# MOCS: an Object-Oriented Programming Model for Multimedia Object Communication and Synchronization

*Chi-Leung Fung* and *Man-Chi Pong*
Department of Computer Science
The Hong Kong University of Science and Technology
Clear Water Bay, Kowloon, Hong Kong
(clfung@cs.ust.hk and mcpong@cs.ust.hk)

## Abstract

*This paper describes MOCS, an object-oriented programming model for* multimedia object communication and synchronization. *MOCS serves to hide the details of* communication *of various types of data over the network and* synchronization *of the playback of various data streams at the destination system through high-level programming support. In MOCS, the concept of mirror object is used to provide the application a consistent view of the objects in different systems. The concept of composite object is employed to encapsulate the synchronization of multiple data streams. A declarative approach to specify synchronization, which is easy to understand, is advocated. A mechanical way to transform a declarative synchronization specification to operational implementation is devised and implemented in SAMOCS* (software architecture for MOCS)*, which provides application programming interface routines to let the programmer develop distributed multimedia programs more easily.*

## 1 Introduction

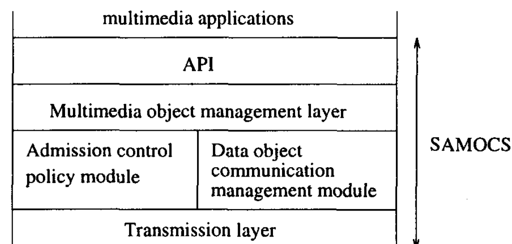
Multimedia applications are becoming popular. Many prototype distributed multimedia applications only use operating system primitives to transmit data because higher-level support is not available. For example, many Unix-based systems use Unix sockets for packet transmission between processes [1, 2]. The need to program at such a primitive level is a chore to the application programmers. We envisage that higher level support should be provided so that the applications can be developed more efficiently.

This paper presents an object-oriented programming model for *multimedia object communication and synchronization*, called *MOCS*. The aim is to provide support for the programmer to develop distributed multimedia applications using an object-oriented *application programming interface* (API). The major advantage is hiding the details of *communication* of various types of data over the network and *synchronization* of the playback of various data streams at the destination system. (The destination system is the place to do the final synchronization because network delay and jitter affect any earlier synchronization.)

MOCS employs inheritance and models different data streams as different data objects. The concepts of *mirror object* and *composite object* are employed to help the programmer to understand the communication and synchronization aspects, respectively. These present to the programmer a clear picture of what data objects exist in a distributed multimedia system, and what the temporal relationships among the objects are.

A prototype implementation, called *SAMOCS* (*software architecture for MOCS*), has been built. Figure 1 shows the software layers in SAMOCS. The details of the layers of SAMOCS are beyond the scope of this paper. Full details of MOCS and SAMOCS are described in Fung's thesis [3].

Figure 1: Software architecture for multimedia object communication and synchronization






1063-6927/94 $03.00 © 1994 IEEE

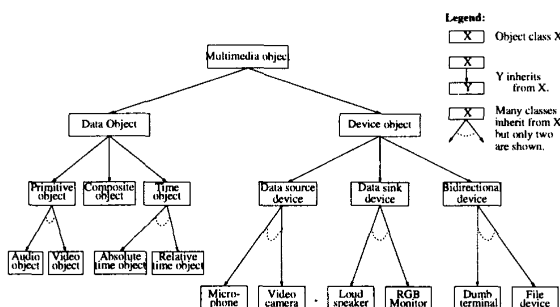
This paper highlights the features of MOCS and in particular, describes our declarative approach to specifying synchronization of multiple data streams. Section 2 describes MOCS and the communication issues. Section 3 describes the synchronization issues. Section 4 describes how to transform the declarative synchronization specification into operational implementation. Section 5 gives the implementation status of SAMOCS and the conclusion.

## 2 Object-oriented model MOCS

Multimedia data can be classified into *continuous media* (CM) data, like audio and video, or *non-continuous media* (non-CM) data, like text and graphics. Different types of data can be transmitted using different protocols which exploit the data characteristics, such as no retransmission for CM data.

