates executable EFSM modules that are defined in a multimedia synchronization specification. Each EFSM module is a template and different module variables can be mapped to the same EFSM module. The INIT statement of an EFSM maps a module name to an EFSM module. Since all EFSMs are considered to be executed in parallel, the M²EST compiler translates each executable EFSM module to an LWP, i.e. a thread.
- The network interface generator: the network interface generator deals with module connections in different sites. In the M²EST compiler, module connections in different sites are translated to socket specifications. The network interface generator also generates message handlers in socket specifications. A message handler puts the messages, which are received from sockets (interactionpoints), to the corresponding interaction points (sockets).
- The inter-LWP communication generator: the inter-LWP communication generator deals with module connections in a single site. In the M²EST compiler, module connections in a single site are translated to a shared global variables section.

In EFSM-based multimedia synchronization environments, all EFSMs are considered to be executed in parallel. The condition checks of the outgoing transitions of an EFSM's current state are also executed in parallel. After parsing an M²EST specification, the compiler translates each EFSIA module to an LWP, i.e. a thread, that is provided in Solaris 2.x. Each transition condition check is also mapped to a thread, such that all of them can be processed in parallel. Fig. 9 shows the execution structure of M²EST in the distributed environment. The currently underlined operating system is Solaris 2.x and the underlined networking environment is Fore's ATM-based networks.

6. Discussion and conclusion

M²EST is currently applied in an ISO Open Document Architecture (ODA) based distributed multimedia document system for the news on demand (NOD) application. The NOD system contains two main components: one is the M²EST compiler and the other one is the authoring and presentation system. The authoring component contains a window-based spatial specification editor (SSE) and a temporal specification editor (TSE). The SSE is in charge of specifying the spatial layout of the components in a multimedia document, and TSE, which is currently a text-based editor and to be a window-based editor for M²EST, is in charge of specifying the temporal relationships among the components in a multimedia document. The internal data structure is ODA-based, in which audio and video parts are defined by ourselves because the current ODA profile does not have audio and video definitions. The presentation component is based on SUN's window system, in which the video-related and image-related parts are based on the Parallax X-video system[4] and the audio-related part is based on the audio system embedded in SUN workstations. The presentation schedules are controlled by the generated C code of the corresponding M²EST specifications.

Formal and executable specification languages for multimedia synchronization are urgently required. The role of M²EST is as follows: an M²EST translator can be bound with multimedia operating systems, vendors' high-speed network interface cards, e.g. ATM-based network interface cards, or vendors' video/image capture/display cards, e.g. the Parallax video card. The networking interface processor in the M²EST system uses the service primitives (system calls) provided by the transport protocols, which are embedded in the corresponding operating systems or the corresponding high-speed network interface cards. The video/image capture/display part invokes the associated system calls provided by the corresponding operating systems or the corresponding video/image capture/display cards. Thus, multimedia application software, e.g. distributed multimedia document systems, video conferencing etc., can have M²EST to generate the presentation schedule controllers.

Many formal models have been proposed/discussed and related systems have been developed and evaluated in formal development of communication protocols. Estelle has been widely used in formal specifications of communication protocols. In the past decade, a lot of Estelle-based methodologies and tools have been developed [18–20]. Researchers and developers for multimedia systems can apply the corresponding practice and experience accordingly. That is, a lot of research results, e.g. theories, formal models, algorithms, and development methodologies, for +Estelle-based and/or EFSM-based specification, verification, implementation, and testing, can be directly or indirectly applied to distributed/networked multimedia computing.

[4] The media elements and files generated by the Parallax card contain some control information, e.g. the size of each video frame. At the source site, the generated audio part and the video part are separated, and control information is stripped off such that only the raw data is transmitted in the associated network channel. At the destination site, the size of each frame is measured and appended with the data, and then our system calls Parallax media display routines to play each medium unit.

