A Unified Approach to Representation, Synchronization and Storage of Temporal Multimedia Objects based on Time Interval Logic

Yoshifumi Masunaga
University of Library and Information Science
1-2 Kasuga, Tsukuba, Ibaraki 305, Japan
masunaga@ulis.ac.jp

Abstract
This paper presents a unified approach to modeling and implementing temporal multimedia objects. The approach is based on both the object-orientation and the time interval logic investigated by Allen [Alle83]. We regard every temporal multimedia data as a time interval. Then, by introducing null time-intervals, it is shown that every temporal relationship is represented using only the equals and the meets relationship. Accordingly, to manipulate heterogeneous temporal multimedia data in a unified manner, normal forms of composite temporal multimedia objects are introduced. The normal forms allow us to implement a multimedia player, where two synchronization schemes are necessary and sufficient to play back any composite temporal object; which are the individual and the start synchronization scheme. These schemes are implemented and evaluation results are shown in this paper. In general, audio-visual data are huge. We adopted MPEG-1 standard to compress video data. A suitable storage structure for MPEG-1 video data is also shown with a class hierarchy.

1. Introduction
In order to manage temporal multimedia data such as sound and video data, it is necessary to provide a good basis for describing their temporal features. The time line approach is one of the well known methods for this purpose. In this approach, every temporal datum is regarded as a line defined on a time axis. This approach has been taken by Gibbs, et al. [GibT93], Little, et al. [LiGh93], Oomoto, et al. [OoTa93], Hamakawa, et al. [HaRe93] and others. A temporal expansion to the relational database done by Snodgrass is based on the concept of time-intervals [Snod87]. Recent activity on multimedia document standardization such as Hyper ODA is also based on this approach [Appc94].

The logical nature of time intervals has been investigated by Allen [Alle83]. It is shown that there are thirteen temporal relationships between two time intervals. We regard every temporal multimedia data as an instance of class TimeInterval or one of its subclasses [Masu95]. Our approach is also based on the time line approach, but it differs from the others in that the entire scheme of modeling and implementing temporal multimedia data is shown in the object-oriented paradigm. The object-orientation is essential, because it makes possible to homogenize heterogeneous multimedia data, to provide a unified interface to users due to the polymorphism, and to store any type of multimedia data in a unified manner.

In the previous paper [Masu96], we have shown that any one of the Allen's thirteen temporal relationships can be represented by using only relationships equals and/or meets if the null time intervals are used. The null time interval is a time interval which is used to represent the absence of audio-visual data in this time interval [FAKM94].

Now, one of the essential problems of temporal multimedia database organization and management is how to represent, synchronize, and store heterogeneous temporal multimedia data in a unified manner. To play back composite temporal objects, a synchronization scheme is necessary. Our approach based on the introduction of the null time intervals has shown that only two synchronization primitives are necessary and sufficient; which are the individual and the start synchronization. However, in order to implement those schemes, the composite temporal multimedia objects must be normalized so that a schedule to play back component
multimedia objects can be determined. The object-oriented approach based on the time interval logic seems strong to resolve this problem.

In general, the volume of audio-visual data is huge. Therefore many data compression and decompression techniques have been investigated. Among them, MPEG-1 is widely accepted. We investigate how to design and implement the storage structure for MPEG-1 video data. This investigation is necessary to show a standard class hierarchy for storing and retrieving MPEG-1 video data in an object-oriented database management system.

To provide the temporal multimedia database system in real use, the evaluation of the synchronization mechanisms is also important. We show an evaluation result by taking a prototype system OMEGA [Masu91] which is under development at our group.

In the following of this paper, the representation issues of composite temporal multimedia data are investigated in Section 2, the synchronization schemes and their implementation with evaluation data are shown in Section 3, and the storage structure for audio-visual data is investigated in Section 4. Section 5 concludes this paper.