Figure 2 shows the class hierarchy of multimedia objects in MOCS. The classes `data object` and `device object` are used to encapsulate the details of various media data streams and the devices associated with the data, respectively. `Data object`'s subclasses model the actual data streams and the synchronization of their playback.

Figure 2: Object class hierarchy in MOCS




### 2.1 Data object

The subclasses of **data object** are **primitive object**, **composite object**, and **time object**. (The latter two are discussed in section 2.3 and 2.4.)

A **primitive object** is used to model a **data stream**, regardless it is CM or non-CM data. Each primitive object has a *data type* attribute, such as **audio**, **video** or **text**.

An application can create a primitive object either at the source system or destination system through an API call. A schematic example is:

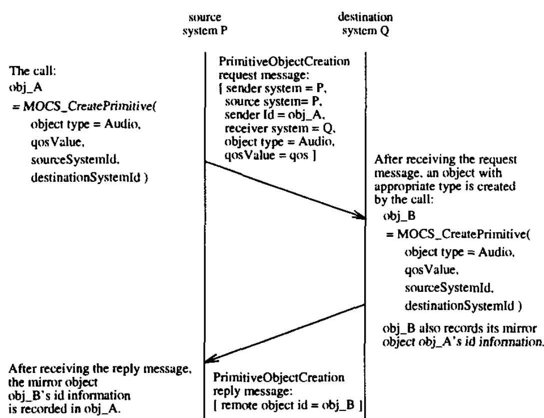
```
obj_id := MOCS_CreatePrimitive(
          object_type = Audio,
          qualityOfServicesParameters,
          sourceSystemInformation,
          destinationSystemInformation )
```

On return from this call, a *mirror object* is also created automatically in the peer system by the run-time support platform. (If the source and the destination are the same system, no mirror object is created. This trivial case would not be discussed further.)

Figure 3 shows the scenario of the creation of a primitive object initiated at the source system. Creation initiated at the destination system is similar in that messages are passed between the systems to create the mirror object at the source system.

The idea of peer mirror object is also applicable for communicating objects in multiple systems. We shall not go into details for the sake of brevity.

Figure 3: Creation of mirror object




After a mirror primitive object is created, we say that a *data message connection* is set up between the peer objects. It supports the transmission of data messages using a network protocol appropriate for that data type.

Besides, a *control message connection* is also set up. The exchange of control information between peer objects is done by passing control messages, which are sent using a reliable network protocol, say, TCP/IP.

After the creation of a primitive object, all operation requests on the primitive object at either the

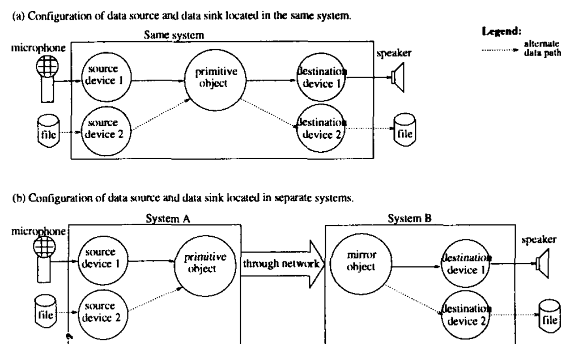
source or destination system will be reflected appropriately to the peer object by control messages. This provides the application a consistent view of the objects in different systems. The application can access the objects in any system as if they are local.

## 2.2 Device object

A data stream may come from or go to different devices. For example, audio data from a file or a microphone may be sent to a destination machine where it can be saved into a file or sent to a speaker for replay. Thus, the concept of device object is used and it has various subclasses. Each device object encapsulates the characteristics of a device. A device object can be a data source (like a microphone), or a data sink (like a speaker), or both a source and a sink (like a file). (See Figure 2.)