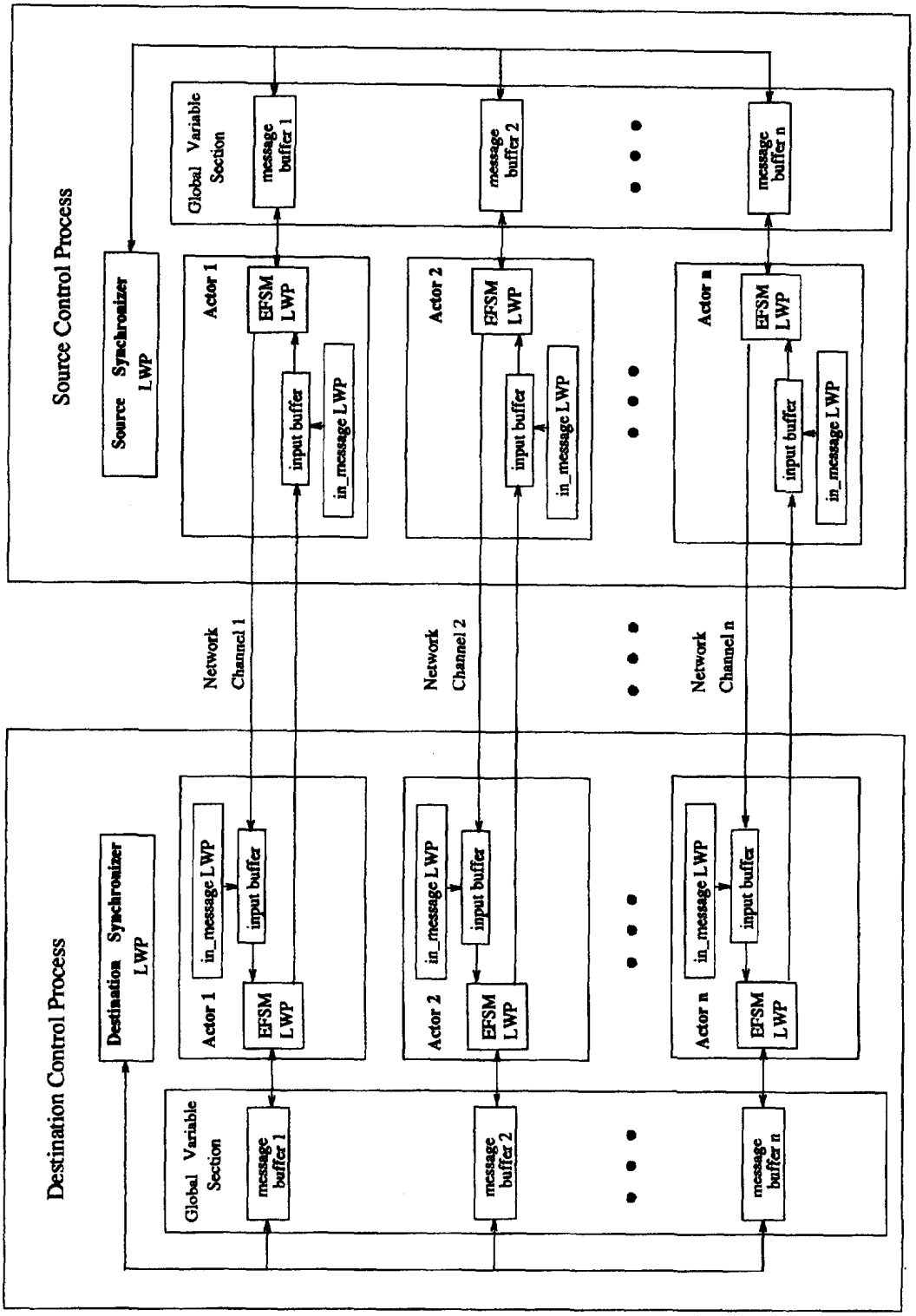


Fig. 9. The abstract execution structure of M²EST in the distributed environment.

The proposed M²EST language and the associated system is a typical example of applying the formal Estelle-based/EFSM-based protocol implementation approach to develop distributed multimedia synchronization systems. The other possible application is verification of synchronization specifications. A lot of non-timed and timed protocol verification methods/systems have been proposed/developed. Two possible approaches of applying Estelle-based/EFSM-based protocol verification methods/systems are as follows: (1) modifying currently existing protocol verification methods/systems according to the syntax/semantics of the formal models/languages that are used for specifying multimedia synchronization; (2) specifying multimedia synchronization based on the exact Estelle or the corresponding EFSM syntax/semantics. The second approach may not be good because the original purpose of Estelle's and EFSMs' language constructs was not designed for multimedia synchronization. The first approach may be a suitable approach. But some effort should be put into modifying the corresponding protocol verification methods/systems based on the modified language constructs.