2. Temporal Multimedia Object Representation

2.1 Object-Oriented Approach

To represent audio-visual data in OMEGA, we regard every temporal data as an object. By regarding every data as an object, the heterogeneity of multimedia data is homogenized [Masu87, -89]. Also, we characterize every temporal object as a time interval object. This is possible because a temporal object has the start and the end time point. To describe time intervals in the object-oriented paradigm, we introduce class TimeInterval. In this framework, an object is temporal if it is an instance of class TimeInterval or one of its subclasses. Definition 2.1 shows a part of class TimeInterval, where stp and etp represent the start time point and the end time point, respectively. Time is a system built-in class whose instance represents a time and date. Notice that an operation named duration() is defined to measure the duration of a time interval. Of course we can add any attributes and operations if necessary.

[Definition 2.1] (Definition of Class TimeInterval)

class TimeInterval: Object{
    // attributes
    Time stp, etp;
    // operations
    Time stp() {return stp;}
    Time etp() {return etp;}
    Time duration() {return etp-stp;}
}

2.2 A Class Hierarchy for Composite Temporal Multimedia Objects

Temporal objects are usually composed of other components. For instance, a video object consists of many consecutive frame objects. In order to represent a time interval where nothing happens in its duration, we introduced the null time intervals [FAKM94]. We can define null videos and null sounds, and we can show the normal forms of composite temporal objects using this concept, which provides a fundamental framework for temporal multimedia synchronization (see Section 3). In OMEGA, class NullTimeInterval is introduced to represent null time interval objects. Obviously, it is a subclass of class TimeInterval. Class NullVideo and NullSound are also introduced to represent the null video ob-
objects and the null sound objects. Figure 2.1 depicts a part of the OMEGA temporal object class hierarchy [Masu95, -95b].

2.3 Temporal Relationship Representation

Since temporal multimedia objects are time intervals, we can apply the well known temporal relationships investigated by Allen [Alle83] to represent the temporal relationships between them. Here we summarize the thirteen temporal relationships, where m1 and m2 represent temporal multimedia objects, and iff is a shorthand notation of if and only if.

[Allen's Temporal Logic]
1) equals(m1, m2) iff m1.stp=m2.stp and m1.etp=m2.etp
2) before(m1, m2) iff m1.etp<m2.stp
3) after(m1, m2) iff before(m2, m1)
4) during(m1, m2) iff m1.stp>m2.stp and m1.etp<m2.etp
5) contains(m1, m2) iff during(m2, m1)
6) overlaps(m1, m2) iff m1.stp<m2.etp and m1.etp>m2.stp and m1.etp<m2.etp
7) overlapped_by(m1, m2) iff overlaps(m2, m1)
8) meets(m1, m2) iff m1.etp=m2.stp
9) meets(m1, m2) iff meets(m2, m1)
10) starts(m1, m2) iff m1.etp<m2.etp and m1.etp<m2.etp
11) starts_by(m1, m2) iff starts(m2, m1)
12) finishes(m1, m2) iff m1.etp=m2.etp and m1.etp>m2.stp
13) finished_by(m1, m2) iff finishes(m2, m1)

Notice that seven relationships among them such as equals, before, during, overlaps, meets, starts, and finishes are essential because the other six are defined as their inverse relationships. For instance, relationship after is the inverse of before.

2.4 Equivalence of Temporal Relationships

We say that two time-interval relationships r1(x, y) and r2(x, y) are equivalent if and only if r1(a, b) = r2(a, b) holds for any (a, b).

[Proposition 2.1]

Any one of the thirteen temporal relationships has its equivalent obtained by using only the relationships equals and/or meets when null time-intervals are used.

[Proof]

The proof is trivial for equals and meets relationships. Therefore, let us prove the proposition for relationships before, during, overlaps, starts, and finishes. First we examine before defined by before(m1, m2) iff m1.etp<m2.etp. Now, let us introduce a null time interval n such that m1.etp=n.stp and n.etp=m2.stp. We construct a predicate m = \(\exists n)(meets(meets(m1, n), m2))\). Then it is easy to see that \(\exists n)(meets(meets(m1, n), m2))\) holds if and only if before(m1, m2) holds, i.e. they are equivalent. Proofs are similar for the other four temporal relationships, and omitted here.

