ELSEVIER

Signal Processing: *Image Communication* 9 (1997) 385–431

SIGNAL PROCESSING:
IMAGE
COMMUNICATION

The MPEG-4 systems and description languages: A way ahead in audio visual information representation

Olivier Avaro[a,*], Philip A. Chou[b], Alexandros Eleftheriadis[c], Carsten Herpel[d], Cliff Reader[e], Julien Signès[f]

[a] *FRANCE TELECOM-CNET, 38,40 Avenue du Général Leclerc, 92131 Issy Les Moulineaux Cedex, France*
[b] *VXtreme, Inc., 701 Welch Road, Palo Alto, California 94304, USA*
[c] *Department of Electrical Engineering, Columbia University, 500 West 120th Street, Mail Code 4712, New York, NY 10027, USA*
[d] *THOMSON Multimedia, Corporate Innovation Research & Deutsche Thomson-Brandt GmbH, Goettinger Chaussee 76, 30453 Hannover, Germany*
[e] *SAMSUNG Semiconductor, Inc., 3655 North First Road, San Jose, CA 95134-1713, USA*
[f] *FRANCE TELECOM-CCETT, 4, rue du clos Courtel, BP 59, 35512 CESSON SEVIGNE CEDEX, France*

Abstract

The state of the standardization of the Systems part of ISO/IEC JTC1/SC29/WG11 (MPEG-4 Systems) as specified early 1997 is presented. First, a rationale and the architecture of the complete Systems are described. Based on the rapid technological advances in software and in hardware, the MPEG-4 Systems provides for a framework for the integration of natural and synthetic streamed and synchronised media. The different fields of interest in Systems are then developed from the description of audiovisual scenes to the definition of the multiplex. The programmability of the standard is described for composition and decompression and the language adopted by MPEG-4 on purpose of syntax description is fully detailed. Finally, a conclusion and the future evolutions of the specifications are presented. © 1997 Elsevier Science B.V.

Keywords: MPEG-4 systems; Composition; Audiovisual scene description; Multiplex; Synchronization; Buffer management; Interaction; APIs; Programmability

1. Introduction

The fundamental goal of standardization has always been to provide universal interoperability. An adjunct to that has also been to consider the cost of implementation, so a secondary goal has been to provide universal affordability. In many cases, notably the case of audiovisual data coding, this has meant that a very specific method has been chosen for standardization, because the lowest cost has been achieved with a fixed-function solution. The penalty has been a limit to flexibility, both in the context of being application-specific, and in extendibility to future applications and operating environments.

* Corresponding author.

0923-5965/97/$17.00 Copyright © 1997 Elsevier Science B.V. All rights reserved.
PII S0923-5965(97)00027-1

This situation is currently changing, under the influence of rapid technological advances, and declining price points. Most notably, the clock speeds of CPUs and DSP chips are increasing very quickly with consequent increase in computational power. As a result, it is becoming cost effective to solve traditional problems in programmable systems. Indeed, the future is in software. Changes are also occurring in display technology, though at a slower place, and there is a trend toward higher resolution, progressive scan, high refresh-rate displays. On the audio side, spatial audio is becoming important for entertainment applications such as games.

Such developments provide a whole new degree of freedom for designing audiovisual communication and storage systems. Being software based, there is less compelling need to standardize a specific algorithm, so a set of algorithms covering a range of applications can be considered. Also the set can be extended in the future. The changes in display technology mean there can be a decoupling between the coded representation of the data and the presentation of the data. It is interesting to note that notwithstanding the advanced technology deployed in the MPEG1/2 standards, the input and output data are still the same analog TV format invented almost 60 years ago! The important point to note is that the data structure of the coded data has been forced in the past to be that of the presentation format. The MPEG1/2 syntax is composed explicitly of coded frames, but these data structure elements are not interesting in terms of comprising real-world objects.

Standards are composed of 'normative requirements', and in the case of the MPEG1/2 standards, it should be noted that most of the normative requirements are expressed in the syntax and semantics of the standardized bitstream. The elements not expressed in the syntax concern the (fixed) data structure and the (fixed) coding algorithm. It is of course tempting to propose that all normative requirements be expressed in the syntax, and then to define the syntax (with semantics) as a programmable communication language, not a rigid specification.

MPEG-4 takes advantage of these underlying developments. It provides a coded representation of real-world audiovisual objects, as opposed to presentation-based images of those objects. It provides a truly generic language for the communication of audiovisual objects. This then establishes a very flexible environment that can be customized for specific applications and that can be adapted in the future to take advantage of new developments in coding technology.

With all the normative requirements expressed in the bitstream, with an object-based data structure, and with a software-based implementation, a user-driven, fully interactive environment is possible. Imagine that the user can access an audiovisual 'scene' that is three-dimensional (both for audio and video), that has a spatial extent and spatial resolution far higher than the presentation device used to access it, and that is composed of audiovisual objects, animated in real time. Further imagine that some objects are being generated by the encoder, but that other objects may have been generated in the past and have been downloaded, or are being generated locally. Interactivity can now take two forms: first the ability to move the presentation window around this scene, and zoom in and out of it, and second to interact with the audiovisual object themselves. This is the environment that MSDL (MPEG-4 Systems and Description Languages) enable.

2. MPEG-4 Systems – the big picture

2.1. Scope

MPEG-4 Systems describes a coherent framework within which the requirements [1] of MPEG-4 can be satisfied. It provides a foundation on which other MPEG-4 areas of work can be built. These areas include:
- the traditional MPEG areas of work: audio representation [2], video representation [3], multiplexing and synchronization [5], and
- new areas of work: composition of audiovisual information [4, 5], representation of synthetic media [4], and flexibility [5, 6], that are now required to achieve the MPEG-4 goals.

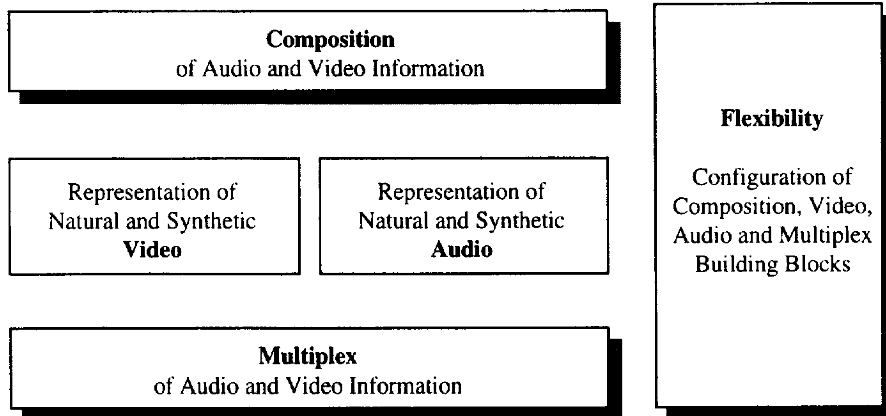


Fig. 1. MPEG-4 areas of work.

Flexibility is a feature that can be applied orthogonally to any of the previous areas (flexible multiplexing, flexible audiovisual representation, flexible composition). Flexibility enables description and configuration of the system.

MPEG-4 Systems specifies a system for communicating audiovisual information, that is, the representation of physical or virtual objects that can be manifested audibly and/or visually. At the encoder, audiovisual information related to a physical scene is compressed, error protected if necessary, and multiplexed in one or more coded binary streams that are transmitted. At the decoder, these streams are demultiplexed, error corrected, decompressed, composited, and presented to the end user. The end user is given an opportunity to interact with the presentation. Interaction information can be processed locally, or transmitted to the encoder.

The current section presents the functional description of the MPEG-4 Systems. Previous MPEG standards were mainly hardware driven. MPEG-4 requires extensibility and therefore needs a software architecture. Flexibility is first presented, followed by the main elements of the software architecture and the associated communication model. Then, the overall structure of an MPEG-4 terminal is described.