The main characteristics of our M²EST approach are as follows.

1. A hybrid specification model is used, i.e. the control part is specified using some state-transitions, and the dynamic behaviors aspect of the control part is specified using Pascal-like programming-language-based statements.
2. A new synchronization control architecture is proposed, i.e. each medium is represented as an actor EFSM, and a synchronizer EFSM is used to control temporal relationships among media. That is, the synchronizer EFSM maintains the inter-media synchronization and the actor EFSM handles the intra-medium synchronization. The synchronization events are performed by message-passing between synchronizer and actors.
3. Synchronization schemes, e.g. (i) parallel-first, restricted parallel-first, and parallel-last, which are used to maintain inter-stream synchronization, and (ii) blocking, restrict-blocking, and non-blocking schemes, which are used to keep intra-stream synchronization, can also be formally specified to reduce the development cost.

The M²EST language and system are still under development. Our future work is to have new versions of M²EST that are able to achieve multimedia synchronization in the interactive presentation environment and the multicast/broadcast presentation environment.

Acknowledgements

This research is supported by the National Science Council of the Republic of China under grant NSC 86-2213-E-006-093.

Appendix A Transitions of the synchronizer EFSM at the destination site

```
{T1}
to S0
begin
output
DSIP1.ConReq;
output DSIP2.Con
Req;
output
DSIP3.ConReq;
end

{T2}
from S0
to S0
when DSIP1.ConConf;
begin
VIDplay: = T;
end

{T3}
from SO
to SO
when DSIP2.ConConf;
begin
AUDplay: = T;
end

{T4}
from S0
to S0
when DSIP3.ConConf;
begin
IMAplay: = T;
end

{T5}
from S0
to S1
provided (VIDplay =
T) and (AUDplay = T)
and (IMAplay = T)
begin
VIDstage: = 1;
output DSIP1.Act;
VIDplay: = F;
AUDstage: = 1;
output DSIP2.Act;
AUDplay: = F;
IMAstage: = 1;
output DSIP3.Act;
IMAplay: = F;
end
IMAplay: = F;

{T6,T9,T13,T16}
from Si
to Si
when DSIP1.Over(-
Vend)
begin
VIDplay: = T;
end
where i = 1,2,3,4

{T7,T10,T14,T17}
from Si
to Si
when
DSIP2.Over(Aend)
begin
AUDplay: = T;
end
where i = 1,2,3,4

{T8,T15}
from Si
to S(i + 1)
provided (VIDplay =
T)
and (AUDplay = T)
begin
VIDstage: =
VIDstage + 1;
output DSIP1.Act;
VIDplay: = F;
AUDstage: =
AUDstage + 1;
output DSIP2.Act;
AUDplay: = F;
end
where i = 1,3

{T11,T18}
from Si
to Si
when
DSIP3.Over(Iend)
begin
IMAplay = T;
end
where i = 2,4

{T12}
from S2
to S3
provided (VIDplay =
T) and (AUDplay = T)
and (IMAplay = T)
begin
VIDstage: =
VIDstage + 1;
output DSIP1.Act;
VIDplay: = F;
AUDstage: =
AUDstage + 1;
output DSIP2.Act;
AUDplay: = F;
IMAstage: =
IMAstage + 1;
output DSIP3.Act;
IMAplay: = F;
end

{T19}
from S4
to S5
provided (VIDplay =
T) and (AUDplay = T)
and (IMAplay = T)
begin
output DSIP1.DisReq;
VIDplay: = F;
output DSIP2.DisReq;
AUDplay: = F;
output DSIP3.DisReq;
IMAplay: = F;
end

{T20}
from S5
to S5
when DSIP1.DisConf
begin
VIDplay: = T
end

{T21}
from S5
to S5
when DSIP2.DisConf
begin
AUDplay: = T;
end

{T22}
from S5
to S5
when DSIP3.DisConf
begin
IMAplay: = T;
end

{T23}
from S5
to S6
provided (VIDplay =
T) and (AUDplay = T)
and (IMAplay = T)
begin
end
```

Appendix B Transitions of the actor EFSMs at the destination site

{T1}
from S1
to S1
when DAIP*i*.ConReq
begin
output DA*i*.ConReq;
end

{T2}
from S1
to S1
when DA*i*.ConInd
begin
output
DAIP*i*.ConConf;
stage: = 1;
end