Q.E.D.

Here we notice that predicates have operational interpretation as well. For example, if meets(m1, m2) holds, then we can say that a composite object meets(m1, m2) is created. Usually, a temporal multimedia object is composed of many primitive component objects. For example, a symphony performance object might consist of four movements and three suspension objects. Of course the thirteen temporal relationships are defined as the composition operators as well. We say that two composite temporal multimedia objects are equivalent if their corresponding logical expressions are equivalent in the predicate sense. For example, a composite temporal multimedia object m = during(starts(m1, m2), m3) is (operationally) equivalent to m' = equals(meets(meets(n11, m1), n12), meets(meets(n21, m2), n22)), m3), where n11, n12, n21 and n22 are null time intervals and m1.etp < m2.etp is assumed, because their corresponding logical expressions are equivalent: during(starts(m1, m2), m3) is (logically) equivalent to \(\exists n11, n12, n21, n22)(equals(meets(meets(n11, m1), n12), meets(meets(n21, m2), n22)), m3))\). This is illustrated in Figure 2.2.
Now the following is a corollary of Proposition 2.1:

**[Corollary 2.1]**

When null intervals are used, any composite temporal multimedia object has its equivalent obtained by using only operator equals and/or meets.

### 2.5 Semantics of Null Video and Sound

Null time intervals play an essential role in modeling and implementing temporal multimedia objects. Here, we examine the real meaning of the null time intervals. In OMEGA, we interpret null time intervals in the following way so that a unified synchronized playback scheme can be implemented (see next section). Suppose that there is a null video object which is an instance of class NullableVideo. In general, more than one interpretation is possible for a null video. For example, we can say that (i) this is a video consisting of no frame in its duration, or (ii) this is a video consisting of a sequence of black frames—a frame totally filled with black.

There is no reason to distinguish which alternative should be taken from logical point of view. Therefore, we decided it from practical point of view: Suppose that we want to play back a composite video object meets(n, m) which is created by applying operator meets to a null video object n and a video object m. Then, it is convenient to interpret the null video object as in (ii) because we can regard meets(m, n) as one of the intrinsic instances of class Video. The sound case is similarly defined.

### 3. Synchronized Playback Scheme

#### 3.1 Multimedia Player

Suppose that a user wants to play back a composite temporal multimedia object meets(m1, m2), where m1 and m2 are instances of class Sound and Video, respectively. To play back m1 an audio player is necessary, while a video player is necessary to play back m2. It is impossible for an audio player to play back both m1 and m2. The same happens for a video player.

A multimedia player is a software or a hardware module which can play back any type of multimedia data. Therefore, it must be able to play back meets(m1, m2) defined above. In order for the multimedia player to do so, two alternative approaches are considered:

(a) Static approach: A multimedia player has a 'normalizer' which first translates the input meets(m1, m2) into its equivalent equals(meets(m1, n1), meets(n2, m2)), where n1 and n2 are a null sound and a null video object, and then it sends two new inputs meets(m1, n1) and meets(n2, m2) to the audio and the video player, respectively, for synchronized playback.

(b) Dynamic approach: A multimedia player has an input 'dispatcher' which sends the input to the audio player when the current input is audio, and to the video player when the current input is video.

The OMEGA adopted the former approach because it matches to the normal form issues of composite temporal multimedia objects, which will be discussed in the following sections.

#### 3.2 Normal Forms of Composite Temporal Multimedia Objects

Suppose that we consider a composite object

$meets(m_1, m_2)$.

If both $m_1$ and $m_2$ belong to the same medium (i.e. class), then it itself is a normal form. If $m_1$ and $m_2$ belong to different media, say class Sound and class Video, respectively, then it is not a normal form. Its normal form is

$equals(meets(m_1, n_1), meets(n_2, m_2))$,

where $n_1$ and $n_2$ are a null sound and a null video object with $n_1.stp=m_2.etp$, $n_1.etp=m_2.stp$, $n_2.stp=m_1.stp$, and $n_2.etp=m_1.etp$.