The separation of the data and its associated devices in MOCS allows a data object to be bound to different device objects of a system at different time in an application. This is conceptual clear and supports flexible application. Figure 4 shows different scenarios of bindings.

Figure 4: Data objects bound to device objects




## 2.3 Composite object

We use the concept of *composite object*, which contains temporally related primitive objects and/or other composite objects, to encapsulate the synchronization relationships among the contained data objects. The application can specify the relationships of the playback of multiple data streams using a *declarative* approach (described in section 3). The declarative specification of synchronization relationships is easy to do and is clear.

In section 4, we show how to transform the declarative specification into an operational implementation that will drive the playback of the objects. This mechanical transformation ensures that the multiple data streams are synchronized properly during playback.

## 2.4 Time object

MOCS is special is using the concept of *time object* in synchronization specification (described in section 3.1.3). This makes the specification relations consistently taking objects of the class `data object` as parameters, and the programmer can specify and understand the specification more easily. Most synchronization specification methods, including those based on the object-oriented approach, do not explicitly use the concept of a time object to specify the time [4, 5, 6, 7].

# 3 Synchronization between objects

Synchronization of temporally related objects is an important issue in distributed multimedia applications. We characterize the multimedia synchronization problem into two types:

- *Boundary synchronization* specifies when to start or stop data streams.
- *Continuous synchronization* specifies how often a set of CM data streams must be synchronized with each other. Among them, one is the *master stream* and others are *slave streams*. The master, say, an audio stream, is regarded as more important with respect to keeping up with the physical time. Other slaves are synchronized with it at specific *synchronization points* along the streams.

Both types of synchronization are important and have to be addressed in multimedia presentation. Boundary synchronization coordinates the start and stop of the playback of data streams and continuous synchronization coordinates their actual playback processes. The application should specify the boundary synchronization of streams in a multimedia presentation while continuous synchronization should be handled by the run-time support system.

Continuous synchronization has been studied by many researchers [4, 8, 9, 10, 11, 12]. Different solutions have been proposed, such as *logical time system* [9] and *restricted blocking* [12]. Continuous synchronization is not the focus of this paper. We adopt the *logical time system* solution in SAMOCS, and uses

*pausing* the playback of a fast data stream and *skipping* the playback of a slow data stream in the implementation.

A contribution of SAMOCS is the provision of declarative API routines for the application to specify boundary synchronization relations. Details are given in next section. (SAMOCS also provides API routines for specifying which data object in a composite object is the master for continuous synchronization.)

## 3.1 Synchronization specifications

Different boundary synchronization specifications have been proposed. We review the popular schemes below and then describe our declarative approach. (Hereafter, the term *synchronization* refers to boundary synchronization unless otherwise stated.)

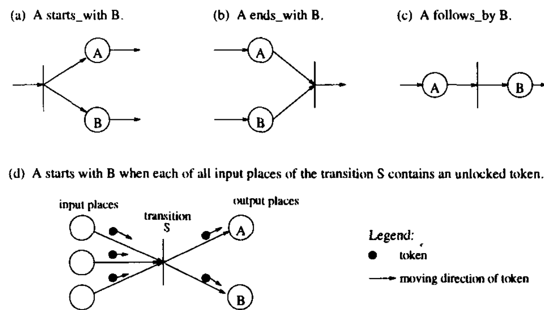
### 3.1.1 Hierarchical specification

In *hierarchical synchronization specification*, multimedia presentation is represented as a tree consisting of nodes that denote serial and parallel presentation of the leaf nodes of data streams [13, 14]. It is claimed that the hierarchical specification is easy to handle [13]. However, there are some boundary synchronization that cannot be expressed by the hierarchical specification [13].

### 3.1.2 Petri net specification

Petri net [15] is a commonly used graphical means to show synchronization relationships. Three basic Petri nets for synchronization are shown in Figure 5(a)-(c) respectively.