{T3}
from S1
to S2
when DAIP*i*.Act
begin
counter: = 0;
end

{T4}
from S2
to S2
delay [T_m,T_M]
provided counter <
Length[stage]
when
DA*i*.MedTrans
(*Media*)
begin
play*X*(*Media*,position,-
width,height);
counter: = counter +
1;
end

{T5}
from S2
to S1
delay [T_m,T_M]
provided counter =
Length[stage]
begin
output
DAIP*i*.Over(stage);
stage: = stage + 1;
end

{T12}
from S1
to S1
when DAIP*i*.DisReq
begin
output DA*i*.DisReq;
end

{T13}
from S1
to S1
when DA*i*.DisInd
begin
output DAIP*i*.DisReq;
end

where *i* is (i) 1 for video actor, (ii) 2 for audio actor, and (iii) 3 for image actor; [T_m,T_M] is (i) [interval(stage,counter) − T_c,interval(stage,counter) − T_c] for video and audio actors, and (ii) [duration(stage,counter) − T_c,duration (stage,counter) − T_c] for image actor; (*X*,*Media*) is (i) (Video,VIDframe) for video actor, (ii) (Audio,AUDseg) for audio actor, and (iii) (Image,IMAdata) for image actor.

Appendix C Transitions of the actor EFSMs at the source site

{T1}
from S1
to S1
when SA*i*.ConReq
begin
output SA*i*.ConInd;
output SAIP*i*.ConInd;
stage: = 1;
end

{T2}
from S1
to S2
when SAIP*i*.Transmit
begin
counter: = 0;
end

{T3}
from S2
to S2
provided counter <
Length[stage]
delay[T_m,T_M]
begin
output
SA*i*.MedTrans(*Media*)
counter: = counter+1;
end

{T4}
from S2;
to S1
delay [T_m,T_M]
provided counter =
Length[stage]
begin
output
SAIP*i*.over(stage);
stage: = stage + 1;
end

{T5}
from S1
to S1
when SA*i*.DisReq
begin
when SA*i*.DisReq
end

{T6}
from S1
to S1
when SAIP*i*.DisResp
begin
output SA*i*.DisInd
end

where *i* is (i) 1 for video actor, (ii) 2 for audio actor, and (iii) 3 for image actor; [T_m,T_M] is [*T*(stage,counter) − T_s,*T* (stage,counter) − T_s]; *Media* is (i) VIDframe for video actor, (5) AUDseg for audio actor, and (iii) IMAdata for image actor.

Appendix D Transitions of the synchronizer EFSM at the source site

{T1}
from S0
to S0
when SSIP1.ConInd
begin
VIDplay: = T;
end

{T2}
from S0
to S0
when SSIP2.ConInd
begin
AUDplay: = T;
end

{T3}
from S0
to S0
when SSIP3.ConInd
begin
IMAplay: = T;
end

{T4}
from S0
to S1
provided
(VIDplay = T) **and**
(AUDplay = T) **and**
(IMAplay = T)
begin
VIDstage: = 1;
output
SSIP1.Transmit;
VIDplay: = F;
AUDstage: = 1;
output
SSIP2.Transmit;
AUDplay: = F;
IMAstage: = 1;
output
SSIP3.Transmit;
IMAplay: = F;
end

{T5,T8,T12,T15}
from S*i*
to S*i*
when SSIP1.Over(-
Vend)
begin
VIDplay: = T;
end
where *i* = 1,2,3,4

{T6,T9,T13,T16}
from S*i*
to S*i*
when
SSIP2.Over (Vend)
begin
AUDplay: = T;
end
where *i* = 1,2,3,4

{T7,T14}
from S*i*
to S(*i* + 1)
provided (VIDplay = T)
and (AUDplay = T)
begin
VIDstage: =
VIDstage + 1;
output
SSIP1.DisResp
VIDplay: = F;
AUDstage: =
AUDstage + 1;
output
SSIP2.Transmit;
AUDplay: = F;
end
where *i* = 1,3

{T10,TI7}
from S*i*
to S*i*
when
SSIP3.Over(Iend)
begin
IMAplay: = T;
end
where *i* = 2,4