The areas of work related to the MPEG-4 Systems (shaded elements in Fig. 1): composition (flexible and non-flexible), flexible decoding (flexible decompression and syntactic decoding) and multiplexing are then detailed in Sections 3–6.

2.2. Flexibility

MPEG-4 specifies a system that allows for communicating audiovisual information. In order that the information transmitted or stored can be understood by the decoder, a template for this information has to be known at the receiver side. In its previous standards, MPEG defined rigid a priori known templates for the transmitted information. The MPEG-4 standard requires a more flexible representation of these templates, so that they can be transmitted and allow for configuring the receiving system. In support of flexibility, MPEG-4 defines two different types of terminals: non-flexible and flexible.

2.2.1. Non-flexible terminals

2.2.1.1. Definition. Non-flexible terminals have a small degree of decoder flexibility. This flexibility corresponds to the flexibility achieved by previous MPEG terminals regarding audiovisual information representation: it is achieved with the use of switches or selectors in the binary stream, which are basically n-ary elements that select which of n pre-defined templates will be used for the coming information. This allows, for

example, the choice of a pre-defined standardized configuration. The switch flexibility is extended in MPEG-4 to other areas of work such as composition and multiplexing.

2.2.1.2. Specification. Non-flexible terminals are based on a set of standardized elements called tools that are combined in pre-determined standardized ways to provide a set of standardized algorithms and profiles [1]. A given standardized combination fully specifies the information template for the coming information. Choices between templates can be achieved with the use of switches.

2.2.1.3. Limitations. This type of flexibility was sufficient within the scope of previous MPEG standards, and has the attractive feature of being simple, practical and bit efficient. It poses significant limitations, however, within the context of MPEG-4:

1. There is no explicit representation of the information template. Therefore, all the possibly needed configurations have to be specified during the standardization process. As a result, terminals will not be able to communicate if the information template is not known a priori, even if they have very compatible capabilities.
2. The switch flexibility is hard to manage when the number of profiles and tools increases. At this stage, it is not known whether or not the switch flexibility will be sufficient to efficiently cover the various applications targeted by MPEG-4.
3. The evolution of the standard is hard to manage: switches have to be anticipated during the standardization process, and future needs are hard to predict. Therefore, new applications may require a new round of standardization, even if they use only already standardized tools. Moreover, introducing new tools requires the explicit standardization of new algorithms and profiles.

2.2.2. Flexible terminals

2.2.2.1. Definition. MPEG-4 defines an enhanced flexibility mode in the representation of information templates. A terminal supporting this mode is called a flexible terminal. To represent the information templates, MPEG-4 decided to use classes (in the object-oriented sense of the word) and to encapsulate all audiovisual information in audiovisual objects (AV objects) instantiated from audiovisual classes (AV classes).

More generally, to have a consistent structure, the MPEG-4 specification adopts an object-oriented approach for the description of all elements of its architecture. Hence, the object-oriented terminology is adopted as much as possible in the rest of this document.

In flexible terminals, flexibility is achieved by the transmission of new classes, defining therefore new templates for the transmitted information. The class definition defines both the data structure and the methods that are used to process the data components. The downloaded classes may be related to the various parts of the MPEG-4 flexible terminal: new composition tools, new decoding or demultiplexing algorithms, etc.

The overall architecture for communicating AV objects and AV classes is as follows. Before the AV objects are transmitted, the encoder and decoder exchange configuration information. The encoder determines which classes of algorithms, tools, and other objects are needed by the decoder to process the AV objects. Each class of objects is defined by a data structure plus executable code. The definitions of any missing classes are downloaded to the decoder, where they supplement or override existing class definitions installed or pre-defined at the decoder. As the decoder executes, new class definitions may be needed. In such a case, the decoder can request that the encoder downloads specific additional class definitions. The additional class definitions may be downloaded in parallel with the transmitted data. The above aspects are illustrated in Fig. 2. Such a mechanism provides the decoder with the flexibility and extensibility desired in MPEG-4.

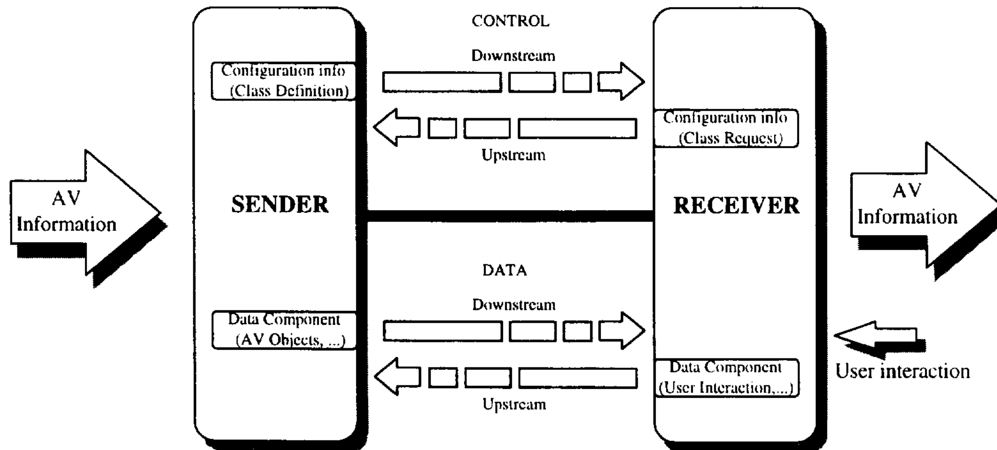


Fig. 2. Architecture of flexible terminals.

2.2.2.2. Specification. A practical specification of flexible terminals needs:
- a standardized class hierarchy: definition of classes, their associated methods, and their interfaces or APIs (Application Programming Interfaces), and
- a transmission format for downloaded classes.

To transmit classes, MPEG-4 investigates both the use of machine independent bytecode (like the Java bytecode) and the use of scripts (like VRML binary scripts). The two approaches are complementary.

The bytecode approach has the power of a general programming language. It will be used when few assumptions can be made on the templates to describe (e.g., the description of a new decoding algorithm, the description of a complex interpolation function). Within the context of the MPEG-4 standard, only the executable equivalent of the language has to be specified; it is this data that eventually are transmitted to the terminal. In the current Verification Model [6], the Java language is used for test and validation purposes, and the bytecode approach would imply the standardization of the Java Virtual Machine. However, the specification is still under development and other suitable object-oriented languages that fulfill the MPEG-4 requirements (including real-time constraints) may be considered.

Scripts are a less powerful but more concise approach to describe information templates. They may be used when the description of a new class does not need the expressiveness of a programming language like C++ or Java. In particular, a wide range of composition information templates can be described this way. An extension of VRML 2.0 is the current reference to test and validate the script approach for describing composition templates. Here also, only a binary equivalent of the script will be standardized and not the textual format.

2.2.2.3. Limitations. From a computational point of view, the language used within flexible terminals has the capability of a full Turing machine and, therefore, any algorithms can be described with this approach. However, due to the machine independence requirement for this language, the description of complete new algorithms could be inefficient, and proprietary code may significantly outperform a decoding system described with this language. At this point, strategies to ensure high-quality performance for compliant terminals are still under development.

A first element of these strategies is to limit bytecode flexibility to MPEG-4 areas of work where the technology allows real-time performances, at the date MPEG-4 becomes a standard. The priority has therefore been given to Composition and AV Objects behavior, where satisfactory results can be obtained. This area is also where the market has shown an increasing interest.

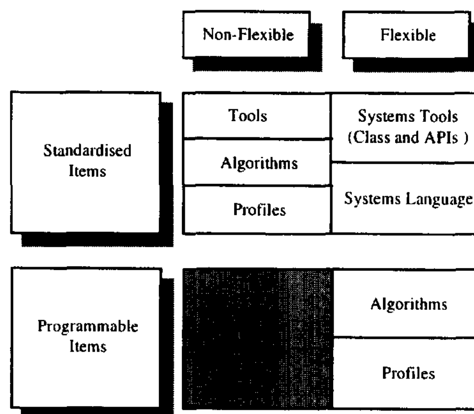


Fig. 3. MPEG-4 flexibility.