Figure 5: Basic Petri nets to specify synchronization



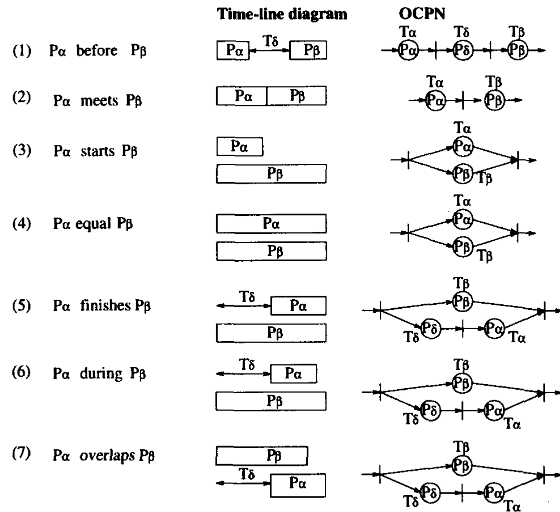

In a Petri net, a *place* (drawn as a circle) represents the playback process of a data stream. A token entering a place is first *locked*, which indicates that the playback process is being executed. Then after the playback process is done, the token is *unlocked*. The movement of tokens among the places (i.e., the *firing rule*) is as follows: when each of all input places of a *transition* (drawn as a short vertical line) has an unlocked token, then the token is consumed and the transition *fires*, meaning that each of all output places of the transition will receive a token.

The simple Petri net has been extended to become *Object Composition Petri Net* (OCPN) that includes duration constraints and resource mappings for specifying the synchronization properties [4, 8, 16]. As illustrated by Hamblin [17], there are seven possible distinct temporal relationships between two data streams. The OCPNs for these relationships are shown in Figure 6, where $T_\alpha$, $T_\beta$ and $T_\delta$ represent the duration of places $P_\alpha$, $P_\beta$ and $P_\delta$, respectively. When simply looking at the OCPN without knowing their timing expression, it is impossible to differentiate the cases between (3) and (4), and between (5), (6) and (7). In fact, the timing expression for cases (3), (4), (5), (6) and (7) are:

(3) $T_\alpha < T_\beta$
(4) $T_\alpha = T_\beta$

(5) $T_\beta = T_\alpha + T_\delta$
(6) $T_\beta > T_\delta + T_\alpha$
(7) $T_\alpha < T_\delta + T_\beta$

Figure 6: Seven temporal relationships

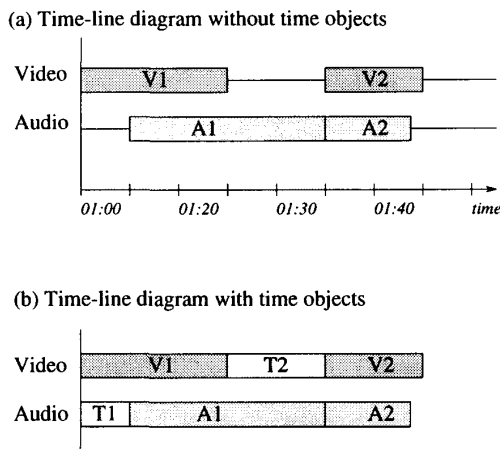


Moreover, OCPN does not convey any operational mechanism to accomplish the specified synchronization constraints. In Figure 5(d), the transition is fired when each of its input places contains an unlocked token. Then both the playback processes of output places A and B should be started *immediately*. However, when the transition is fired, network delay may make streams A and B unable to start at the same time.

### 3.1.3 Time-line specification

An intuitive way to specify a multimedia presentation is to use a time-line diagram, e.g., as in QuickTime [18] and MAEStro [19]. In a time-line diagram, a rectangular strip represents a data object to be played back. The playback time of an object can be projected onto the time axis. Figure 7(a) shows a synchronization specification of a multimedia presentation with two video and two audio data streams.