In general, normal forms are not unique because there may exist more than one operational equivalents. For example, suppose that there is a composite object before(m1, m2), where m1 and m2 belong to different media, say Sound and Video, respectively. Then, by Corollary 2.1 both

$meets(meets(m_1, n), m_2)$ and $equals(meets(m_1, n_1), meets(n_2, m_2))$

are the operational equivalents to it, where n, n1, and n2 represent certain null time intervals. However, the former cannot be played back if it is input to the static multimedia player if m1 and m2 belong to different media.

We introduce the normal forms of composite temporal multimedia objects as follows: We first define normal forms for seven primitive composite temporal multimedia objects, which is shown in Figure 3.1. Notice that only equals and meets operators are used to define normal forms. Then the normal forms of more complicated composite temporal mul-
timedia objects are obtained by applying the normal forms in
Figure 3.1 recursively. For example, as it is depicted in Fig-
ure 2.2, a normal form of a composite object
during(starts(m1, m2), m3) is calculated as
equals(equals(meets(meets(n11, m1), n12),
meets(meets(n21, m2), n22)), m3), where n11, n12, n21, and
n22 represent certain null time-intervals.

3.3 Start and Individual Synchronization
Scheme
As we noted just above, only equals and meets opera-
tors are used to define normal forms. This means that only
two synchronization schemes are necessary to perform syn-
chronized playback of any composite temporal objects. Sup-
pose that two temporal multimedia objects m1 and m2 are related under equals, i.e. \(\text{equals}(m_1, m_2)\). This means that playback of m1 and that of m2 should start and end at the same time. Suppose that processes P1 and P2 are created to play back m1 and m2, respectively. Then P1 may want to communicate with P2 so that they can start at the same time (we call this 'start' synchronization), and then communicate with each other periodically to synchronize the speed of playback and end at the same time. However, this periodic communication cost may not be negligible. Therefore, in addition to the start synchronization we introduced yet another way to perform the latter type of synchronization. This is called the 'individual' synchronization. Under the individual synchronization, each playback process is responsible for playing back its temporal multimedia object on time. For example, if a playback process has a 10 second length video to play back, it manages the playback of a sequence of frames so that they are displayed on a screen on time. If the processing speed becomes slow, then it may skip to display a certain number of frames. As a result, if two processes P1 and P2 obey to the individual synchronization, then we can say that two playbacks are synchronized eventually.

Moreover, the individual synchronization is used to play back a composite temporal object \(\text{meets}(m_1, m_2)\), where m1 and m2 belong to the same medium. In order to play back \(\text{meets}(m_1, m_2)\), we can construct a seamless concatenation of m1 and m2 first, which becomes the object under an individual synchronization process. Therefore, by Corollary 2.1 and the normal form issues investigated above, we can obtain the next result:

[Theorem 3.1]

The start and the individual synchronization scheme are necessary and sufficient to play back any type of composite temporal objects synchronously if null time-intervals are used.

In OMEGA the start synchronization is designed and implemented using a very traditional way, i.e. a shared memory approach. That is, when a playback message is sent to a composite temporal object \(\text{equals}(m_1, m_2)\), then two playback processes P1 and P2 are created, and if, for example, P1 becomes ready to start playback, then it sets the value of a common variable \(v_1\) to 1 and delays execution until the value of a common variable \(v_2\) is set to 1 when P2 is ready to start. Similarly, P2 can also be initiated by P1. Since UNIX is a time sharing system, a delay between the start at the two objects arises, however this delay was not visible as reported in the next section.