A second element is envisioned that compliant terminals would only have to provide performances for standardized tools, or standardized sets of tools. Based on the tools APIs and on the downloaded bytecode, new algorithms could be described with satisfactory performances. Flexibility could then be extended to other MPEG-4 areas of work such as audio and video algorithms configuration.

Finally, depending on the evolution of the technology and market needs, flexibility could be extended to downloading of tools in later phases, but is currently out of the scope of MPEG-4.

A pictorial representation of the terminal's capabilities within the current scope of MPEG-4, with respect to standardized and programmable components, is given in Fig. 3. Tools, algorithms and profiles refer to any of the previously defined MPEG-4 areas of work.

2.3. *Objects in the MPEG-4 Systems context*

The MPEG-4 specifications adopt an object-oriented approach for the description of all the elements of its architecture. Basic object-oriented terminology can be found in [14]. This section gives an overview of the more relevant MPEG-4 objects.

2.3.1. *MPEG-4 objects*

MPEG-4 objects are objects in the object-oriented terminology, that is, entities that combine a data structure (defining the object's state) with a set of methods (defining the object's behavior). A method is an executable procedure associated with an object that operates on information in the object's data structure. Classes are templates for objects. MPEG-4 standardizes a number of pre-defined classes. This set of classes is called the MPEG-4 standard class library. In flexible terminals, based on this library, the user or encoder will be able to produce new encoder-defined classes and instantiate objects according to these class definitions.

Classes in general, and MPEG-4 classes in particular, are organized hierarchically. The root of MPEG-4 classes is named `MPEG4Object`. The hierarchy specifies how a class relates to other classes, in terms of inheritance, association or aggregation. Graphical methods to represent this hierarchy are commonly used. The OMT (Object Modeling Technique) notation has been chosen within the context of MPEG-4 Systems.

2.3.2. *Audiovisual objects*

`AVObject` is an MPEG-4 class derived from `MPEG4Object`. The `AVObject` class is used to represent natural or synthetic objects that can be manifested audibly and/or visually. AV objects are generally hierarchical, in the sense that they may be defined as composites of other AV objects, which are called

sub-objects. AV objects that are composites of sub-objects are called compound AV objects. All other AV objects are called primitive AV objects. The top-most object in the hierarchy is called the Scene. The procedure of building a compound AV object from other AV objects is called composition. All AV objects have a method to composite the visible or audible representation of an `AVObject` at a given point in time, the `render` method.

AV objects, like other objects, have a behavior defined by their methods. In particular, AV objects have an audiovisual behavior. This behavior, which defines how the object is manifested, can be programmed in continuous time (e.g., a bouncing ball), can be updated with input data like user events or input streams (e.g., a moving picture, a game character), or both. Each `AVObject` has a `handle` method to specify the behavior of the object in case of external events. Each `AVObject` may be connected to one or more data streams to receive input data.

2.3.3. Process objects

AV objects may contain embedded or streaming data. In general, these data are stored or transmitted in a compressed format that cannot be directly used for the rendering procedure. Signal processing operations thus have to be performed to reconstruct the audiovisual information that will subsequently be rendered. Each distinct processing operation (e.g., linear transformation, prediction, filtering) is modeled as a process object. These objects have an `apply` method taking parameters like AV objects or data streams. They define processing operations used to modify other MPEG-4 objects.

MPEG-4 defines a language, the syntactic description language (MPEG-4 SDL), to describe the precise binary syntax of an AV object's compressed information. This language is used to describe the syntactic representation of objects in an integrated way with the overall class definitions.

2.3.4. Stream objects

An AV object reads the information it needs to update its state from `InputStream` objects. These objects constitute the interface to MPEG-4 entities, named elementary streams. Data and control information (description of classes, configuration information, etc.) for a Scene may be carried in several concurrent elementary streams. It is the purpose of the MPEG-4 multiplexer to control the delivery schedule of these elementary streams and, hence, real-time decoding with limited delay, jitter and buffer requirements. For transport through a network, or storage, these streams are packed into entities called logical channels and may be multiplexed in a single data stream. Logical channels are virtual links between a sending and a receiving multiplexer entity and are characterized by the quality of service parameters negotiated when opening the channel. AV objects whose data are represented in individual elementary streams may be subjected to bitstream editing. This allows, e.g., extracting all information related to such an AV object by extracting their elementary streams without the need for any decoding.

2.4. Communication structure

The MPEG-4 specification adapts the traditional two-terminal model of a communication system, in which one terminal (the encoder) encodes a sender's message and transmits it across a communication channel, and the other terminal (the decoder) decodes the message and presents it to a recipient.

This traditional model is augmented by allowing multiple channels and by allowing 'upstream' channels (outgoing from the decoder) in addition to 'downstream' channels (incoming to the decoder). In particular, point-to-point, multipoint-to-point, and point-to-multipoint communications may be supported in this way. The MPEG-4 specification assumes that the primary flow of information is downstream. Whenever they are available, upstream channels may be used for informing the encoder of the decoder's capabilities, negotiating the transmission profile, requesting additional information from the encoder, reporting errors, feeding back user controls, and so forth.

Streams (either downstream or upstream) are also classified either as control or data streams. Control streams carry control information, such as connection setup, profile, and class definition information. Data streams carry all other information, principally coded audiovisual objects, but possibly also information such as Huffman tables, filter coefficients, or interaction information.

The communication session is partitioned into five phases that are run sequentially or in parallel:

- *Connection*: In this phase, using services of the underlying transport layer, a connection is established between the encoder and the decoder by exchanging *Encoder Start of Session* (*ESS*) and *Decoder Start of Session* (*DSS*) information. Either encoder or decoder may initiate the connection.
- *Configuration*: In this phase, the encoder and decoder agree on a profile (set of tools and algorithms) by exchanging *Encoder Configuration Information* (*ECI*) and *Decoder Configuration Information* (*DCI*) messages describing their capabilities (implemented classes, computational capabilities, etc.).
- *Learning*: In a communication involving flexible terminals, one or more class definitions follow one after another in sequence. A *Class Definition* (*CD*) specifies both a data structure and executable methods. It is possible that somewhere in the chain of actions resulting from a call of a method at the decoder side, a reference to an undefined class is made. This generates error at the decoder, unless upstream control is available. In that case, the decoder can issue a *Class Definition Request* (*CDR*) to the encoder, load the class, and proceed.
- *Transmission*: In this phase, coded audiovisual objects (and possibly other information) are placed in the data streams by the encoder and removed from the data streams by the decoder. The information format is determined entirely by the current classes in use.
- *Disconnection*: In this phase, the connection between the encoder and the decoder is broken by exchanging *Encoder End of Session* (*ESS*) and *Decoder End of Session* (*DES*) information.

These communication phases are illustrated in Fig. 4, along with the collection of streams established in general between the encoder and the decoder, and the typical flow of information in those streams.

2.5. Processing stages in the MPEG-4 terminal

An MPEG-4 terminal is a system that allows presentation of an interactive audiovisual scene from audiovisual coded information. It can be either a standalone application, or part of a multimedia terminal that needs to deal with MPEG-4 audiovisual information representation, among others.

The basic operations performed by such a terminal are the following: a description of the audiovisual scene is sent downstream to the decoder. The scene is instantiated. Data representing AV objects are demultiplexed and then processed by the relevant decoders. The decoded AV objects are then composited and rendered in order to be presented on the terminal presentation device. These basic operations are depicted in Fig. 5.

In the current implementation, the operation performed by such a viewer is the following: one class at the viewer side defines a 'Main' AV class derived from the class `AVObject`. This class is instantiated as a 'main' AV object. This 'main' object is the AV scene, i.e., the top-most AV Object in the hierarchy. The main object's `render` method (inherited from `AVObject`) is then repeatedly executed. This operation produces primitive AV objects, such as `Image` and `Waveform` objects, which will then be presented to the user through output devices.