Figure 7: A presentation specified in time-line diagram




Often, relative relationships are more useful than the absolute time specification because for most of the cases, the application only specifies the relative relationships such as “A1 is followed by A2” rather than “A1 stops at *01:30* and A2 starts at *01:30*”. Figure 7(b) shows a modified time-line diagram of Figure 7(a) using a relative relationship approach. It also introduces *time objects* $T1$ and $T2$. The class `time object` is a subclass of `data object` (see Figure 2). A time object is like a data object but has no data to be play backed. It can be a relative or an absolute time object. A relative time object only has a duration attribute, *lifetime*. It starts when its previous stream is ended and terminates after its lifetime is expired. An absolute time object has an additional attribute of *absolute start time*, the time at which its lifetime starts.

The use of the time object concept enables us to specify boundary synchronization easily with the following three relations:

- *starts_with(reference object, synchronizing object).*
  The synchronizing object should start at the same time with the reference object.
- *ends_with(reference object, synchronizing object).*
  The synchronizing object should stop at the same time with the reference object.
- *follows_by(reference object, synchronizing object).*
  The synchronizing object should start at the stop time of the reference object.

These three relations are called *declarative relations* because they only declare *what* is expected, but not *how* to achieve it.

For example, referring to Figure 7(b), let T1 be an absolute time object with a lifetime of 5 minutes, and its start time is the start time of the whole presentation; and let T2 be a relative time object with a lifetime of 10 minutes. Then we can specify the presentation using SAMOCS API routines as follows:

```
c := MOCS_CreateCompositeObject();
c.AddObject( V1 ); c.AddObject( V2 );
c.AddObject( A1 ); c.AddObject( A2 );
c.AddObject( T1 ); c.AddObject( T2 );
c.starts_with( T1, V1 );
c.follows_by( V1, T2 );
c.follows_by( T2, V2 );
c.follows_by( T1, A1 );
c.follows_by( A1, A2 );
c.ends_with( T2, A1 );
c.starts_with( V2, A2 );
```

To help the application programmer further, a graphical tool could be built to let him specify the desired synchronization requirements using a time-line diagram, and then the tool would generate the declarative relations.

In the implementation of SAMOCS, a mechanical way to synchronize the multiple data streams in a composite object based on the declarative specification is devised and is described below.

# 4 Transform synchronization specification to operational implementation

The playback of each data stream is performed by a process which is subdivided into three subprocesses: *StartPlay*, *MainPlay*, and *EndPlay* (Figure 8).

Figure 8: Subprocesses handling a data stream

- *StartPlay*
  It does not consume any data, and its main aim is to prepare the data to be played back in *MainPlay*. For most data streams, *StartPlay* simply waits for sufficient data and buffers them up.
- *MainPlay*
  It feeds data to the output device, which presents the data to the end-user. The start time of *MainPlay* is the start time of the data stream perceived by the end-user. When all the data has been presented, *MainPlay* may have to do some actions depending on the data type, such as displaying the last frame of a video stream, until some stop signal of the playback is received.
- *EndPlay*
  This is a clean-up subprocess. It stops the whole playback process.

Each subprocess is started only when some *operational pre-conditions* are satisfied. We now show what their operational pre-conditions are.

Let `Completed(subprocess)` be a boolean function that returns *true* only if `subprocess` is completed, and `AbleToStart(subprocess)` be another boolean function that returns *true* only if `subprocess` is ready to start.

Four sets of data objects are used to record the synchronization relationships between a data object $i$ and other data objects. They are:

$S^i_{start}$: the set of objects that starts with object $i$,
$S^i_{end}$: the set of objects that ends with object $i$,
$S^i_{after}$: the set of objects that follows object $i$,
$S^i_{before}$: the set of objects that precedes object $i$.

Initially, when the first object is put into the composite object, these four sets are empty.

Assume that the synchronization relationships among $n$ objects in a composite object have been specified. When another synchronization relation of object $B$ with reference to object $A$ is specified, the synchronization information is updated as follows.