The individual synchronization scheme is implemented in OMEGA as follows: Suppose the duration of a video object is \(d\) seconds. A video playback process is created to play it back in \(d\) seconds. Since video data are compressed by MPEG-1, decompression of a frame is first done. Then a presentation subprocess is run to display the decompressed frame on a target device such as an X-window. We define a total processing time of one frame as the sum of the time needed for decompression and presentation. Then, the following individual synchronization scheme holds: The playback process sets time 0 to the start time point of the video object. The first video frame is numbered one, and so on. It is expected that the first frame is displayed by time \(t_1=1/30\) (second) assuming that the video is captured as a sequence of 30 consecutive frames per second. However, the playback processing speed may vary time to time depending on the current cpu performance as well as data transmission rate between main memory and secondary storage. If the total processing time of the first frame is less than \(1/30\) second, then the playback process delays to start the second frame processing until \(t_1\). If the processing of the first frame does not end by \(t_1\), then the playback process delays the next frame processing until the first-frame processing finishes. Suppose the first-frame processing finishes at time \(t_1\) which is greater than or equal to \(t_1\) and less than \(t_1 + 1\) for some \(i\) (\(1 <= i\)). Then the playback process starts the \(i\)-th frame processing. If this frame decompression finishes by \(t_{i+1}\), then its presentation processing is initiated. The presentation processing may finish at time \(t'\) which is greater than or equal to \(t_i\) and less than \(t_{i+1}\) for some \(j\) (\(1 <= j\)). Then the process starts to decompress the \(j\)-th frame at \(t'\). Now, if the \(i\)-th frame decompression does not end by \(t_{i+1}\), then the decompression of the \(k\)-th frame is initiated when that decompression is finished at time \(t''\) where \(t''\) is greater than or equal to \(t_k\) and less than \(t_{k+1}\) for some \(k\) (\(1 <= k\)). The playback process repeats the above scheme until the last frame of the video object is processed. Figure 3.2 depicts how this algorithm works.

One of the characteristics of this algorithm is that every end of decompression time is used to check whether the
succeeded presentation (=repainting) sub process can be started or not, which ensures time-rigorous playback.

Notice that the MPEG 1 standard is adopted to compress video data in OMEGA. Therefore, a video frame is retrieved from the GOP to which it belongs, where GOP is standing for 'group of pictures' which is a storage and an access unit to the database.

### 3.4 Evaluation of the Synchronization Schemes

The purpose of this section is to report a part of the measurement results which we have done for evaluating the synchronization schemes implemented in OMEGA. Notice again that video data are compressed using MPEG-1 standard and are stored in ONTOS DB 3.1, a commercial object-oriented database management system. As we will see in the next section, the video is break down into a series of GOPs. The evaluations are done for the following two cases:

(E1) Checking how the individual synchronization scheme works.

(E2) Checking how the start synchronization scheme works.

To check the former, a video object of 10 second length is prepared, where the frame capturing rate is 1/30 second. Then we measured the time needed to play it back by our software multimedia player (written in C++) under the three different cpu environments: (i) Only this playback process is running, (ii) in addition to this playback process, a simple calculation program is running concurrently, and (iii) in addition to this playback process, a graphics demonstration program is running concurrently, each of which corresponds to the number 0, 26, and 100 of the cpu loading ratio number of Sun SPARCstation 20, respectively.

The result was that the individual synchronization scheme worked without problem in that the time needed to play back the 10 second length video was 10.0002 seconds, 10.02 seconds, and 10.1 seconds in average (we measured ten times) corresponding to the three different cpu loading ratios. The reason why the total time to play back is always a bit (0.0002 second in the first case) over 10 seconds is that even though the presentation (=repainting) process to the last frame starts before 10 second, it takes time to finish it which ends slightly after 10 seconds have passed. Note also that the time necessary to decompress a frame is much shorter than the time necessary to repainting in that the former is about 100 to 200 times faster than the latter. Another interesting point was that the frame rate, i.e. the number of frames displayed in one second decreased corresponding to the increase of the cpu loading ratio, which are the 28.5, 15.6, and 7.0, respectively.

The start synchronization scheme implemented in OMEGA worked also without problem. No start time delay was observed even when four video objects were played back. To evaluate how the individual synchronization scheme works in conjunction with the start synchronization scheme, we measured the playback time needed to play back the composite temporal object equals(v, v), where v represents the 10 second length video which was used in the above measurement. The result was that it required 10.001, 10.03, and 10.1 seconds, corresponding to the increase of the cpu loading ratio mentioned above. However, the number of frames displayed in each window (the window size is set 320
by 240 pixels) were 8.0, 4.4, and 2.0, respectively, which were slow.