The main object's `render` method is called once for each audiovisual frame that the viewer wishes to present. This method invokes other methods and other objects as for example:

- calls to `render` methods of other AV objects related to the 'Main' class,
- calls to `apply` methods of process objects to recover image and audio waveform objects from compressed data,
- calls to syntactic decoding methods to extract compressed data from AV objects data streams, and
- calls to demultiplexing methods to extract elementary data streams from logical input channels.

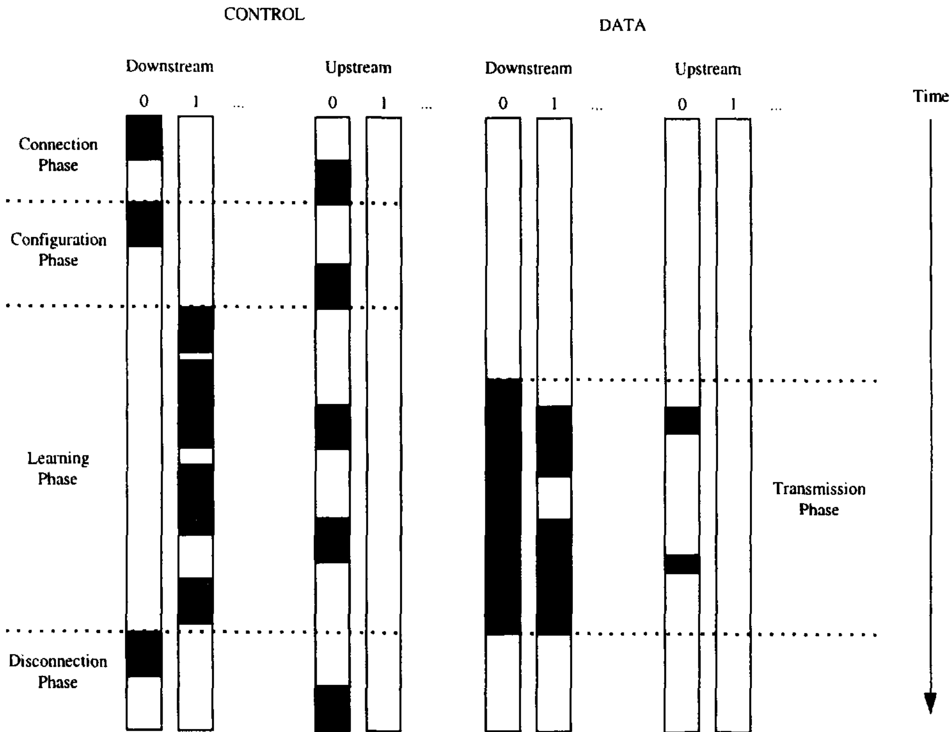


Fig. 4. Information streams and communication phases.

In this way, repeated calls to the main `render` method cause the input information to be decoded and rendered. Basic functions related to the terminal behavior are now described.

2.5.1. Composition and rendering

Composition organizes the AV objects in the scene. Each AV object has a local coordinate system, and the local coordinate system of the main AV object (the scene) is also the world coordinate system. To composite is to place all AV objects in the scene, i.e., to define the mapping from the AV objects' local coordinate systems to the world coordinate system. Rendering projects the scene onto one or more audiovisual frames, for subsequent presentation to the user. Composition is currently implemented by calling the `render` method of AV sub-objects, each in its own composition context.

In non-flexible composition, the composition information is transmitted (retrieved) at the receiver with a given syntax and semantics (e.g., transformation parameters and timing information coded with a given number of bits). This syntax is specified in Section 3 for the 2D case.

More complex scene descriptions are needed for some applications. In support of these applications, MPEG-4 defines a flexible composition mode consisting in a rich scene description format for composition of streamed and synchronized audiovisual information. In this flexible mode, configuration information (classes or scripts), can be sent to the receiver to complete composition functionalities and describe complex or custom composition.

2.5.2. Decompression

Decompression recovers the AV objects' data from their encoded formats, and provides the composition layer with this information. Decompression is currently performed by successive calls to the `apply` methods of `ProcessObjects` to recover the needed image and audio waveforms from the input streams.

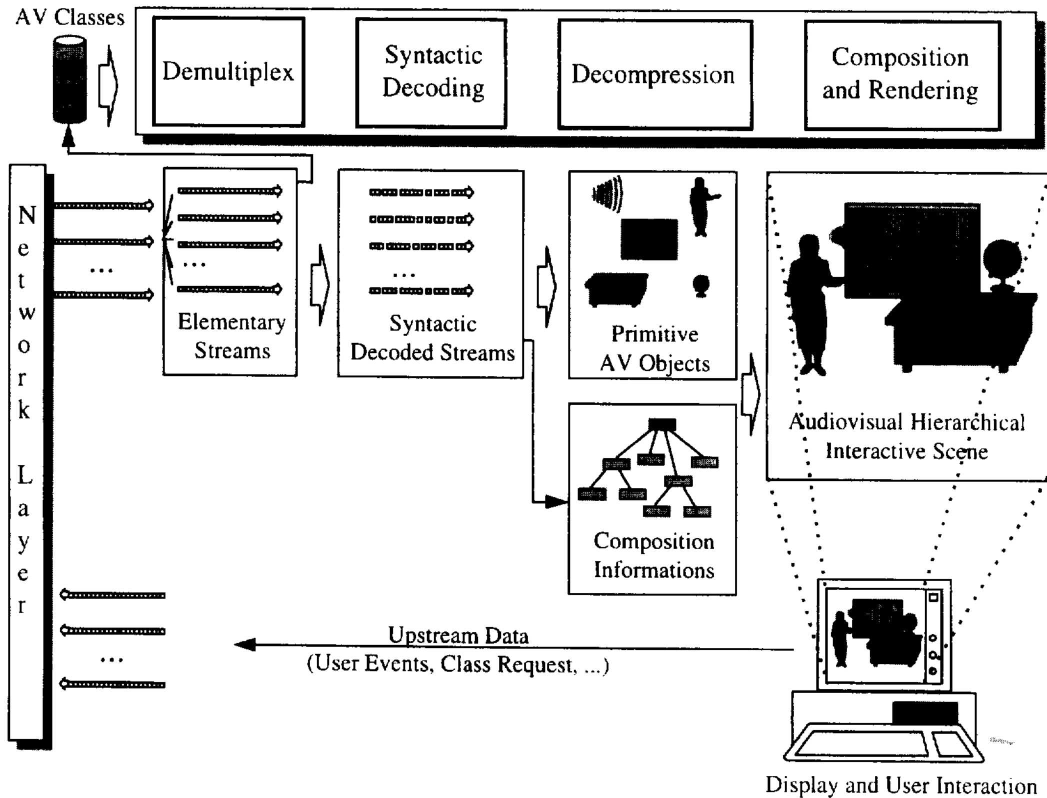


Fig. 5. Processing stages in the MPEG-4 terminal.

To the non-flexible decompression algorithms, specified by the Video, Audio and SNHC sub-groups, will correspond specific standardized `ProcessObjects`. These objects can then be used to construct for example new compound interactive scenes.

For a given AV scene, the best suited compression format for a given AV object in this scene may change, during the scope of the application, or from one application to another. Therefore, different coded representations for a given AV object could be used. Such a flexibility can be achieved by defining and downloading `ProcessObject` classes that will replace the ones previously called by the decompression procedure. The MPEG-4 Systems thus allow a separate description of the decompression and composition procedures.

2.5.3. Syntactic decoding

Syntactic decoding is the process of recovering basic semantic entities (such as quantized values and time references) from elementary data streams and mapping them to the internal machine representation.

In order to allow separate and efficient configuration of syntactic decoding, the MPEG-4 specification made the choice to decouple as much as possible the definition of the bitstream syntax (including entropy coding) from the subsequent operation of decompression (including de-quantization) and rendering.

