A Standard Model for Multimedia Synchronization: PREMO Synchronization Objects

Ivan Herman[1], **Nuno Correia**[2], **David A. Duce**[3], **David J. Duke**[4], **Graham J. Reynolds**[5], **James Van Loo**[6]

[1] Centrum voor Wiskunde en Informatica (CWI), Kruislaan 413, 1098 SJ Amsterdam, The Netherlands

[2] Instituto de Engenharia de Sistemas e Computadores (INESC), 9, Alves Redol, 1000 Lisboa, Portugal

[3] Rutherford Appleton Laboratory, Chilton, Didcot, Oxon OX11 0QX, United Kingdom

[4] University of York, Heslington, York YO1 5DD, United Kingdom

[5] Commonwealth Scientific and Industrial Research Organization (CSIRO), Division of Information Technology, GPO Box 664, Canberra ACT 2601, Australia

[6] SunSoft, MTV 10-228, 2550 Garcia Avenue, Mountain View, CA 94043-1100, United States of America

Abstract. This paper describes an event–based synchronization mechanism, which is at the core of the inter–media synchronization in the upcoming standard for Multimedia Presentation, PREMO. The synchronization mechanism of PREMO is a powerful tool, based on a small number of concepts, and on cooperation among active objects, and represents a synthesis of various synchronization models described in the literature. This model can serve as a basis for the implementation of complex synchronization patterns in multimedia presentations, both purely event–based, as well as time–based.

Key Words: PREMO, multimedia systems, active objects, standards, multimedia synchronization, inter–media synchronization

1 Introduction

1.1 Synchronization problems in multimedia

The term “multimedia” is frequently used, but rarely defined. It is perhaps difficult to pin down the essence of multimedia since the term appears in very different contexts, including non–technical ones. It is not the purpose of this article to enter this terminological debate; however, one generally accepted and important characterization of multimedia systems, applications, and programming environments, etc., is that they manage *continuous media* data. “This term refers to the temporal dimension of media, such as digital video and audio in that at the lowest level, the data are a sequence of samples — each with a time position. The timing constraints are enforced during playback or capture when the data are being viewed by humans.”[19] In some cases, such as animation and synthetic 3D sound, the samples may result from (sometimes complex) internal calculations (synthesis) whereas, in other cases, the samples are available through some data capture process.

Correspondence to: I. Herman
e–mail: ivan@cwi.nl, nmc@inesc.pt, dad@inf.rl.ac.uk, duke@minster.york.ac.uk, graham.reynolds@cbr.dit.csiro.au, james.vanloo@sun.com

Maintaining the presentation of a continuous media data stream at a sufficient rate and quality for human perception represents a significant challenge for multimedia systems, and may impose significant resource requirements on the multimedia computing environment. Aside from this inherent constraint (sometimes referred to as the problem of *intra–media synchronization*) a further difficulty arises from the fact that multimedia applications often wish to use *several* instances of continuous media data at the same time: an animation sequence with some accompanying sound, a video sequence with textual annotations, etc. The difficulty here is that not only should the individual media data be presented with an acceptable quality, but well–defined portions of the various media content should appear, at least from a perceptual point of view, simultaneously: some parts of a sound track belong to a specific animation sequence, subtitles should appear with specified frames in a video sequence, etc. This problem is usually referred to as *inter–media* synchronization. The specific problems raised by intra–media synchronization will not be addressed in this article; in what follows, the term synchronization is always used to refer to inter–media synchronization.

Synchronization has received significant attention in the multimedia literature, see, for example, the recent book by Gibbs and Tsichritzis[8] or the article of Koegel Buford[19] for further information and references on the topic. An efficient implementation of inter–media synchronization represents a major load on a multimedia system, and it is one of the major challenges in the field. What emerges from the experience of recent years is that, as is very often the case, one cannot pin down one specific place among all the computing layers (from hardware to the application) where the synchronization problem should be solved. Instead, the requirements of synchronization should be considered across all layers, i.e., in network technology, operating systems, software architectures, programming languages, etc. and user interfaces.

This article concentrates on one aspect of a complete solution, namely, on a conceptual model and software architecture aimed at inter–media synchronization. The object–oriented model is currently part of the PREMO specification[18], an ISO/IEC standard under development for multimedia programming. Being part of an upcoming ISO/IEC standard, the model represents a synthesis of the various synchronization techniques used in practice. It has also been inspired by developments carried out by some of the authors (I.H., G.J.R, N.C)

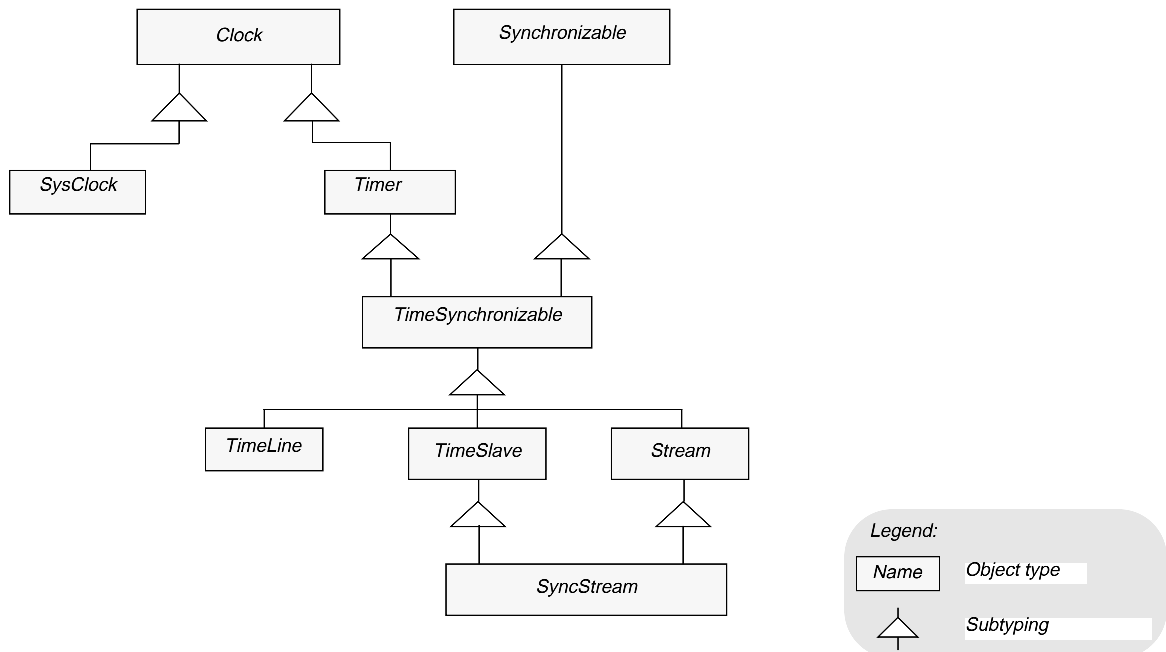

Fig. 1. Type hierarchy of synchronization objects in PREMO. The various object types are further described in the paper.

in the course of the MADE project[13], and by the specification of the Multimedia System Services, as defined by the International Multimedia Association[16,27]. A revised version of the Multimedia System Services is now an integral part of the PREMO document, and the PREMO model can be viewed as a superset of the so–called stream model defined in the IMA document. However, the original ideas have been revised by a number of independent experts before its incorporation as part of the PREMO specification. The synchronization model presented below relies on advanced technologies in networking and operating systems and an application making use of the model may have to build a more abstract layer on top of this basis, e.g., on some constraint–based systems or other forms of reasoning techniques. Fig. 1 gives a schematic overview of the various object types involved in the PREMO synchronization approach; further specialization of these objects leads to concrete media objects, such as video, audio, animated graphics, etc. However, this paper does not go into the details of these concrete media objects, and concentrates on the underlying synchronization paradigms only.

PREMO standardization is still at a development stage, hence a short overview of the main goals of this Standard are given below in Sect. 1.2. The details of the PREMO synchronization model are presented in Sects. 2, 3, and 4, with a short example in Sect. 5. Sect. 6 compares the PREMO model with some other synchronization mechanisms.