When *Starts_with*$(A, B)$ is specified (Figure 9(a)),

$$\begin{array}{ll} & S^A_{start} := S^A_{start} \cup \{B\} \\ & S^B_{start} := S^B_{start} \cup \{A\} \\ \forall x \in S^A_{before} \mid & S^x_{after} := S^x_{after} \cup \{B\} \\ \forall x \in S^A_{before} \mid & S^B_{before} := S^B_{before} \cup \{x\} \\ \forall y \in S^A_{start} \mid & S^y_{start} := S^y_{start} \cup \{B\} \\ \forall y \in S^A_{start} \mid & S^B_{start} := S^B_{start} \cup \{y\} \end{array}$$

When *Ends_with*$(A, B)$ is specified (Figure 9(b)),

$$\begin{array}{ll} & S^A_{end} := S^A_{end} \cup \{B\} \\ & S^B_{end} := S^B_{end} \cup \{A\} \\ \forall x \in S^A_{after} \mid & S^x_{before} := S^x_{before} \cup \{B\} \\ \forall x \in S^A_{after} \mid & S^B_{after} := S^B_{after} \cup \{x\} \\ \forall y \in S^A_{end} \mid & S^y_{end} := S^y_{end} \cup \{B\} \\ \forall y \in S^A_{end} \mid & S^B_{end} := S^B_{end} \cup \{y\} \end{array}$$

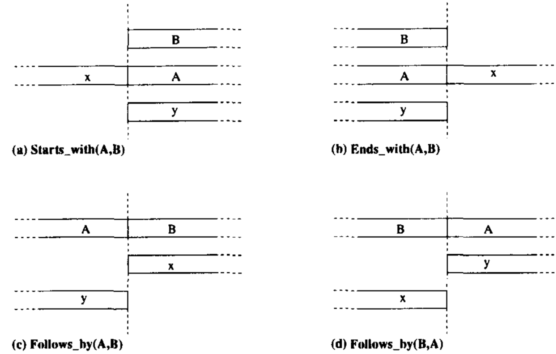
When *Follows_by*$(A, B)$ is specified (Figure 9(c)),

$$\begin{array}{ll} & S^A_{after} := S^A_{after} \cup \{B\} \\ & S^B_{before} := S^B_{before} \cup \{A\} \\ \forall x \in S^A_{after} \mid & S^x_{start} := S^x_{start} \cup \{B\} \\ \forall x \in S^A_{after} \mid & S^B_{start} := S^B_{start} \cup \{x\} \\ \forall y \in S^A_{end} \mid & S^y_{after} := S^y_{after} \cup \{B\} \\ \forall y \in S^A_{end} \mid & S^B_{before} := S^B_{before} \cup \{y\} \end{array}$$

When *Follows_by*$(B, A)$ is specified (Figure 9(d)),

$$\begin{array}{ll} & S^B_{after} := S^B_{after} \cup \{A\} \\ & S^A_{before} := S^A_{before} \cup \{B\} \\ \forall x \in S^A_{before} \mid & S^x_{end} := S^x_{end} \cup \{B\} \\ \forall x \in S^A_{before} \mid & S^B_{end} := S^B_{end} \cup \{x\} \\ \forall y \in S^A_{start} \mid & S^y_{before} := S^y_{before} \cup \{B\} \\ \forall y \in S^A_{start} \mid & S^B_{after} := S^B_{after} \cup \{y\} \end{array}$$

Figure 9: New synchronization relation specified




Based on these four sets of data objects, we can derive the pre-conditions for the start of each subprocess such that the presentation of the data streams will conform to the specified relations.

Let $x_{subprocess}$ be data object $x$'s *subprocess*.

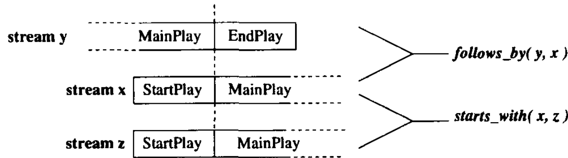
(1) $AbleToStart(x_{StartPlay})$ returns *true* if
not( $x_{MainPlay}$ or $x_{EndPlay}$ in action )

Since the *StartPlay* subprocess is only responsible for preparing the start-up buffer, it is independent of other data streams and can start whenever the multimedia presentation is started. In practice, it can be started some time before the actual start time of $x$, depending of the type of $x$.