{T11}
from S2
to S3
provided (VIDPlay =
T) **and** (AUDplay = T)
and (IMAplay = T)
begin
VIDstage: =
VIDstage + 1;
output
SSIP1.Transmit;
VIDplay: = F;
AUDstage: =
AUDstage + 1;
output
SSIP2.Transmit;

{TI8}
from S4
to S5
provided (VIDplay = T)
and (AUDplay = T) **and**
(IMAplay = T)
begin
VIDplay: = F;
AUDplay: = F;
IMAplay: = F;
end

{T19}
from S5
to S5
when SSIP1.DisInd
begin
VIDplay:T;
end

{T20}
from S5
to S5
when SSIP2.DisInd
begin
output SSIP2.DisResp
AUDplay: = T;
end

{T21}
from S5
to S5
when SSIP3.DisInd
begin
output SSIP3.DisResp;
IMAplay: = T;
end

{T22}
from S5
to S6
provided (VIDplay = T)
and (AUDplay = T) **and**
(IMAplay = T)
begin
end

```
AUDplay: = F;
IMAstage: =
IMAstage + 1;
output
SSIP3.Transmit;
IMAplay: = F;
end
```

References

[1] B. Furht, Multimedia systems: an overview, IEEE Multimedia 1 (1) (1994) 47–59.

[2] W.I. Grosky, Multimedia information systems, IEEE Multimedia 1 (1) (1994) 12–24.

[3] G. Blakowski, R. Steinmetz, A media synchronization survey: reference model specification, and case studies, IEEE J. on Selected Areas in Communications 14 (1) (1996) 5–35.

[4] R. Steinmetz, Synchronization properties in multimedia systems, IEEE J. on Selected Areas in Communications 8 (3) (1990) 401–412.

[5] R. Steinmetz, Human perception of jitter and media synchronization, IEEE J. on Selected Areas in Communications 14 (1) (1996) 61–72.

[6] M. Woo, N.U. Qazi, A. Ghafoor, A synchronization framework for communication of preorchestrated multimedia information, IEEE Network 8 (1) (1993) 147–155.

[7] M.C. Buchanan, P.T. Zellweger, Automatically generating consistent schedules for multimedia documents, ACM Multimedia Systems 1 (2) (1993) 55–67.

[8] R. Hamakawa, J. Rekimoto, Object composition and playback models for handling multimedia data, ACM Multimedia Systems 2 (1) (1994) 26–35.

[9] F. Horn, J.B. Stefani, On programming and supporting multimedia object synchronization, Computer J. 36 (1) (1993) 4–18.

[10] T.D.C. Little, A. Ghafoor, Synchronization and storage models for multimedia objects, IEEE J. on Selected Areas in Communications 8 (3) (1990) 413–427.

[11] T. Agerwala, Putting Petri nets to work, IEEE Computer 12 (12) (1979) 85–94.

[12] J.L. Peterson, Petri nets, Computing Survey 9 (3) (1977) 225–252.

[13] C.M. Huang, C.M. Lo, An EFSM-based multimedia synchronization model and the authoring system, IEEE J. on Selected Areas in Communications 14 (1) (1996) 138–152.

[14] S. Budkowski, P. Dembinski, An introduction to Estelle: a specification language for distributed systems, Computer Networks and ISDN Systems 14 (1987) 3–23.

[15] ISO Information Processing Systems, Open Systems Interconnection, 1987. Estelle: A Formal Description Technique Based on Extended State Transition Model, DIS.9074.

[16] C.M. Huang, C.H. Lin, Backus naur form of M^2EST. Technical Report MingSoft-TR-MM-I-96-6-1, Institute of Information, National Cheng Kung University, June 1996.

[17] C.M. Huang, R.Y. Lee, Quantification quality-of-presentation (QOP) for multimedia synchronization schemes, ACM Computer Communication Review 26 (3) (1996) 76–104.

[18] S. Budkowski, Estelle development toolset (EDT), Computer Networks and ISDN Systems 25 (1992) 63–82.

[19] C.M. Huang, J.M. Hsu, An incremental protocol verification method, Computer J. 37 (8) (1994) 698–710.

[20] C.M. Huang, S.W. Lee, Timed protocol verification for Estelle-specified protocols, ACM Computer Communication Review 25 (3) (1995) 4–32.