1.2 A short overview of PREMO

This section gives a very short overview of PREMO; for a more detailed presentation the interested reader should consult [12] or [14][1].

Today's application developers needing to realize high–level multimedia applications which go beyond the level of multimedia authoring do not have an easy task. There are only a few programming tools that allow an application developer the freedom to create multimedia effects based on a more general model than multimedia document paradigms, and these tools are usually platform specific (e.g., QuickTime[26] used as a programming interface). In any case, there is currently no available ISO/IEC standard encompassing these requirements. A standard in this area should focus primarily on the *presentation* aspects of multimedia, and much less on the coding, transfer, or hypermedia document aspects, which are covered by a number of other ISO/IEC or de–facto standards (for example, MHEG[17]). It should also concentrate on the *programming* tool side, and less on, e.g., the (multimedia) document format side. These are exactly the main concerns of PREMO.

[1] The reader may also refer to the current draft of the PREMO document itself, which is publicly available. The World Wide Web site "`http://www.cwi.nl/Premo/`" gives a good starting point to navigate through and access all available documents.

It is quite natural that the initiative for a standardization activity aiming at such a specification came from the group which has traditionally concentrated on presentation aspects over the past 15 years, namely ISO/IEC JTC1/SC24 (Computer Graphics). Indeed, this is the ISO subcommittee whose charter has been the development of computer graphics and image processing standards in the past. The Graphical Kernel System was the first standard for computer graphics published in this area; it was followed by a series of complementary standards, addressing different areas of computer graphics and image processing. Perhaps the best known of these are PHIGS, PHIGS PLUS, and IPS (see, e.g., Arnold and Duce[1] for an overview of all these Standards). The subcommittee has now turned its attention to presentation media in general as a way of augmenting traditional graphics applications with continuous media such as audio, video, or still image facilities, in an integrated manner. The need for a new generation of standards for computer graphics emerged in the past 4–5 years to answer the challenges raised by new graphics techniques and programming environments and it is extremely fortunate that the review process to develop this new generation of presentation environments coincided with the emergence of multimedia. In consequence, a synergistic effect can be capitalized on.

The JTC1 SC24 subcommittee recognised the need to develop such a new line of standards. It also recognised that any new presentation environment should include more general multimedia effects to encompass the needs of various application areas. To this end, a project was started in SC24 for a new standard called PREMO (Presentation Environment for Multimedia Objects) and is now a major ongoing activity in ISO/IEC JTC1 SC24 WG6. The subcommittee's goal is to reach the stage of a Draft International Standard in 1997.

The major features of PREMO can be briefly summarised as follows.

- *PREMO is a Presentation Environment.* PREMO, as well as the SC24 standards cited above, aims at providing a standard "programming" environment in a very general sense. The aim is to offer a standardized, hence conceptually portable, development environment that helps to promote portable multimedia applications. PREMO concentrates on the application program interface to "presentation techniques"; this is what primarily differentiates it from other multimedia standardization projects.
- *PREMO is aimed at a Multimedia presentation,* whereas earlier SC24 standards concentrated either on synthetic graphics or image processing systems. Multimedia is considered here in a very general sense; high–level virtual reality environments, which mix real–time 3D rendering techniques with sound, video, or even tactile feedback, and their effects, are, for example, within the scope of PREMO.
- *PREMO is Object Oriented.* This means that, through standard object–oriented techniques, a PREMO implementation becomes extensible and configurable. Object–oriented technology also provides a framework to describe distribution in a consistent manner.

A precise object model constitutes a major part of PREMO. The object model is fairly traditional, and is based on the concepts of subtyping and inheritance. It is also very pragmatic in the sense that it includes, for efficiency reasons, the notion of non–object (data) types, as is the case with a number of object–oriented languages, such as C++ or Java, and in contrast to "pure" object–oriented models, such as SmallTalk. The PREMO object model originates from the object model developed by the OMG consortium for distributed objects, but some aspects of the OMG model have been adapted to the needs of PREMO. A strong emphasis is placed in the model on the ability of objects to be active. This means that PREMO objects have, conceptually, their own thread of control; objects can communicate with one another through messages, i.e., through the operations defined on the object types. Objects can become suspended either by waiting for an operation invocation to return, or by waiting on the arrival of an operation request. Consequently, operations on objects serve as a vehicle to synchronize various activities (note that this concept of object synchronization is not the same as media synchronization although, of course, the concepts are related). Whether the concurrent activity of active objects is realized through separate hardware processors, through distribution over a network, or through some multithreaded operating system service, is oblivious to PREMO and is considered to be an implementation dependency.

The emphasis on the activity of objects stems primarily from the need for synchronization in multimedia environments and forms the basis of the synchronization model presented in this paper. Using concurrency to achieve synchronization in multimedia systems is not specific to PREMO. Other models and systems have taken a similar approach (see, for example, [5,8,13,21,24]) and PREMO, whose task is to provide a synthesis for standardization, has obviously been influenced by these models.

2 Event–based synchronization

As described above, the PREMO synchronization model is based on the fact that objects in PREMO are active. Different continuous media (e.g., a video sequence and corresponding sound track) are modelled as concurrent activities that may have to reach specific milestones at distinct and possibly user definable synchronization points. This is the *event–based* synchronization approach, which forms the basic layer of synchronization in PREMO. Although a large number of synchronization tasks are, in practice, related to synchroniza-

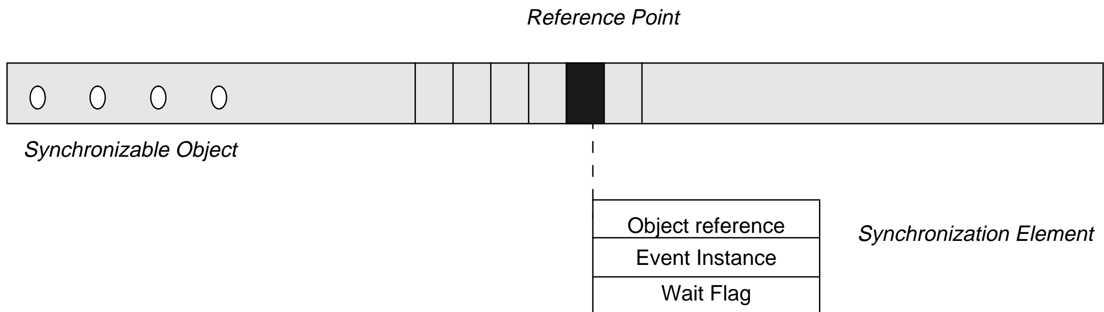

Fig. 2. A `Synchronizable` object

tion in time, the choice of an essentially "timeless" synchronization scheme as a basis offers greater flexibility. While time–related synchronization schemes can be built on top of an event–based synchronization model (see Sect. 3), it is sometimes necessary to support purely event–based synchronization to achieve special effects required by some application (see, for example, the application described in Sect. 5).

In line with the object–oriented approach of PREMO, the synchronization model defines abstract object types that capture the essential features of synchronization. For the event–based synchronization scheme two major object types are defined:

- *synchronizable objects*, which form the supertypes of, e.g., various media object types;
- *synchronization points*, which may be used to manage complex synchronization patterns among synchronizable objects.

These objects are described in somewhat more detail below.

2.1 Synchronizable objects in PREMO

Synchronizable objects in PREMO are autonomous objects, which have an internal progression along an internal one dimensional coordinate space. This space can be:

- extended real ($\Re_\infty$), or
- extended integer (Z_∞),

where "extension" means the inclusion of positive and negative "infinity" to the real and integer numbers, respectively. (The symbol *"C"* is used in this section to denote either an extended real or an extended integer.) The obvious extension of the notions "greater than", "smaller than", etc., on these types allows the behaviour of synchronizable objects to be defined more succinctly. Subtypes of synchronizable objects may add a semantic meaning to this coordinate space. For example, media objects may represent time, or video frame numbers along this space. Attributes of the progression, such as span (the relevant interval on this coordinate space), can be set through operations defined on these objects.