(2) $AbleToStart(x_{MainPlay})$ returns *true* if
$\forall y \in S^x_{before} \mid Completed(y_{MainPlay})$ returns *true*
$\bigwedge \forall z \in S^x_{start} \mid Completed(z_{StartPlay})$ returns *true*
$\bigwedge Completed(x_{StartPlay})$ returns *true*.

Figure 10 shows the dependencies of the *x's MainPlay* subprocess with other objects. From the end-user's point of view, for any *y*, *follows_by(y, x)* means that the completion of *y*'s *MainPlay* is immediately followed by *x*'s *MainPlay*. Similarly, for any *z*, *starts_with(x, z)* means that *x*'s and *z*'s *MainPlay* must start together, and thus the completion of *x*'s and *z*'s *StartPlay* are also among the pre-conditions for *x's MainPlay* to start.

Figure 10: AbleToStart conditions for MainPlay

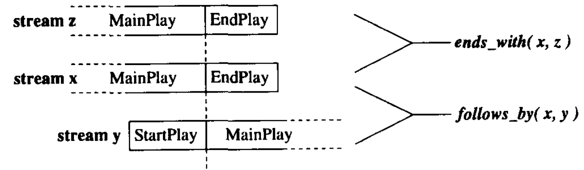



(3) $AbleToStart(x_{EndPlay})$ returns *true* if
$\forall y \in S^x_{after} \mid Completed(y_{StartPlay})$ returns *true*
$\bigwedge \forall z \in S^x_{end} \mid Completed(z_{MainPlay})$ returns *true*
$\bigwedge Completed(x_{MainPlay})$ returns *true*.

Similar to Figure 10, Figure 11 shows that *x*'s *EndPlay* depends on *y*'s *StartPlay* in the *follows_by(x, y)* relations and on *z*'s *MainPlay* in the *ends_with(x, z)* relations.

By checking the *AbleToStart* conditions of *StartPlay*, *MainPlay*, and *EndPlay* for each data object, the implementation can schedule its playback accordingly and the synchronization specification is realized.

Figure 11: AbleToStart conditions for EndPlay




## 5 Implementation and conclusion

A prototype implementation of SAMOCS has been implemented on SUN SPARCstation 2/ SunOS 4.1.3 platform using the programming language Modula-3 [20]. Modula-3 is a modular strongly-typed object-oriented language with garbage collection. These features make the programming task easy and safe.

The Unix version of Modula-3 is distributed with a thread module. In our prototype implementation, we use a thread to represent a primitive data object. The performance of using a thread per such object is not a problem in our platform.

Continuous media (CM) data is sent using UDP/IP protocol while non-CM data and control messages among peer objects are sent using TCP/IP protocol.

Audio streams are used as CM objects and text as non-CM objects in our demonstration programs running over an ethernet. Boundary synchronization is implemented as described in section 4 and performs as expected.

Continuous synchronization is implemented using the *logical time system* [9]. SAMOCS maintains the progress of the playback of each data object in terms of the length of the data processed or clock signals for time object. Exceptional conditions, like long delay of arrival of data, are detected by SAMOCS and reported to the application. Based on the exception type, the application can handle accordingly.

Since the thread scheduling is not priority based and the Unix system schedules the SAMOCS application process among other processes in Unix, jitter in the playback of multiple continuously synchronized data streams is perceivable by the end-user. This problem cannot be solved satisfactorily without the real-time support of the operating system kernel.

We shall re-implement SAMOCS in C++ on Solaris 2.3 operating system, which has priority thread scheduling support.

Currently, we assume that the network can provide sufficient bandwidth for the data traffic. We shall

look into the issues of admission control and policing to guarantee the applications' quality of services. We shall also acquire a high speed ATM network. Distributed multimedia applications including video data, like teleconferencing system, will be built with the support of SAMOCS. We envisage that the development of such applications will be easier because and the application programmers need not worry about the details of communication and synchronization of data objects.