To achieve this, MPEG-4 defines a language, the syntactic description language (MPEG-4 SDL), to describe the precise binary syntax of an AV object's compressed information. This language is used to describe the syntactic representation of objects in an integrated way with the overall class definitions.

2.5.4. Multiplexing

Multiplexing provides the interfaces between the network and storage media and the other system layers. Its basic function is to recover elementary data streams from logical downstream channels and to multiplex upstream data in logical upstream channels. These elementary streams may carry either object data or control information related to objects or to system management. The multiplex layer provides system control tools, e.g., to receive new classes, to manage synchronization, to recover the system time base, and to manage system buffers.

2.6. Evolution of MPEG-4 Systems

The MPEG-4 Systems is the result of needs and convergence in the technology of audiovisual representation. The three key technologies: audiovisual representation (compressed, streamed and synchronized, natural and synthetic, 2D and 3D audiovisual objects), audiovisual composition (2D and 3D worlds composed of objects having behavior and responding to interaction) and programmability (downloading of software modules in a platform independent bytecode) have to converge in a consistent way in order to provide a satisfactory and integrated solution to market needs.

This integrated solution does not yet exist. Indeed, the three previous key technologies are just becoming sufficiently mature to acknowledge their inadequacy to solve alone the entire problem. To provide the place where the best technical solutions in each of these fields can be integrated in a consistent way, the current MPEG-4 Systems specification still needs to evolve: the description of composition information using scripts needs to be integrated with the current APIs, which itself need to be completed, a possible harmonization of 2D and 3D representations could be done, events models have to be developed.

To achieve this, the Verification Model methodology, applied with success within the context of Audio and Video algorithms specifications, is now set up for Systems activities [6]. This methodology provides a framework for collaborative development and software sharing within the MPEG community. Evolutions are driven by the results of experiments made on the Verification Model and ensure a constant validation and evaluation of the specifications.

3. Composition – preparation for presentation

3.1. Scope

In the graphic arts, 'compositing' refers to the process of creating a single image from multiple overlapping images, by blending them together in an appropriate way.

The Compositor, in MPEG-4, performs compositing in this traditional sense (using digital images of course), but it also does much more. In particular, it coordinates the process of composition, in which two- and three-dimensional time-varying visual objects are rotated, scaled, and positioned with respect to one another in space and in time, and it handles the process of rendering, in which the composited visual objects (at a given instant in time) are 'imaged' or 'rasterized' to produce image fragments. The resulting image fragments are finally blended to produce the desired image. In addition to the above, the Compositor in MPEG-4 performs the analogous composition, rendering, and blending (i.e., mixing) operations for audio objects.

The effects that can be produced with the MPEG-4 Compositor include the following:

- allowing user navigation through a time-varying 3D environment;
- placing a scaled-down video in the corner of another video (i.e., picture-in-picture);
- overlaying text, such as subtitles or scrolling credits, on top of a movie;
- defining moving 'hot' objects in a scene that respond to user interaction;
- providing audible or visual feedback (e.g., color changes) as the result of user selection;
- integrating video, still images, graphics, animated graphics (e.g., sprites) and audio into a single audiovisual presentation;

– positioning buttons, pull-down menus, and other user-interface elements on top of a presentation;
– modulating point sources of audio to appear spatially localized (e.g., becoming louder when approached);
– stereo vision, multichannel audio;
and so forth.

Coordinate systems play a central role in producing these effects. Coordinate systems are the 'handles' by which audiovisual objects can be rotated, scaled, translated, delayed, and otherwise manipulated. For this purpose, every audiovisual object in MPEG-4 has an associated 'local' coordinate system. The pose of an audiovisual object within a scene is determined by a coordinate transformation from the object's local coordinate system into the scene's 'global' coordinate system.

In MPEG-4, coordinate systems generally have four dimensions: three spatial dimensions (x, y, z) and one temporal dimension (t). This is sometimes denoted '3D+T'. By convention, the spatial portion of the coordinate system is right-handed,[1] and the temporal portion of the coordinate system has its origin at the beginning of the audiovisual object, with t increasing towards the end of the object.

Transformations between coordinate systems are generally affine (i.e., linear with a translation component). Thus affine coordinate transformations mapping (x, y, z, t) to (x', y', z', t'), can be described in matrix notation, as follows:

$$\begin{bmatrix} x' \\ y' \\ z' \\ t' \end{bmatrix} = \begin{bmatrix} A_{xx} & A_{xy} & A_{xy} & 0 \\ A_{yx} & A_{yy} & A_{yz} & 0 \\ A_{zx} & A_{zy} & A_{zz} & 0 \\ 0 & 0 & 0 & A_t \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ t \end{bmatrix} + \begin{bmatrix} B_x \\ B_y \\ B_z \\ B_t \end{bmatrix}.$$

Note that the extra zeros in the matrix force the spatial and temporal components to be separable, in the sense that spatial components cannot 'leak' into the temporal component, and vice versa. Similarly, other components may be constrained. For example, when full 3D manipulation is impractical, coordinate systems may be restricted to '2.5D + T', which means that coordinate transformations take the form

$$\begin{bmatrix} x' \\ y' \\ z' \\ t' \end{bmatrix} = \begin{bmatrix} A_{xx} & A_{xy} & 0 & 0 \\ A_{yx} & A_{yy} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & A_t \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ t \end{bmatrix} + \begin{bmatrix} B_x \\ B_y \\ B_z \\ B_t \end{bmatrix}.$$

In this case, transformations in the x–y plane cannot leak into the z dimension, and vice versa. The z dimension is provided only for non-scaleable translations in depth.

In general, a scene has temporal extent, and is composed of audiovisual objects in various spatio-temporal poses determined by coordinate transformations. The parameters of these transformations (and hence the poses) may change continuously and smoothly over time. Furthermore, the audiovisual objects in a scene may disappear and reappear, and new ones may be introduced, at any positive instant in time. Thus a scene and its objects have continuous-time behavior, regardless of any discrete units used to sample, code, or present the objects. This continuous-time model of object behavior makes it possible to decouple the presentation frame rate from any underlying sampling and coding rates, so that audiovisual objects with differing underlying sampling or coding rates can be presented together. It also makes it possible for presentation frame rates to vary from decoder to decoder, or even vary in time within a decoder, so that the presentation can scale to the computational capacity of the decoder.

The pose and rendering properties of each audiovisual object in a scene as a function of time may be described by a program or may be interpolated from sampled data, or both. Such information is called

[1] In a right-handed coordinate system, if the x–y coordinate plane is viewed such that x increases from left to right and y increases from bottom to top, then z increases towards the viewer.

composition information. In non-flexible terminals, this information takes the form of coded parameters. In flexible terminals, this information takes the form of executable instructions, which trigger operations through the API. Non-flexible composition is described next, followed by flexible composition.

3.2. The non-flexible approach to composition

The non-flexible approach to composition can vary significantly, depending on the Systems profile. This section describes a simplified version of the non-flexible composition syntax currently specified in the Systems Working Draft [5]. It should be taken as an example of the kind of Systems profile for composition that can be achieved with non-flexible terminals.

In this non-flexible syntax, the pose and rendering properties of each audiovisual object in the scene can be established, and can be updated occasionally or even continually.

Whenever composition information (i.e., pose and rendering properties) for an audiovisual object must be established for the first time, or changed, the composition information is coded and transmitted in an elementary stream dedicated for this purpose. The composition information is tagged with a code for the audiovisual object to which it should apply, and the information is timestamped to indicate the time at which the information is to take effect, i.e., the time at which it is to supersede old information for that audiovisual object, if any. Composition information may be sent once, and never updated, or it may be updated nearly continuously (at the presentation frame rate, for example).