Reference points are points on the internal coordinate space of synchronizable objects where *synchronization elements* can be attached (see also Fig. 2). Synchronization elements contain information on an event instance (which is, essentially, a structure containing the object reference of the sender, a unique event type identity, and some event–dependent data), a reference to a PREMO object, a reference to one of the operations of this object, and, finally, a boolean `Wait` flag. When a reference point is reached, the synchronizable object makes a message call to the object stored in the reference point, using the operation reference to identify which operation it has to call, and using the event instance as an argument to the call. Finally, it may suspend itself if the `Wait` flag is set to *TRUE*. Through this mechanism, the synchronizable object can stop other objects, restart them, suspend them, etc. Operations are defined on synchronizable objects to add and delete reference points, and to add and delete synchronization elements associated with reference points.

In more precise terms, a `Synchronizable` object type is defined in PREMO as a supertype for all objects which may be subject to synchronization. This object is defined to be a finite state machine. The possible states, the major state transitions, and the operations resulting in state transitions, are shown in Fig. 3. The initial state is *STOPPED*. Note that no operation is defined for a transition into state *WAITING*; the only way a `Synchronizable` object can go into the *WAITING* state is through its internal processing cycle (see below).

If the object's state is *PLAY,* the object carries out its internal processing in a loop of processing stages. Each stage consists of the following steps:

1) The value of the current position is advanced using a (protected) operation `progressPosition` (defined as part of the object's specification) which returns the required next position.

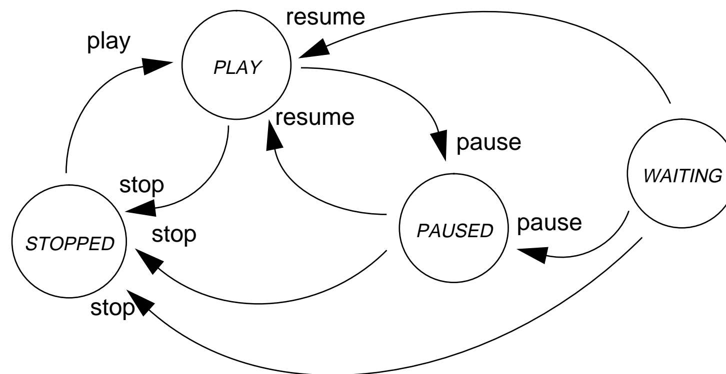

Fig. 3. `Synchronizable` object state machine

2) This required position is compared with the current position and the end position, and the following actions are performed:
   i) If there are reference points lying between the current position and the newly calculated position, then any associated synchronization actions are performed (in the order in which they are defined on *C*). This means:
      - perform data presentation for any data identified by the points on the progression space between the current position or the previous reference point and the next reference point or the end point;
      - invoke the operation, whose description is stored in the reference point, on the object whose reference is stored at the reference point, using the stored event as an argument;
      - if the `Wait` flag stored in the synchronization element belonging to the reference point is set to *TRUE*, the object's state is changed to *WAITING*. If the state of the object is set back, eventually, to *PLAY*, the stage continues at this point.
   ii) If the required position is smaller than the end position, then this becomes the local position and the processing stage is finished.

If the object is *PAUSED* or *WAITING*, then it can only react to a very restricted set of operation requests: the attributes of the object may be retrieved (but not set) and the `resume` or `stop` operations may be invoked, which may result in a change in state. The difference between *PAUSED* and *WAITING* is that, in the latter case, the object returns to the place where it had been suspended by a `Wait` flag, whereas, in the former case, a complete new processing stage begins. The differentiation between these two states, i.e., the usage of the `Wait` flag, is essential; this mechanism ensures an instantaneous control over the behaviour of the object at a synchronization point. If the object could only be stopped by another object via a `pause` call, an unwanted race condition could occur.

The progression of the object along its internal coordinate space happens within a (possibly infinite) interval of this space, called the *span*. Facilities are provided to modify the span, to ensure a cyclic behaviour, i.e., to return to the beginning of the span when the end is reached, to change the direction of the progression, etc.

Note that two aspects of this specification are left unspecified in the definition of `Synchronizable`:

- what "data presentation" means, and
- the semantics of the `progressPosition` operation.

Both these aspects should be specified in the appropriate subtypes of `Synchronizable`. The abstract specification of a synchronizable object is such that no media specific semantics are directly attached to it. Subtypes, realizing specific media control should, through specialization, attach semantics to the object through their choice of the type of the internal coordinate system, through a proper specification of what data presentation means, and through a proper specification of the `progressPosition` operation. The latter defines what it really means to "advance" along the internal coordinate system. For example, this progression may mean the generation of the next animation frame, decoding the next video frame, advance in time, etc.

The complete and detailed specification of a synchronizable object is too complex to be fully reproduced in this paper. The reader should refer to Part 2 of the PREMO document itself[18] for further details; note that a more formal specification of the object's behaviour, using Object–Z[7], is also available[6].

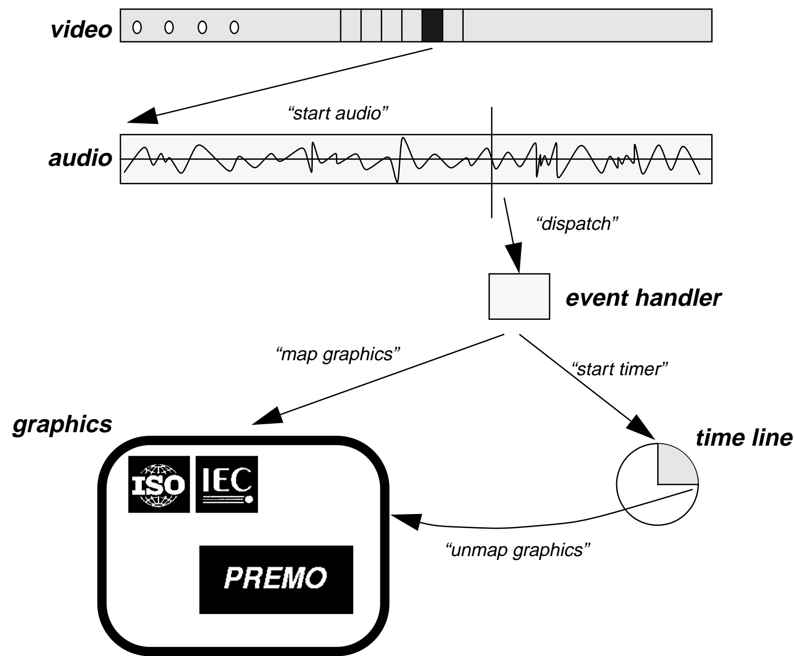

Fig. 4. An example for synchronization

The "target" object in the synchronization element (i.e., the object which has to be notified that the synchronizable object has reached a reference point) can be any PREMO object, provided some simple restrictions on the signature of the target operation are fulfilled. PREMO offers several different types of objects which can reasonably be used as targets in a synchronization scheme, e.g.,

- so–called "controller" objects, which are essentially finite state machines;
- event handler objects, which can dispatch events among several registered targets;
- other synchronizable objects.

These alternatives, with some more examples, will be elaborated in further detail in Sect. 2.2.

Fig. 4 shows a very simple example using the synchronization mechanism described above. The three media objects (video, audio, and graphics) in the figure are subtypes of `Synchronizable`, as is the time line object. They all add specific semantics to this supertype. Reference points and synchronization points are set up for the objects; the name of the operation referred to in synchronization elements are denoted on the figure. The effective synchronization pattern is:

1) The video starts to play; when it reaches its reference point, it sends a message to the audio object. The video object then continues to progress.
2) As a result of the message received from the video object, the audio object begins to play (in parallel with the video object). When it reaches its reference point, it sends a message to both the graphics and the time line objects (the role of the event handler object is to dispatch the same event among several targets). The audio object then continues to progress.
3) The graphics appears on the screen and, in parallel, a timer begins to tick. The timer has its own reference point set to, e.g., 15 seconds; when this reference point is reached, the message "unmap graphics" is sent to the graphics media object, which unmaps the image from the screen.

Although this example is obviously a simplified one, it illustrates the main mechanism at work when event–based synchronization is used.

2.2 Synchronization points

The simplicity of the example in Fig. 4 is partly due to the fact that there are only a few "interactions" among the participating media objects. The video object starts up the audio, which starts up the graphics and the timer but, once the start up actions have been performed, both the video and audio are free to continue their own activity independently from whatever happens to the other objects.