## Acknowledgement

This research is partially supported by a grant from the Hong Kong Research Grants Council.

## References

[1] H. M. Vin, P. T. Zellweger, D. C. Swinehart, and P. V. Rangan, "Multimedia conferencing in the etherphone environment," *IEEE Computer*, vol. 24, no. 10, pp. 69–79, 1991.

[2] T. Crowley, P. Milazzo, E. Baker, H. Forsdick, and R. Tomlinson, "Mmconf: An infrastructure for building shared multimedia applications," in *CSCW'90 Proceedings*, pp. 329–338, Oct 1990.

[3] C.-L. Fung, *A software Architecture for Multimedia Object Communication and synchronization.* M.Phil. thesis, The Hong Kong University of Science and Technology, Hong Kong, August 1993.

[4] T. D. C. Little and A. Ghafoor, "Synchronization and storage models for multimedia objects," *IEEE Journal on Selected Areas of Communications*, vol. 8, pp. 413–427, Apr 1990.

[5] S. Gibbs, "Composite multimedia and active objects," in *OOPSLA'91*, pp. 97–112, 1991.

[6] F. Horn and J. B. Stefani, "On programming and supporting multimedia object synchronization," *Computer Journal*, vol. 36, no. 1, pp. 5–18, 1993.

[7] M. Vazirgiannis and C. Mourlas, "An object-oriented model for interactive multimedia presentations," *Computer Journal*, vol. 36, no. 1, pp. 79–86, 1993.

[8] T. D. C. Little and A. Ghafoor, "Network consideration for distributed multimedia object composition and communication," *IEEE Network Magazine*, pp. 32–49, Nov 1990.

[9] D. P. Anderson and G. Homsy, "A continuous media i/o server and its synchronization mechanism," *IEEE Computer*, vol. 24, pp. 51–57, Oct. 1991.

[10] M. M. Mourad, "Some issues in the implementation of multimedia communication systems," in *Proc. Second IEEE Workshop on Future Trends of Distributed Computing Systems*, pp. 4–13, 1990.

[11] C. Nicolaou, "An architecture for real-time multimedia communication systems," *IEEE Journals on Selected Areas in Communications*, vol. 8, pp. 391–400, Apr 1990.

[12] R. Steinmetz, "Synchronization properties in multimedia systems," *IEEE Journal on Selected Areas in Communications*, vol. 8, pp. 401–412, Apr 1990.

[13] G. Blakowski, J. Hübel, and U. Langrehr, "Tools for specifying and executing synchronized multimedia presentations," in *Network and Operating System Support for Digital Audio and Video*, pp. 271–282, Nov 1991.

[14] M. Salmony and D. Shepherd, "Extending osi to support synchronization required by multimedia applications," Technical Report 43.8904, IBM European Networking Center, Tiergartenstr, 8,6900 Heidelberg P.O. Box 103068, Germany, Apr 1989.

[15] E. Best and C. F. C., *Nonsequential Processes*, vol. 13 of *Monographs on Theoretical Computer Science.* Springer-Verlag, 1988.

[16] T. D. C. Little and A. Ghafoor, "Spatio-temporal composition of distributed multimedia objects for value-added networks," *IEEE Computer*, pp. 423–450, Oct 1991.

[17] C. L. Hanmblin, "Instants and intervals," in *Proc. 1st Conf. Int. Soc. for the Study of Time*, pp. 324–331, Spring-Verlag, 1972.

[18] Apple Computer, Inc., *QuickTime Developer's Guide*, 1991.

[19] G. D. Drapeau and H. Greenfield, "Maestro - a distributed multimedia authoring environment," in *Proc. Summer 1991 USENIX Conference*, p. 473, 1991.

[20] G. Nelson, *System Programming with Modula-3.* Prentice Hall, 1991.