The composition information for an audiovisual object, as illustrated in Fig. 6, consists of first one bit to indicate whether or not the object is visible (or audible). If the object is invisible (or inaudible), then no further information is included. This bit can be use to hide temporarily transmitted AV objects. Otherwise, five bits are used to code a composition order parameter, which indicated the order in which the audiovisual object will be rendered relative to the other objects in the scene. Each object should have a unique composition order number. The remaining bits are used to code the parameters of the coordinate transformation that maps the object's local coordinate system into the scene's coordinate system. This is accomplished using one bit to indicate 2D or 3D, a few more bits to indicate how many parameter sets are coded, and finally ten bits or more for each of the coded parameters. The parameters are ordered by their importance (e.g. the x and y translation parameters, B_x and B_y, are the most important), so that the *number* of parameter sets coded, along with the 2D/3D flag, determines *which* parameters are coded.

Thus initial poses can be established for all objects that are initially visible in a scene, by transmitting for each object an object identifier, a timestamp, a visibility bit, a composition order parameter, and some number of coordinate transformation parameters. The visibility, composition order, and pose of any audiovisual object can be changed at any time.

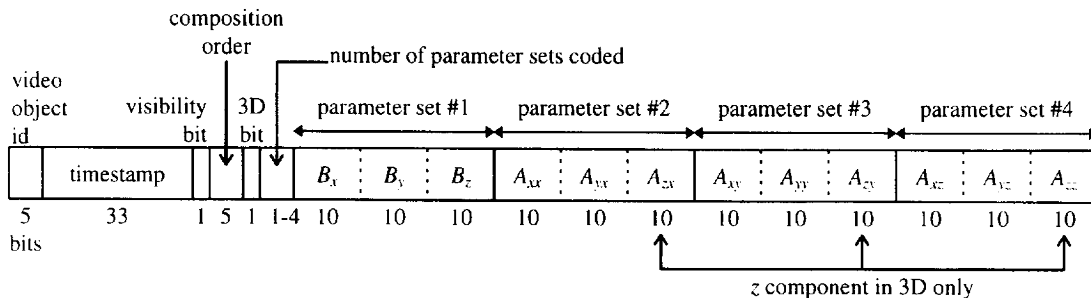


Fig. 6. Example of a non-flexible composition syntax.

Between updates, the composition information is held constant in this non-flexible syntax. This is a piecewise-constant, continuous-time model, in which the composition information is defined for all positive time. Likewise, audiovisual objects themselves are defined for all positive time (although their values may be temporally interpolated from an underlying discrete-time representation). Thus it is possible to render (onto an output frame) a scene composed of audiovisual objects, at any positive instant in time. Indeed, MPEG-4 does not specify a presentation frame rate, and it is not necessary to do so.

For brevity, the discussion now is restricted to *visual* objects. The appearance of a visual object at the time it is rendered onto a video frame for presentation is determined by a number of factors, including both the object's composition order and its depth, or distance from the projection plane. The depth information is used to determine the appearance of an object in the presence of other opaque occluding objects. However, for objects at exactly the same depth, and for semi-transparent objects at any depth, the order in which the objects are blended onto the video frame is also needed to determine the objects' appearance. This order information is given by the objects' composition order parameters.

Rendering of a visual object is in general accomplished by projecting the object onto the projection plane, resampling the object to match the pixelization of the target video frame, computing the color and transparency of each object pixel (which may vary according to the object's orientation, surface normals, and lighting conditions), and finally blending each object pixel with its corresponding target pixel, according to the object's blending factors, if the object pixel is not hidden by a previously rendered, opaque object. Of course, much of this procedure is simplified in the '2.5D + T' case.

Color, transparency, and blending factors are properties of visual objects. Color is specified by a triple, (c1, c2, c3), which along with a colorspace (YUV, RGB, XYZ, etc.) determines the object's color in a standard component color model. Transparency is specified by a single component, alpha. The color components, and alpha, take values in the unit interval [0, 1], although they are usually uniformly quantized to 8 bits of precision. Blending factors are the 4-dimensional vectors ν and μ by which the components (c1, c2, c3, alpha) are multiplied component-wise in both the source image N (e.g., a rasterized object) and the destination image M (e.g., the current output video frame), before adding to form a new destination image P (e.g., the new output video frame), as follows:

$$\mathrm{P.c1} = \mathrm{clip}(\mathrm{N.c1} * \nu.\mathrm{c1} + \mathrm{M.c1} * \mu.\mathrm{c1}),$$

$$\mathrm{P.c2} = \mathrm{clip}(\mathrm{N.c2} * \nu.\mathrm{c2} + \mathrm{M.c2} * \mu.\mathrm{c2}),$$

$$\mathrm{P.c3} = \mathrm{clip}(\mathrm{N.c3} * \nu.\mathrm{c3} + \mathrm{M.c3} * \mu.\mathrm{c3}),$$

$$\mathrm{P.alpha} = \mathrm{clip}(\mathrm{N.alpha} * \nu.\mathrm{alpha} + \mathrm{M.alpha} * \mu.\mathrm{alpha}).$$

Here, 'clip' means that the computed elements of the destination image, e.g., P.ci, are restricted to the unit interval [0, 1].

The blending factors ν and μ can be independently specified to be one of the following combinations:

ZERO = (0,0,0,0),
ONE = (1,1,1,1),
DST_COLOR = (M.c1,M.c2,M.c3,M.alpha),
SRC_COLOR = (N.c1,N.c2,N.c3,N.alpha),
ONE_MINUS_DST_COLOR = (1,1,1,1) − (M.c1,M.c2,M.c3,M.alpha),
ONE_MINUS_SRC_COLOR = (1,1,1,1) − (N.c1,N.c2,N.c3,N.alpha),
SRC_ALPHA = (N.alpha,N.alpha,N.alpha,N.alpha),
ONE_MINUS_SRC_ALPHA = (1,1,1,1) − (N.alpha,N.alpha,N.alpha,N.alpha),
DST_ALPHA = (M.alpha,M.alpha,M.alpha,M.alpha),
ONE_MINUS_DST_ALPHA = (1,1,1,1) − (M.alpha,M.alpha,M.alpha,M.alpha),
SRC_ALPHA_SATURATE = (f,f,f,1), where f = min{N.alpha,1 − N.alpha}.

Common selections are ν = ONE and μ = ZERO, in which case the source image is always painted on top of the destination image, or ν = SRC_ALPHA and μ = ONE_MINUS_SRC_ALPHA, in which case the source image is weighted by the object's opacity, and the destination image is weighted by the object's transparency.

As previously stated, color, transparency, and blending factors are properties of visual objects. Hence they can all be changed in this non-flexible syntax to the extent allowed by individual objects. For example, in a video object, color, transparency, and perhaps blending factors can be changed every encoded frame.

Many interesting effects can be achieved with non-flexible composition. However, the approach is limited. In the above non-flexible composition example, the encoder must explicitly transmit every change in a scene's composition, using valuable bandwidth. Groups of objects do not share related composition information. Thus a scene containing thousands of polygons representing animated objects would use an extraordinary amount of bandwidth, even if the animation can be naturally described with only a few time-varying parameters. Furthermore, none of the examples above involving user interaction can be achieved with the non-flexible composition syntax alone, as it is defined above. The syntax is too rigid to express the range of behaviors that one may desire to provide *at the decoder* in response to user interaction and other local state changes. One solution to the problem is to enlarge the non-flexible bitstream syntax to accommodate a specific set of behaviors, such as the ones above. However, the range of possible behaviors is extremely large. The ultimate solution is to provide a mechanism for the encoder to download code in a programming language to describe the desired behaviors. This is the approach taken in the flexible approach to composition.

3.3. *The flexible approach to composition*

In the flexible approach to composition, it is assumed that a script in a flexible programming language, such as Java™ or a variation of it, can be downloaded into the MPEG-4 decoder. The language will not have powerful composition operators built in (as PostScript does, for example). Rather, downloaded scripts in the language will perform powerful composition operations by passing data through a standardized interface, the composition API. This is similar to the flexible approach to decompression (Section 4).