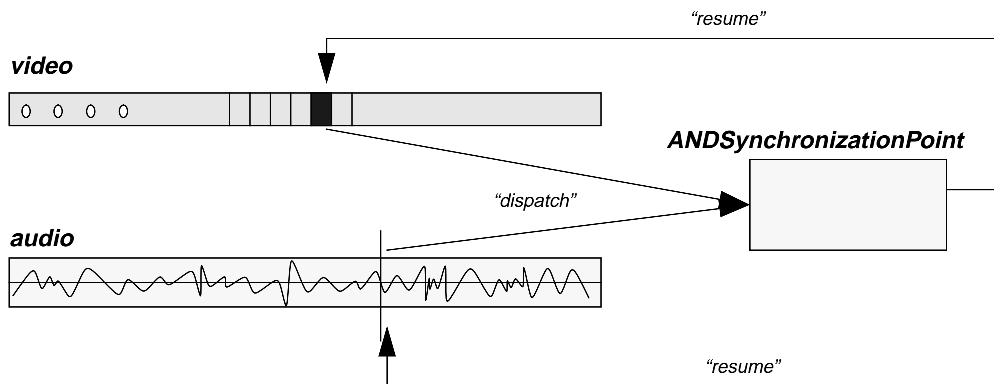

Fig. 5. Another example for synchronization using a synchronization point

Clearly, most applications have more complex requirements; mechanisms for "feedback" and mutual synchronization are also necessary. This kind of control is delegated in PREMO to the objects which are the targets of the message invocations at each reference point.

As mentioned earlier, the PREMO specification does not impose any significant restriction on what this target object may be. PREMO includes the specification of a number of other objects, defined independently of the synchronization mechanism, which are useful in combination with synchronization objects. These are:

- *Controller Objects.* A controller object is an autonomous finite state machine (FSM). State transitions are triggered by a special operation invoked by other objects. The actions related to a specific state of a controller object may cause messages to be sent to other PREMO objects, including controller objects, permitting a hierarchy of controllers to be defined. Subtypes of a controller can be defined which either have a specific state transition pattern coded into them or which allow the end user to freely program the FSM (e.g., through some script language). Complex synchronization patterns can be modelled through an FSM, which then becomes the main focal point of a specific synchronization scheme.
- *Event Handler Objects.* These objects provide control over event propagation. Events in PREMO are structures, containing a name together with event data, including a reference to the source of the event. The essential service provided by event handlers is the separation between the source of the events (e.g., a mouse, some external hardware, or, in the case of synchronization, a synchronizable object), and the recipient of these events. Sources broadcast events without having any knowledge of which objects will eventually receive them; this is achieved by forwarding the event instance to the event handler objects. Prospective recipients of events register themselves with these event handler objects, placing a request based on the event type. The recipients are then notified by the event handler on the arrival of an event, together with information about the event source. This event handling model is compatible with (and has drawn upon) the event service specification of OMG[23].

Fig. 4 already shows a typical, albeit simple use of an event handler: events are multiplexed to several event recipients at the same reference point of a synchronizable object.

PREMO also contains object types that have been added with the requirements of synchronization in mind (although they could be used for other purposes, too). These are all subtypes of event handlers, abstracting some common requirements of synchronization. Note that further work in this respect is still going on in the PREMO group and additional object types may be added to the Standard. The current object types are:

- *Synchronization Point.* Event handler objects do not impose any general constraints on the dispatched events, i.e., all incoming events are automatically broadcast to the registered event recipients. Synchronization points are special subtypes of event handlers, which can be used, for example, to restrict the object instances which can broadcast events. In other words, event handlers restrict the destination of events, synchronization points can, in addition, restrict the source. To achieve this additional functionality, synchronization points maintain a separate, internal set of events registered through special operations; events are dispatched if and only if they have been previously registered in this set. (Events include information on their sources.)
- *AND Synchronization Point.* A further specialization is offered by the `ANDSynchronizationPoint` object

(defined as a subtype of a Synchronization Point) which redefines the behaviour of event dispatching. In an `ANDSynchronizationPoint` object events are not automatically forwarded to event recipients; instead, the arrival of an event is recorded using a boolean flag associated with each element of the set of registered events. Only if all events, registered in this object, have this flag set to *TRUE,* are the event recipients notified.

Fig. 5 shows another simple example of synchronization using the AND synchronization point. The goal of the example is to fully synchronize both the video and audio objects at a specific point, i.e., the objects should continue their respective progression if and only if both objects have reached the specified reference point. This kind of requirement is very typical when, for example, audio and video samples are to be presented together. The synchronization pattern in the example is as follows:

1) The corresponding reference points of both media objects are connected to an instance of an AND synchronization point. The `Wait` flags in both synchronization elements are set to *TRUE*, and the `resume` operations of the media objects are registered as event recipients with the synchronization point.
2) When a media object reaches its reference point, it dispatches an event to the synchronization point and, because the `Wait` flag is true, it then performs a state transition to *WAITING* state, i.e., it suspends its own progression.
3) When both media objects have reached their reference points, the synchronization point sends a `resume` message to *both* media objects; consequently, both objects perform a state transition to *PLAY* and they are then able to continue displaying subsequent video and audio samples, respectively.

Of course, in a more realistic situation many reference points may be set on the media objects to achieve the desired effects; however, the synchronization mechanism is identical.

3 Time and synchronization

The synchronization model presented in Sect. 2 is event–based, i.e., the notion of time is not part of that abstraction level in the model. Clearly, applications also require a more elaborate version, which would allow them to reason with time. This is achieved in PREMO through the specification of a separate hierarchy of clock objects and the specialization of the basic synchronizable object to include the notion of time.

3.1 Clock objects

The abstract `Clock` object type provides PREMO with an interface to whatever notion of time is supported by its environment. This type assumes the existence of two non–object types: `Time`, to measure elapsed ticks (realized, for example, as a 64 bit integer), and `TimeUnit`, which defines the unit represented by each clock tick, for example an hour or a micro–second. The clock object type supports an operation, `inquireTick`, to measure the time elapsed since a specific moment. Subtypes of the clock object attach a more precise semantics to what kind of value this operation returns. Operations are also defined to measure the accuracy of the clock.

PREMO defines two specific subtypes of clock objects, which are as follows.

- *System Clock objects.* `SysClock` is a subtype of `Clock`, and provides real–time information (modulo the accuracy of the clock) to PREMO systems. `SysClock` does not add any new operations to `Clock`, but attaches a final semantics to the operation `inquireTick`, which is defined to return the number of ticks that have occurred since 00:00am, 1st January 1995, UTC.
- *Timer objects.* A `Timer` is a subtype of `Clock`, and provides facilities modelled after a stop–watch. It has operations to start, stop, pause, and reset the clock; all these operations are formally defined as state transition operations of a finite state machine containing the states *RUNNING*, *PAUSED*, and *STOPPED*. The operation `inquireTick` is defined to return the elapsed time the object spent in the *RUNNING* state since the last reset, without counting any time spent in the *PAUSED* state.

3.2 Time synchronizable objects

A `TimeSynchronizable` object type is a `Synchronizable` object type enriched with a `Timer` interface through multiple subtyping. Multiple subtyping means that the behaviours of `Timer` and `Synchronizable` objects are merged. This merge has several aspects, and introduces some new attributes and operations on `TimeSynchronizable`. These aspects are as follows.

1) Both the `Timer` and the `Synchronizable` object type are defined in terms of finite state machines. In `TimeSynchronizable`, these finite state machines are merged, i.e., the *RUNNING* state of the `Synchronizable` "part" is merged with the *RUNNING* state of the `Timer` "part", etc. The result is that the finite state machine governing `TimeSynchronizable` has the same states as `Synchronizable`, but the semantics of each of these states includes the semantics of both the `Timer` and the `Synchronizable`. Also, all state transi-

tion operations defined both in Timer and in Synchronizable can be used to induce the appropriate state transitions.

2) An attribute is defined, conveniently called speed, which relates progress through the progression space, inherited from Synchronizable, with time as measured by the Timer. The value of speed defines the number of units (e.g., number of frames) that the object will progress through in one tick. By default, this value may be set by the application, which can therefore exhibit control, e.g., over the playback speed. Various subtypes of TimeSynchronizable objects may restrict their behaviour so that the speed becomes read–only.