Concurrent playback of a video and a sound works without any problem in any CPU environment because audio playback process is so light in comparing with the video playback process. As a conclusion it was observed that both the individual and the start synchronization schemes work as we designed.

4. Storage Structure for Temporal Multimedia Objects

4.1 Determining a Storage Structure for MPEG-1 Video Data

By using MPEG-1 standard to compress video data, we can achieve the reduction of size at the factor of several ten. For example, it requires approximately 50 GB (gigabytes) to store a video of two hour length without any compression, but it is reduced to approximately 1 GB if MPEG-1 is applied. However, MPEG-1 converts a normal video data into a sequence of GOPs which consists of a sequence of several to several ten frames with different natures labeled as picture type I or B or P. The GOPs are the units of random access to video data.

First we investigated which one of the following approaches is reasonable to store MPEG-1 video data in object-oriented databases:

(A1) Introduce a class to define a whole MPEG-1 video data as one object.

(A2) Introduce a class to define one GOP as one object.

(A3) Introduce a class to define each frame of MPEG-1 video data as one object.

The first approach was excluded, because of the size of an object which can be stored in an object-oriented database system. From theoretical point of view, there is no restriction on it. But, from practical point of view, there is a heavy restriction. For example, the object-oriented database management system ONTOS DB 3.1 used to implement OMEGA has a restriction in that its maximum size is approximately the 90% of the size of the swapping area of the UNIX system on which ONTOS is running, because the object is unfolded first on the swapping area, and then sent to the secondary storage. In OMEGA, the swapping area is set to 150 MB and therefore 135 MB is approximately the maximum size of one object which can be stored in the database.

Let us examine the third approach. In this case, it is easily understood that it is impossible to play back the retrieved MPEG-1 video frame by itself if its picture type is B or P. In general, it is necessary to retrieve all other frames from database which consist of the GOP to which the desired video frame belongs. Therefore, if a GOP consists of fifteen frames, then fifteen object retrievals occur. This is time consuming. Also this neglects the features of the MPEG-1 coding algorithm. Therefore, we concluded that this approach should not be taken.

We adopted the second approach because this matches the MPEG-1 nature. In this approach, a GOP object is a unit of storage and retrieval to the database. Any video frame can be played back without retrieving other GOP objects. In principle, a desired frame can be retrieved quickly when the number of frames stored in one GOP object is small. The OMEGA is implemented on a Sun SPARCstation 20 with a SunVideo board, and XIL (X Image Library) is used for MPEG-1 codec. It retrieves a frame from a GOP, but we couldn’t realize any time difference for retrieval between the cases where 30 and 300 consecutive frames are packed in one GOP object. Therefore, we decided to pack fifteen consecutive frames in one GOP object with the corresponding picture-type pattern of IRBBPBBRBBPPPBB.

Figure 4.1 shows the class hierarchy in OMEGA to store MPEG-1 video data, where class OC_Object is the root class provided by ONTOS DB 3.1, class TimeInterval, OmegaTemporalObject, Video which is an abstract class to define a video object, are the classes already introduced in Figure 2.1, and class MPEG-1Video is a class to represent a video content compressed by MPEG-1 algorithm which is a sequence of GOPs, and class GOP is defined to store GOPs.

4.2 Size of an MPEG-1 Video Data

As we investigated in the previous section, video data are stored according to the class hierarchy shown in Figure 4.1. We calculate how big the video data is when it is stored in the above way. Notice that the object size of one GOP object is calculated as the sum of the size of the attribute values of those which are inherited from the classes located higher than class GOP in the class hierarchy shown in Figure
4.1 and the size of the attributes values of those defined in class GOP. In the current implementation of OMEGA, the former is calculated at 120 bytes, and the latter is calculated by the next formula:

\[(5,008) \times \text{number of MPEG-1 frames stored in one GOP object)} \times f \times g\] (byte),