The composition API consists of modules, or classes of objects. Each module defines a data structure and a collection of methods (i.e., functions) for its class of objects, as is typical in object-oriented programming methodologies. Discussed here are only the major modules, and the generic functions they perform. The composition API, like other MPEG-4 Systems APIs, is under continual development. The latest version of the API can be found at the MSDL web site `http://www-elec.enst.fr/msdl/`.

The central module in the composition API is the Compositor. The Compositor composites, renders, and blends audiovisual objects onto output audio and video frames. Other modules in the composition API basically support these functions.

The Compositor maintains one audio frame or one video frame for each output channel. For example, a Compositor that outputs five-channel audio and two-channel (stereo) video maintains five audio frames and two video frames. An audio frame is a finite sequence of audio samples; a video frame is a rectangular array of pixels.

Audiovisual effects are produced frame by frame. At some presentation frame rate (which may vary from one decoder implementation to another, or may even be time-varying), a sequence of instructions in the scripting language is sent to the Compositor. These instructions (actually, method calls) produce all audio and video frames (one per output channel) in parallel. Thus the Compositor produces video and audio frames in lock-step, at the presentation frame rate. By definition, the length of the audio frames is adjusted to the interval between video frames. Once the audio and video frames are produced, a mechanism

outside the scope of the API concatenates the audio and video frames onto output buffers for presentation to the user.[2]

Instructions in the programming language to the Compositor are fairly low level, and fall into three categories:
- instructions to render pre-defined (i.e., standardized) and encoder-defined (i.e., downloaded) audiovisual objects,
- instructions to change the current state, or 'context', and
- instructions to save or restore the current state.

Sequences of such instructions to the Compositor are used to produce the video and audio frames. A typical sequence of instructions to composite, render, and blend an audiovisual object into a scene might be: save the current state, modify the state to affect the next audiovisual object to be rendered, render the object, and restore the saved state. Such sequences are repeated for each object in the scene.

The current state, or context, maintained by the Compositor includes the following elements:
- a coordinate transformation matrix that maps the local spatio-temporal coordinate system of an audiovisual object into the global spatio-temporal coordinate system of the scene,
- a properties sheet that contains the default color, transparency, blending factors, line width, and so forth, for audiovisual objects as they are rendered,
- an array of 'cameras', which specify the projection and clipping planes (with respect to the global coordinate system) for each video channel,
- an array of 'microphones', which specify the location and directional sensitivity of an acoustic sink point, for each audio channel,
- an input stream, from which audiovisual objects may read data as they are rendered,
- an output stream, to which audiovisual objects may write data as they are rendered,
- and a dictionary of attribute/value pairs, which may be used to pass generic information from one audiovisual object to the next.

The coordinate transformation defines the coordinate transformation discussed in Section 3.1, and can be represented by the parameters $A_{xx}, A_{xy}, \ldots, B_t$. The properties sheet defines the default parameters for any rendering operations that require parameters. For example, the properties sheet contains parameters c1, c2, c3, alpha, which represent the color and transparency of objects that do not explicitly specify color and transparency, e.g., 3D objects without associated texture maps. The cameras define the precise relationships between the coordinate systems of the logical displays and the coordinate system of the scene. These relationships are represented by coordinate transformations. The cameras also define parameters relating to aspect ratio, clipping volume, etc. The microphones define analogous relationships between the logical audio output devices and audio sources in the scene. They also define parameters relating to directional sensitivity, gain, attenuation, and so forth. The input stream is included in the Compositor state because the contents of the input stream determine how most audiovisual objects (audio, video, synthetic animation, and other coded objects) appear when rendered. The output stream is included because audiovisual objects are able to produce output information when they are processed. Finally, the dictionary of attribute/value pairs is a catch-all mechanism for passing generic information from one audiovisual object to another. The value of an arbitrary attribute can be set by one audiovisual object when it is rendered, and can be read by another audiovisual object when it is rendered. Like 'environment variables' in UNIX, this mechanism is especially effective for broadcasting information across a collection of objects.

[2] Note that the audio and video frames produced by the Compositor are *presentation* frames, and do not in general correspond to any periodic structures such as 'frames', that may be used to *code* audiovisual objects. In fact, since the presentation frame rate may vary from one decoder to another, or may be time-varying depending on the computational load at the decoder, it is impossible for an encoder to keep track of the presentation frame rate. This is an example of the architectural separation between the presentation of audiovisual objects and their underlying representation.

These state elements are simply objects from different classes in the composition API. The Compositor maintains one object from each of these classes. Thus, the Compositor maintains one object from a coordinate transformation class, one object from a properties sheet class, and so forth. The elements of the current Compositor state are therefore manipulable individually, by means of method calls to the individual objects in the Compositor. For example, instructions may be sent to the Compositor's current coordinate transformation matrix to modify its spatial or temporal components. This is the way to set the spatio-temporal pose of the next object to be rendered.

The Compositor also maintains stacks of objects to facilitate saving and restoring some elements of the current state. In particular, the Compositor maintains a stack for coordinate transformation matrices and a stack for properties sheets. Thus these state elements can be saved and restored by pushing and popping their associated stacks.

The difficult work done by a Compositor is rendering audiovisual objects. The Compositor has methods for rendering all pre-defined (i.e., standardized) audiovisual objects, and also has a method for rendering all encoder-defined (i.e., downloaded) audiovisual objects.

When a method is called for rendering a pre-defined, static visual object, such as an image or a polygon, the following is performed. The Compositor performs the 2.5D or 3D spatial coordinate transformation specified by the current coordinate transformation matrix, thereby positioning the object in the scene. (The temporal portion of the coordinate transformation matrix is ignored, because the object is static, or time-invariant.) Then for each camera, the Compositor:
- clips those portions of the object outside the camera's clipping volume,
- projects the object onto the camera's projection plane,
- resamples the object to match the target pixels of the projection plane,
- computes the color and transparency of each object pixel, which may vary according to the object's orientation, surface normals, and lighting conditions, and finally
- blends each object pixel with its corresponding target pixel, according to the source and target blending factors, if the object pixel is not hidden by a previously rendered, opaque object.

Of course, much of this procedure is simplified in the 2.5D case (e.g., lighting and surface normals are ignored).

When a method is called for rendering a pre-defined, time-varying visual object, such as a video object, steps identical to the above are performed. First, however, the time-varying object is reduced to a static object. This is done by temporally sampling, or 'slicing' the time-varying, 2.5D + T or 3D + T object, at some temporal offset in its local coordinate system, to produce a static, 2.5D or 3D object. The local temporal offset t at which the object is sliced corresponds to global time $t' = 0$.

The rationale for slicing time-varying audiovisual objects at global time $t' = 0$ is the following. The global scene consists of one or more time-varying audiovisual objects. The temporal coordinate system of each object has been stretched and delayed, in general, by an affine coordinate transformation:

$$t' = (A_{tt})^{*}t + (B_t)$$

that maps the object's local temporal coordinate system into the scene's global temporal coordinate system. Fig. 7 shows a local coordinate system that has been stretched (slowed down) by a factor of $A_{tt} = 2$, and delayed by $B_t = 4$ seconds, relative to the global coordinate system.

A mechanism beyond the scope of this API generates video frames at a presentation frame rate. Conceptually, each frame represents the scene, temporally sliced at advancing global time instants $t' = 0.0$, 0.1, 0.2, ..., if the presentation frame rate is 10 frames per second, for example. By convention, instead of slicing the scene at advancing global time instants $t' = 0.0, 0.1, 0.2, \ldots$, all the local coordinate systems in the scene are advanced (temporally shifted to the left) by 0.0, 0.1, 0.2, ..., and then the scene is sliced at global time $t' = 0$. In this way, the local time at which a visual object should be sliced can be determined by inspecting the temporal components of the current coordinate transformation matrix, and solving the

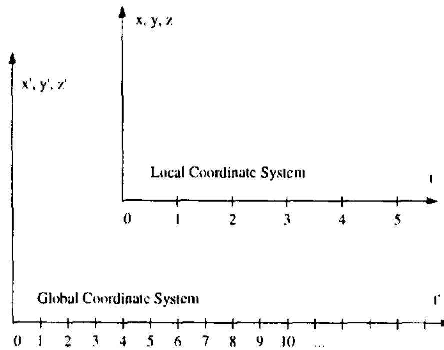


Fig. 7. Global and local coordinate systems.

equation

$$0 = (A_{tt})^* t + (B_t)$$

for the local time,

$$t = -(B_t)/(A_{tt}).$$

In summary, when a Compositor method is called for rendering a time-varying visual object, the 2.5D + T or 3D + T object is first temporally sliced at local time $t = -(B_t)/(A_{tt})$, to produce a 2.5D or 3D object, which can be subsequently rendered as a static object. Static objects are considered to be 2.5D+T or 3D+T objects, which happen to be constant for all time.