3) Synchronizable has a number of attributes and operations to set/retrieve reference points, set/retrieve minimum and maximum positions, etc. which are expressed in terms of the native progression space. When using TimeSynchronizable the client may want to use the abstraction offered by the notion of relative time, i.e., the time possibly returned by an inquireTick operation call. For this purpose, the reset operation (inherited from Timer) is redefined in TimeSynchronizable to, conceptually, put a marker against the current position on the native progression space as well as to reset the time register. This marked position on the progression space will serve as the zero point for relative positioning expressed by time values. The various operations on setting/retrieving reference points are conceptually overloaded, i.e., the client may also set these values using relative time as arguments, and the object will internally transform these values to the progression space.

The TimeSynchronizable object has all the usual synchronization features attached by various multimedia systems to their basic media representation. However, in most of the systems, the distinction between (relative) time and the internal progression space (e.g., video frames) is blurred, usually in favour of time only. PREMO maintains this dual nature of media data, and leaves it to applications to decide which aspect of media behaviour is more relevant in a concrete synchronization setting. This separation is one of the advantages of a clear object oriented specification offered by a standard such as PREMO.

3.3 Time slave objects

TimeSynchronizable objects are appropriate for creating complex synchronization patterns involving time. In an ideal world, where all local timers would represent an absolutely precise real time, this would be enough. However, multimedia systems rarely operate in an ideal world, and in practice all local timers will have a slightly different speed, accuracy, etc. Hence the necessity of implementing mechanisms which may monitor possible discrepancies.

PREMO does not aim at offering a full solution for this problem, because the necessary reactions, the tolerated discrepancies, etc., are usually application dependent. PREMO defines the basic mechanism which allows applications to implement a specific behaviour, and it does this in terms of a new object type, called a TimeSlave object. What this object essentially does is to control its own behaviour in terms of the timer data of another TimeSynchronizable object.

In more precise terms, a TimeSlave object is a subtype of TimeSynchronizable which permits synchronization over multiple TimeSynchronizable object instances. A *master* object can be attached to a TimeSlave object, and the latter will attempt to synchronize its progression with its master. This means the following:

1) The speed value of the TimeSlave object (relating the progress through progression space with time ticks) is measured in terms of the ticks as returned by the master. This also means that if the client changes the way the master Timer operates (i.e, changing the ticks) this will influence all TimeSlave objects attached to the same master.

2) The TimeSlave object measures the alignment between its own Timer values and the one of the master. A client of the TimeSlave object may either inquire the alignment, or attach EventHandler-s to various thresholds values. The TimeSlave object will raise specific events if the alignment between the master and the slave time values exceeds the threshold.

In order to calculate the possible alignment between the master and the slave time values, the reset operation of TimeSlave also stores all necessary information on the master clock (current value of tick, accuracy, units of measurement). Alignment values are always referred to in the units of TimeSlave. Using these terms, the alignment value is:

$$|Tick_{slave} - g(Tick_{master})|$$

where *g()* is a function which transforms the ticks of the master into the units of the slave, and takes into account the tick value of the master when the reset operation has been invoked on TimeSlave.

3.4 Time lines

The TimeLine object of PREMO does not add any significantly new feature to PREMO, but is a good example of how the abstractions of the various objects may be used to derive a specific, and useful object type. A TimeLine object is de-

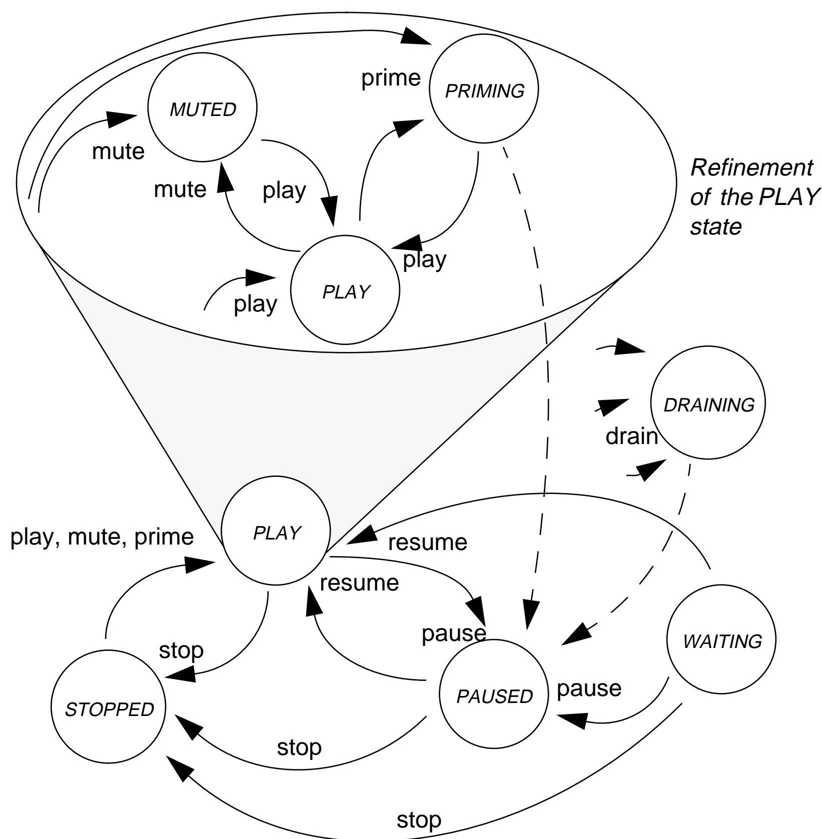

Fig. 6. State transition diagram for a *Stream* object

fined as a subtype of `TimeSynchronizable`, where the progression space is defined to be an integer large enough to represent time, and the value of `speed` is set to be of constant value 1. This object can be used to send events at predefined moments in time to dedicated PREMO objects, and may thereby serve as a basic tool for time–based synchronization patterns.

4 Streams

`Stream` objects represent a further step in specialization toward a possible representation of concrete media objects. As suggested by their name, these objects are closely related to the progression of media data where various media streams are connected in a dataflow–like network. Although this is not the only possible way of managing media control, this model of multimedia processing is very widespread in practice. For example, the Multimedia System Services of IMA[16,27] gives a framework for such dataflow–like multimedia processing and, in fact, most of the content of this section originates from the adaptation of the original IMA document into a part of PREMO.

In a dataflow–like network of streams one can refer to the "source" and the "destination" of media data, and the progression of such data also involves some form of buffering. Hence one of the difficulties of this model: applications are supposed to have control over buffering, because the way buffering is done may affect an application as a whole.

The `Stream` type of PREMO, and its subtypes, provide a single point of focus for all inquiry and control of media stream progress in a media type independent way. The `Stream` object is defined as a subtype of `TimeSynchronizable` and, as such, adds a finer media control to its supertype. This finer control is achieved through the refinement of the finite state machine governing the behaviour of `TimeSynchronizable`.

The `Stream` object adds three new states to the state machine of `TimeSynchronizable`, namely *MUTED*, *PRIMING*, and *DRAINING.* Three new operations are also defined, which control the state transitions to and from these new states: `mute`, `prime`, and `drain` (see also Fig. 6).

MUTED and *PRIMING* are refinements of the *PLAY* state of `TimeSynchronizable`. The additional semantics in these states is related to the notion of data presentation. As emphasized in Sect. 2.1, the specification of `Synchronizable` re-

fers to data presentation in very general terms only, as one abstract processing step of the object at that level of abstraction. The specification leaves the semantic details of what presentation means to the various subtypes of `Synchronizable`. Although the `Stream` object does not specify what presentation means either (and leaves the details to the subtypes of `Stream`), the specification of *MUTED* and *PRIMING* gives a somewhat finer control on the behaviour of the `Stream` object with respect of presentation. This refinement is as follows:

- *MUTED*: no presentation occurs while the object is in this state, and media data are discarded. In other words, progression on the stream occurs (i.e., all synchronization actions are performed) without any presentation effects.
- *PRIMING*: no presentation occurs while the object is in this state, and the media data are buffered in an internal buffer. In other words, progression on the stream occurs (i.e., all synchronization actions are performed) and the media data are stored internally instead of being presented. If the internal buffer of the object becomes full, i.e., no stream data can be stored any more, the object makes an internal state transition to *PAUSED*.