where the number 5,008 is the sum of the size of a variable of type char to store one MPEG-1 frame, which is 5,000 bytes, and the size of a variable of type double to record the duration of the MPEG-1 frame, which is 8 bytes. To store a GOP object in the database, it is necessary to create objects in the upper classes in the class hierarchy so that the GOP object is accessed. In our current implementation, the total size for those objects is 200 bytes. Therefore, the storage space which is required to store a GOP object is given by \[s(f, g) = (200 + 5,008 \times f + 120 \times f / g) \text{ (byte)}\], where \(f\) and \(g\) represent the total number of frames and the number of frames stored in one GOP object, respectively.

### 4.3 Storage Structure for Audio Data

Audio data have different nature from video data in that human is very sensitive to noise in audio environment. In OMEGA, we first implemented sound data as we did for video data. That is, in order to make audio data manageable in a computer, digitization is performed first by sampling the original sound signal, and then quantization is done. By digitization, the sound wave becomes a sequence of bits. We divided this sequence into a piece of one second length, and made them objects. Therefore, in order to play back, for instance a ten second length sound, ten objects were connected to make one ten-second length object. (Notice that \(\text{meets}(\text{meets}(\text{meets}(\text{meets}(\text{meets}(\text{meets}(\text{meets}(s1, s2, s3, s4, s5, s6, s7, s8, s9, s10)\text{ is the composition expression to obtain the ten-second length sound in our framework, where } s1, s2, ..., s10\text{ represent the first, the second, ..., and the tenth piece of sound to be played back.})\text{ In principle there is no problem, but in practice it was difficult to eliminate a click noise occurred at the joint of two consecutive piece of sound units which is very offensive to the ear. Therefore, we abandoned this approach.}

In OMEGA sound data is processed by the sound board attached to the SPARCstation 20. Its sampling frequency is 8 kilo Hertz (KHz), and the number of the quantization level is 256 which is represented using one byte, i.e. one character. Also, the amount of audio data is not huge. For example, the total amount of two hour sound data sampled at 8 KHz and quantized at 8 bits is 37.6 MB. Therefore, we implemented sound data using a variable of type char. By this implementation we can easily access any time point of the sound. For example, if a user wants to play back a portion of a sound data beginning at 10 minutes past from the beginning and for ten seconds, then the character sub string starting from the character numbered 4,800,000 to the character numbered 4,880,000 should be played back. In OMEGA an attribute named \text{soundContents}\text{ of type char} is defined in class \text{Sound} for this purpose. This scheme is working without any problem.

### 5. Conclusions

In this paper we report the object-oriented approach to temporal multimedia database organization, storage, and synchronization. Based on the time interval logic investigated by Allen, we have shown that the normal forms exist for composite temporal objects, where the equals and meets operations are used if null time intervals are used. Then, we investigated how to play back composite temporal objects. It was shown that two synchronization schemes; the individual
and the start synchronization scheme are necessary and sufficient to play back any type of composite temporal objects. These schemes are implemented in OMEGA, an object-oriented multimedia database management system. The evaluation is done, which revealed that our schemes work as we designed. In order to store audio-visual data in OMEGA, we have developed a class hierarchy where a class named GOP is defined so that MPEG-1 video data can be stored.

To improve the current system, the introduction of the QOS (Quality of Service) concept to the multimedia player is necessary. Also, the introduction of an interactive scheduling capability is needed. In the next stage of the OMEGA development, we will investigate the design and implementation of the languages for querying and authoring temporal multimedia objects. In the future, an application-oriented class library will be developed to apply our system easily to the real world use. Then the scalability of OMEGA should be addressed.

[Acknowledgements]
The author expresses his sincere thanks to Mr. H. Kiyomitsu and Mr. K. Nonaka who worked for the OMEGA project. He is also indebted to Dr. F. J. Berberich and the referees of this paper for their helpful comments. This work is partly supported by the Grant-in-Aids for Scientific Research on Priority Area of the Ministry of Education, Science, Culture and Sport of Japan; Grant Number 08244101.

[References]