The third additional state, *DRAINING*, is the counterpart of *PRIMING* in buffer control. When set to this state, the object empties the buffer filled up by a previous *PRIMING* state; when the buffer is empty, the object makes an internal state transition to *PAUSED*. The operation `drain` is defined to set the `Stream` object into *DRAINING* state; although not clearly stated on Fig. 6, this state transition operation can be issued in any state, except *STOPPED*.

Subtypes of `Stream` may add additional semantics to buffer control. As a typical case, if the streams are aware of their positions within a dataflow network, some of the operations, like `prime` or `drain`, may also generate private control flow among the streams in this network. For example, `prime` on a `Stream` may also generate a control information to the `Stream` object "up–stream", i.e., the stream providing the data. Whether such additional protocol is defined or not depends on the subtypes of the `Stream` object and is currently not standardized by PREMO.

Finally, the `SyncStream` type is designed to permit the synchronization of multiple media streams. The client specifies a second `Stream` object to provide a master position reference to the `SyncStream`. This functionality is achieved by inheriting the behaviour of the `TimeSlave` objects. `SyncStream` is indeed defined as a (multiple) subtype of both the `Stream` and the `TimeSlave` object types, thereby refining the finite state machine of a `TimeSlave` object the same way as `Stream` objects refine the behaviour of `TimeSynchronizable` objects.

5 An example of event based synchronization usage

There is a large literature, as well as application programs, which describe and use various synchronization processes involving time, time discrepancies, sophisticated time control, etc. Some classifications of these, and their relations to the PREMO model, is described in Sect. 6 below. However, applications requiring a purely event–based synchronization mechanism are less frequent, and hence less well–known. This section gives a very short overview of one such application, which requires an event based synchronization mechanism like the one described by PREMO.

The application involves so–called *cineloops*, which are a movie–like representation of a sequence of ultrasound scans made for medical purposes. Details of how these cineloops are created are not of real interest here; they should be considered as special media objects which behave much like a sequence of images. Fig. 7 shows a screen snapshot involving two cineloops, taken by scanning the human heart. When a cineloop is recorded, one will usually also record an ECG trace (an electrocardiogram), shown below the cineloop images.

In many cases, it is useful to compare cineloops recorded under different conditions or at different times. For example, a stress test compares the movement of the heart wall after a rest and just after the person has exercised, when the heart rate is much higher. Fig. 7 shows two such recordings, each with its corresponding ECG trace.

The difficulty of playback, when these cineloops are used in a medical application, is that physicians want to see side by side a particular event of the heart beat, e.g., the start or the end of a contraction of a heart chamber. They are not really interested in the timing of the movements. However, the different phases of movement that constitute a heartbeat do not speed up with the same rate when the heart beats faster, i.e., it is not sufficient to just speed up or slow down the cineloops. In other words, synchronizing between the two cineloops cannot be done in terms of time; indeed, the notion of time does not really make any sense in this particular case.

The synchronization problem can be solved if event–based approach is used. Synchronization elements are set against reference points representing the required events in the cineloop, and the playback can be synchronized within a framework similar to that used in Fig. 5. Details of how this can be done can be found in the paper of Lie and Correia[20], which uses the MADE toolkit[13] for this purpose in a real–life medical application involving such cineloops (this toolkit uses a very similar synchronization mechanism to the one proposed by PREMO).

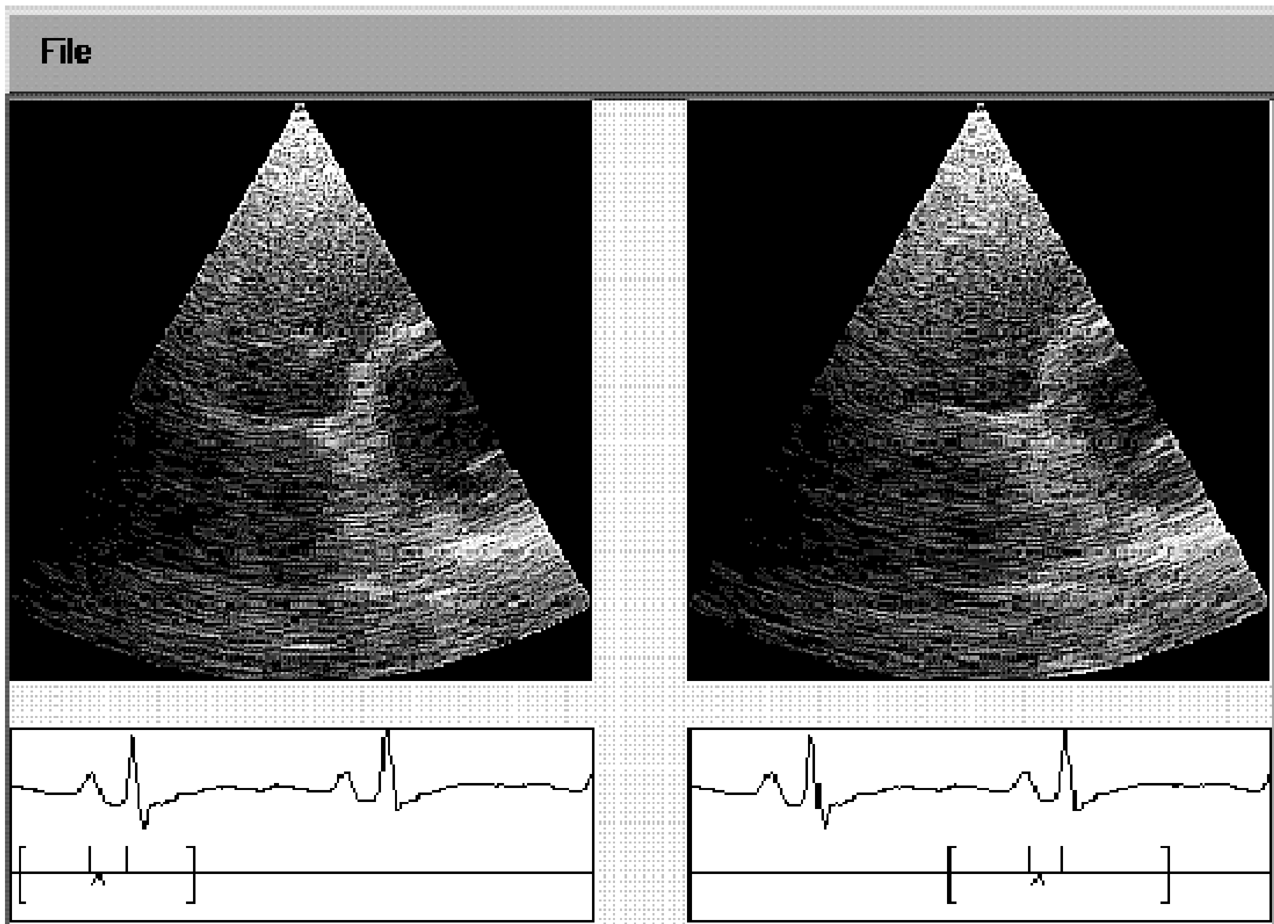

Fig. 7. Example for an event–based synchronization application

6 Comparison with other approaches

This section gives a review of some characterisations of systems and models that provide mechanisms for multimedia synchronization and discusses their relationship to the PREMO model. To begin with, however, we describe some generic definitions and requirements for multimedia synchronization and evaluate the PREMO synchronization objects against these.

The MHEG Standard[17,22], a standard for multimedia interchange, defines the following requirements for multimedia synchronization:

- *Elementary Synchronization*: Synchronization of two objects, either both with regard to the same reference origin time, or one with regard to the other.
- *Chained Synchronization*: Presentation of a set of objects one after the other in the form of a chain.
- *Cyclic Synchronization*: Repetitive presentation of one or more objects.
- *Conditional Synchronization*: Presentation of an object linked to the satisfaction of a condition.

These requirements give a good characterization of the synchronization mechanisms offered by a number of multimedia systems.

All of these requirements can be satisfied within the current synchronization model of PREMO, i.e., the model offers an appropriate framework to describe and/or to implement all these schemas. Elementary synchronization can be achieved directly using the synchronization element mechanism by relating the start or any other event in one object with the activation of the other. Chained synchronization can be achieved by setting synchronization elements such that stopping one object would start another. Cyclic synchronization for one object is supported directly in the `Synchronizable` object type by the

attribute repeatFlag. Cyclic synchronization in multiple objects can be achieved by proper setting of synchronization elements at the end of the last object. Finally, conditional synchronization can be supported either directly by the basic synchronization model (when one object reaches a point in its internal coordinate space the condition is true and an action is invoked), or by the synchronization point mechanism combining several conditions into a more complex one. Some of this functionality could also be achieved by manager objects with specific semantics (see below) that would program the temporal and reference point behaviour of the objects they manage.

In all these examples, clients have the choice to set the synchronization data either using the native progression space of the object (i.e., stay on the Synchronizable level) or to use the relative, inherent time values of the objects (i.e., using the abstraction offered by TimeSynchronizable).

Another classification of multimedia synchronization, which focuses on the way synchronization is modelled and implemented, is presented in Blakowski et al.[4]. Some of the functional requirements above can be realized in more than one of the synchronization categories below.

- *Hierarchical synchronization*: Multimedia objects are regarded as a tree consisting of nodes denoting serial or parallel presentation.
- *Synchronization on a time axis*: Single–media objects are attached to a time axis that represents an abstraction of time.
- *Synchronization at reference points*: Single–media presentations are composed of subunits presented at periodic times. The position of a subunit in an object is called a reference point.

Synchronization by means of reference points is the most flexible approach, allowing objects to be synchronized not only at the beginning or end of presentations, but also during a presentation. It is also well–behaved when objects have unpredictable duration.

The PREMO synchronization model is based on the reference point synchronization model, and other synchronization mechanisms, such as time–axis and hierarchical, are built on top of the fundamental objects and concepts. The Synchronizable object implements reference point synchronization and the TimeSynchronizable object adds time–axis behaviour. Hierarchical synchronization requires the use of appropriate Controller objects whose purpose is to manage several simple or mono–media objects. These objects program the reference points and synchronization elements and any time related behaviour, such as duration, start time, in the objects they manage. Currently, such special Controller objects should be created by the application, although later revisions of PREMO may also introduce such specialized "utility" objects as part of the Standard.

In Gibbs et al.[8], an object–oriented framework for composite multimedia is described. Composite multimedia is constructed from multimedia primitives and temporal transformations. The Gibbs system is also based on the concept of active objects. A set of object classes ("multimedia primitives") provide access to several kinds of media (e.g., CD audio, MIDI). Multimedia object classes inherit from ActiveObject and MultimediaObject. The methods of ActiveObject provide activity control for a multimedia object, such as start, stop. The methods of MultimediaObject deal with temporal coordinates, composition and synchronization. These methods make use of two temporal coordinate systems: world time and object time. The origin and units of world time are set by the application; object time is relative to a multimedia object. Each object can specify the origin, units and orientation of object time. The methods of MultimediaObject that affect the order and timing of the presentation of a multimedia object are called temporal transformations (e.g. Scale scales the duration of an object).

Some of the concepts in this model are present in the Synchronizable and TimeSynchronizable objects of PREMO, including the methods to control activity (start, stop, pause, resume) and the time scales. However, the PREMO model provides a more general framework, since the notion of time is not introduced in the base class, thereby allowing for different coordinate spaces. Also, the notification between objects supported by the reference points/synchronization elements mechanism of PREMO allows for complex synchronization patterns and not only for start/stop relationships.

QuickTime[9,26] is an extension to the Apple Macintosh System 7.0 operating system for multimedia application support. This extension comprises system software, file formats, compression management, and human interface standards. The Movie format is used to manage different forms of dynamic data. QuickTime uses a track model for organizing temporally related data of a movie. Tracks are time–ordered sequences of media types and they begin and end at different times during a presentation. Each track has its own time scale as its own coordinates in the screen. Tracks of the same or different media can be grouped for synchronization, ordered in sequences, and joined in transitions. The system part of QuickTime synchronizes the playing of tracks and makes sure that all data are decompressed when needed.

The track concept in QuickTime is similar to the Athena Muse[15] concept of time dimension: elements of information to be displayed can be attached to points in this time dimension. Dimensions other than time are also allowed. These dimensions do not need to represent physical time and space, but can represent any changing parameter in the system.

The track or dimension concepts can be mapped in the coordinate space concept defined in the Synchronizable objects; in other words, the model advocated by QuickTime can

be implemented within the framework of the PREMO model without further complications. Of course, applications using the PREMO synchronization mechanisms could also use the QuickTime format for storing or interchanging multimedia presentations.

The model described in Hamakawa et al.[11] introduces the concept of temporal glue as an extension of TeX's glue. An object composition model is built using this glue concept, describing the static aspects of multimedia interfaces. It is essentially a hierarchical synchronization model allowing relative positioning of objects. It also has some features to bypass the strict hierarchy and attach constraints at certain points in time. The resulting library is a set of C++ classes and there are several composite objects with predefined layouts for the objects they manage. This approach has similarities with the work reported in Guimarães et al.[10] although the latter uses Xt rather than Interviews.

The main limitation of these models, which use concepts based on graphical user interface toolkits, is the difficulty of integrating the usual hierarchical organization between objects derived from the graphical model with the need to bypass this hierarchy to have generic synchronization relations between objects. Nevertheless, the concept of space/time isomorphism gives a uniform way of programming relations and more importantly, composition in multimedia toolkits. Proposals for extensions to the PREMO synchronization objects related with composition should take these graphical models into account, too.

The MHEG standard[17] provides an encoding scheme for multimedia/hypermedia information that can be used and interchanged by applications. The standard aims to provide generic multimedia/hypermedia structures also suited for synchronization. The MHEG standard defines a number of attributes and behaviours that content objects and "rt-objects" (view objects) may have under the control of an "MHEG engine". Several of these attributes and behaviours are used by temporal relations, for example, speed, temporal position and timestones. Although the Standard sets requirements, as already cited above, it does not give details on how these requirements should be fulfilled. This is where the two standards, i.e., MHEG and PREMO, meet. The focus of PREMO is to give a model for the presentation process, rather than on the interchange requirements.

All the attributes of MHEG are, in general, supported by PREMO objects, so they can be used as the basis for the (conceptual or concrete) implementation of an MHEG engine. As an example, the speed and temporal position attributes are supported within the `Synchronizable` object and timestones can be supported by the reference point mechanism. A complete mapping of MHEG to PREMO would be to complex to include here, but the PREMO objects were designed taking into account the time and synchronization requirements of the MHEG standard.

7 Conclusions and related work

This paper has presented a standard, object–oriented model for inter–media synchronization in multimedia systems to become part of an international ISO Standard on multimedia presentation. The model represents a clean integration of various synchronization methods used in practice. The approach chosen maintains a clear conceptual separation between event–based and time–based synchronization, thereby allowing applications to choose whichever view is more appropriate to their specific application.

Although no complete implementation of PREMO yet exists, the major components of the model presented in the paper have been tested through the various systems which have influenced the design. The purely event–based synchronization (Sect. 2) has been implemented through the MADE toolkit[13,20], although the specification currently in PREMO is much more precise and more detailed than the toolkit version. Similarly, experimental implementation of the MSS system[16], which has deeply influenced the specification of streams (Sect. 4), have been explored by SunSoft but, here again, integration of the initial MSS design with the PREMO framework has lead to a much cleaner and more precise specification.

Although the synchronization objects described in this paper may be used in complete isolation from the rest of the architecture defined in PREMO, they are primarily intended to be used as building blocks of other, complex units within PREMO called virtual devices. These virtual devices are active entities which are responsible for the input, output, and the processing of multimedia data. The synchronization objects appear as controlling entities on the input and the output ports of these devices. Together they form a portable, i.e., platform independent abstraction for processing multimedia data. Unfortunately, the detailed description of these objects would go beyond the scope of this paper. However, some of the problems arising in multimedia systems are solved through the complex interaction of these entities. Some of these are addressed below.

7.1 Complex synchronization patterns

Multimedia presentations are often defined through the description of complex structures, involving time–based constraints (an entity should be presented before the other, in parallel with another, etc.). These descriptions are usually of a descriptive nature, and a special processing entity is responsible for parsing such descriptions and driving the underlying presentation units accordingly.

Whereas the synchronization objects described in this paper control the low–level processing of media data propagation, such descriptive structures are also defined in PREMO

(modelled after well–understood descriptive approaches, see, for example, [2] or [25]), together with special objects (called *Synchronizers*) whose role is to parse these descriptions and to set the synchronization points on the ports of the various virtual devices. The reader should refer to the PREMO document for the details of this mechanism.

7.2 Quality of service

The notion of *quality of service* (QoS) has received significant attention in the past. This term is used to represent the application requirements for specific resources, such as minimal or maximal resolution, allowed error rates, timing requirements, etc. Systems including QoS services are expected to dynamically change their behaviour to fulfil the required QoS, for example, reducing the quality of the generated image, lowering the sample rate, etc.

PREMO does include a mechanism for a rudimentary control over quality of service, primarily in the framework of virtual devices. This mechanism, and the corresponding tools used by it, may be used to control the degradation of media flow. Description of this mechanism would go beyond the scope of this paper. In any case, further work to develop a consistent and general model for QoS for PREMO is still necessary. Such a model would have some influence on the synchronization model (and vice versa).

7.3 Constraint satisfaction problems

The requirements of synchronization, primarily of time–based synchronization, are very close to general constraint management problems. The PREMO group has investigated the possibility of including “hooks” for general constraint satisfaction algorithms into PREMO. However, in view of the conflicts between the object–oriented paradigm and the requirements of constraint satisfaction (e.g., encapsulation of global state), at the present time it does not seem feasible to include a general definition of such hooks. This investigation will, however, continue in future.

Acknowledgements. Most of the work presented in this paper has been carried out under the auspices of the ERCIM Computer Graphics Network, funded under the European Communities CEC HCM Programme. The authors would like to acknowledge the participation of members of this programme, and particularly those involved in Task 1.

References

1. D.B. Arnold and D.A. Duce, *ISO Standards for Computer Graphics: The First Generation*, Butterworths, London, 1990.
2. W. Appelt and A. Scheller, “HyperODA — Going Beyond Traditional Document Structures”, in: *Computer Standards & Interfaces*, 17(1):13-21, 1995.
3. P.J. Barnard and J. May, “Interaction with Advanced Graphical Interfaces and the Deployment of Latent Human Knowledge”, in: *Design, Specification and Verification of Interactive Systems DSV-IS’94,* F. Paternò (editor), Springer Verlag, Focus on Computer Graphics Series, Heidelberg, 1995.
4. G. Blakowski, J. Hübel, and U. Langrehr, “Tools for Specifying and Executing Synchronized Multimedia Presentation”, in: *Proceedings of the Second International Workshop on Network and Operating System Support for Digital Audio and Video*, 1992.
5. R.B. Dannenberg, T. Neuendorffer, J. Newcomer, D. Rubine and D. Anderson, “Tactus: Toolkit-level Support for Synchronized Interactive Multimedia”, *Multimedia Systems Journal*, 1(2):77-86, 1993.
6. D.J. Duke, (editor), *PREMO Object Types for Synchronization, A Formal Specification*, doc. no. ISO/IEC JTC1/SC24/WG6/OME–116, 1995, available on line on the URL address ftp://ftp.cwi.nl/pub/premo/RapporteurGroup/Miscellaneous/OME-116.ps.gz.
7. R. Duke, G. Rose, and G. Smith, “Object–Z: A Specification Language Advocated for the Description of Standards”, *Computer Standards & Interfaces*, 17(5-6):511-534, September 1995.
8. S.J. Gibbs, L. Dami, and D.C. Tsichritzis, “An Object–Oriented Framework for Multimedia Composition and Synchronisation”, in: *Multimedia (Systems, Interaction and Applications)*, L. Kjelldahl (editor), EurographicSeminar series, Springer Verlag, Berlin — Heidelberg — New York, 1992.
9. S.J. Gibbs and D.C. Tsichritzis, *Multimedia Programming*, ACM Press, Addison-Wesley, Wokingham — Reading, 1995.
10. N. Guimarães, N. Correia, and T. Carmo, “*Programming Time in Multimedia User Interfaces*”, in: *Proceedings of the UIST’92 Conference*, Monterrey, CA, USA, 1992.
11. R. Hamakawa, and J. Rekimoto, “Object Composition and Playback Models for Handling Multimedia Data Processing”, in: *Proceedings of the First ACM International Conference on Multimedia (MM’93),* Anaheim, CA, USA, 1993.
12. I. Herman, G.S. Carson, J. Davy, P.J.W. ten Hagen, D.A. Duce, W.T. Hewitt, K. Kansy, B.J. Lurvey, R. Puk, G.J. Reynolds, and H. Stenzel, “Premo: an ISO Standard for a Presentation Environment for Multimedia Objects”, in: *Proceedings of the Second ACM International Conference on Multimedia (MM’94)*, San Francisco, D. Ferrari, editor, ACM Press, 1994.

13. I. Herman, G.J. Reynolds, and J. Davy, "MADE: A Multimedia Application Development Environment", in: *Proceedings of the IEEE International Conference on Multimedia Computing and Systems (ICMCS'94)*, Boston, L.A. Belady, S.M. Stevens, and R. Steinmetz, editors, IEEE CS Press, Los Alamitos, 1994.
14. I. Herman, G.J. Reynolds, and J. Van Loo, "PREMO: An Emerging Standard for Multimedia Presentation", *IEEE MultiMedia, 3(3),* 1996.
15. M. Hodges, R. Sannett, and M. Ackerman, "A Construction set for Multimedia Applications", *IEEE Software*, January 1989, 22(1):37–43.
16. IMA, *Multimedia System Services*, Interactive Multimedia Association, September 1994, ftp://ima.org/pub/mss/.
17. International Organization for Standardization, *Information Technology — Coding of Multimedia and Hypermedia Information (MHEG) (ISO/IEC 13522–1)*, October 1994.
18. International Organization for Standardization, *Information processing systems — Computer graphics — Presentation environment for multimedia objects (PREMO), ISO/IEC 14478*, September 1996, http://www.cwi.nl/Premo/docs.html.
19. J. F. Koegel Buford, "Architecture and Issues for Distributed Multimedia Systems", in: *Multimedia Systems,* J.F. Koegel Buford (editor), ACM Press, Addison Wesley, Reading, 1994.
20. A. Lie and N. Correia, "Cineloop Synchronization in the MADE Environment", in: *Proceedings of the IS&T/SPIE Symposium on Electronic Imaging, Conference on Multimedia Computing and Networking*, San Jose, 1995.
21. V. de May and S. Gibbs, "A Multimedia Component Kit", in: *Proceedings of the First ACM International Conference on Multimedia (MM93),* P.V. Rangan, (editor), Anaheim, ACM Press, 1993.
22. T. Meyer–Boudnik and W. Effelsberg, "MHEG Explained", *IEEE Multimedia*, 2(1):26–38, 1995.
23. OMG, *Joint Object Services Submission (JOSS), Event Service Specification,* OMG TC Document 93.7.3, Object Management Group, July 1993.
24. H. Tokuda, "Operating System Support for Continuous Media Applications", in: *Multimedia Systems,* J.F. Koegel Buford (editor), ACM Press, Addison Wesley, Reading, 1994.
25. M. Vazirgiannis and T. Sellis, "Event and Action Representation and Composition for Multimedia Application Scenario Modelling", in: *Interactive Distributed Multimedia Systems and Services, Proceedings of the European Workshop IDMS'96*, Berlin, B. Butscher, E. Moeller, and H. Pusch, editors, Springer Verlag, 1996.
26. P. Wayner, "Inside QuickTime", *Byte*, 37–43, December 1991.
27. "Middleware System Services Architecture", in: *Multimedia Systems,* J.F. Koegel Buford (editor), ACM Press, Addison Wesley, Reading, 1994.