Audio objects are time varying by nature. When a Compositor method is called for rendering an audio object, the audio object is similarly temporally sliced, but the slices have width. To be precise, the portion of the audio object rendered onto the audio frame(s) is given by the interval $[t, t+T)$ in the object's local coordinate system, where t and T satisfy

$$0 = (A_{tt})^* t + (B_t),$$

$$T' = (A_{tt})^* T + (B_t),$$

where T' is the instantaneous interval between presentation frames, in the global coordinate system. The Compositor 'renders' other pre-defined objects (such as cameras, microphones, and light sources) differently, depending on the object. 'Rendering' such objects usually means registering them in an appropriate way with the Compositor, using the current coordinate transformation matrix to determine the pose and directional characteristics of the objects. For example, a camera object defines a projection plane and a clipping volume in its local coordinate system; likewise, a light source defines a radiation pattern in its local coordinate system. When these objects are 'rendered', the current coordinate transformation matrix determines the direction, scale, and translation of these objects.

Key to the flexible composition approach are encoder-defined audiovisual objects. An encoder can define new classes of audiovisual objects, by downloading the new class definitions in the programming language. All encoder-defined classes of audiovisual objects (as well as all pre-defined classes of audiovisual objects) must by definition be derived from an abstract base class called `AVObject`, provided by the API. The `AVObject` base class defines a generic public interface for audiovisual objects of all kinds. Since all audiovisual objects must be derived from this base class, all audiovisual objects must provide the minimal set of methods specified by the base class. Two methods in particular are defined by the base class, and hence must be provided by all encoder-defined (and pre-defined) audiovisual objects:

- `render`, which takes a Compositor as an argument, and
- `handle`, which takes an event structure as an argument.

Of course, audiovisual objects may also provide additional, class-specific methods.

The purpose of an audiovisual object's `render` method is to render the object onto the specified compositor. In an encoder-defined audiovisual object, for which the class definition is downloaded in the programming language, the body of the routine constitutes a script that describes step by step how to render the object onto the compositor. (The script is executed once per presentation frame by a mechanism outside the scope of the API.)

The script may contain an arbitrary sequence of instructions, but at the very least it contains instructions to the Compositor to render the object's audiovisual sub-objects in appropriate poses. The script may also contain instructions to save and restore the current state, instructions to compute the local time from the current coordinate transformation, instructions to obtain decoded information from the appropriate decoders, instructions to read input, instructions to set and test state variables, branch conditionally, and so forth.

The purpose of the audiovisual object's `handle` method is to handle events, whether synchronously generated by a script, or asynchronously generated by user input for example. In an encoder-defined audiovisual object, for which the class definition is downloaded in the programming language, the body of the routine constitutes a script that describes step by step how to handle events. (The script is executed once per asynchronous event by a mechanism outside the scope of the API). The script may contain an arbitrary sequence of instructions, but at the very least it contains instructions to examine the information in the event structure, and to synchronously pass the event to the event handlers of the object's sub-objects. The script may also contain instructions to modify state information in the object. This state information can be tested in the `render` method to influence the way the object is rendered.

Because rendering an encoder-defined audiovisual object involves rendering sub-objects, which can in turn be encoder-defined audiovisual objects, audiovisual objects are in general hierarchical. The hierarchy constitutes a scene graph. The scene graph is dynamic, because any audiovisual object can conditionally render its sub-objects, so the scene may change from frame to frame. At the root of the scene graph is the scene. Thus the scene is itself an audiovisual object. This has implications for bitstream editing. Two audiovisual objects can be authored in different institutions, and can individually play on any MPEG-4 decoder. Then, a new scene class can be defined, with the two audiovisual objects as sub-objects. The new scene can again be played on any MPEG-4 decoder. 'Plug-and-play' of audiovisual objects is guaranteed by virtue of their standard interface, enforced by derivation from the `AVObject class`. Temporal synchronization between audiovisual objects is guaranteed by the temporal coordinate system model.

When an instruction is sent to the Compositor to render an encoder-defined audiovisual object, the Compositor does not know intrinsically how to render encoder-defined audiovisual objects, but encoder-defined audiovisual objects know how to render themselves. So when the `render` method of the Compositor is called upon to render an encoder-defined audiovisual object, it turns around and calls the object's render method, with itself as the argument. It is not necessary to register encoder-defined audiovisual objects with the Compositor, for it to know how to render them; the Compositor makes use only of the generic properties of audiovisual objects. The ability to deal with a variety of different types of objects behind a single interface is called polymorphism in object-oriented terminology.

Although 'the' Compositor has been described until this point, the Compositor module is actually a class or template for compositor objects. Thus in principle many compositor objects may be instantiated simultaneously within a decoder. Although one of the compositor objects must be connected to the output frame buffers for each audio and video channel, the other compositor objects may be constructed with arbitrary numbers of audio and video frames. For example, a compositor object may be constructed with one video frame and no audio frames.

It is an important architectural principle that a compositor object's output video and audio frames are themselves primitive audiovisual objects, which are therefore directly renderable onto another compositor object's frames. In this way, compositor objects can be networked together, with some compositor objects producing intermediate results for others. For example, one compositor object may be set up to compute 3D projections, while another compositor object may be set up to compute 2D overlay planes. A final

compositor object may simply blend the overlay planes with the 3D projections. Such configurations can be set up in the render method of the downloaded scene.

3.4. Examples

The following scripts are working examples of encoder-defined AV objects from a current implementation of the API. It should be noted that these are only illustrative examples. The API is subject to change in future revisions of MSDL; furthermore, the syntactic decoding operations are treated here differently from what is presented in Section 5 in order to make the examples self-contained.

3.4.1. ReversibleVideoObject

This first example shows a video playing in the center of the display, rotating counterclockwise at 90.0°/s. When the user points to the object, and clicks, the object reverses direction.

This effect is achieved by overriding the handle method. When a user event is passed to the object, the handle method checks the event structure to see if the label on the object matches the object's identity. If so, it stores the time and the angle of rotation when the click occurred. It also reverses the sign of the rotation speed.

The render method computes the local time, and computes the angle of rotation since the last click. It then rotates the object by the computed angle, labels the object with its identifier, and renders the object. The current state is saved and restored.

```
public class ReversibleVideoObject extends AVObject {
  VideoObject videoObject = new VideoObject();
  byte ident = 17;
  double time, angle;
  double timeAtLastClick = 0;
  double angleAtLastClick = 0;
  double degsPerSec = 90.0;
public void render(Compositor c) {
    time = c.transform.localTime();
    angle = (time-timeAtLastClick) */degsPerSec + angleAtLastClick;
    c.pushTransform();
    c.pushProperties();
    c.transform.rotate(angle);
    c.properties.label(ident);
    c.render(videoObject);
    c.popProperties();
    c.popTransform();
    }
public void handle(Event e) {
    if (((PressEvent) e).label == ident) {
      timeAtLastClick = time;
      angleAtLastClick = angle;
      degsPerSec* = -1;
      }
    }
public ReversibleVideoObject(int id) {ident = (byte)id; }
public ReversibleVideoObject() { }
}
```

3.4.2. DoubleReversibleVideoObject

This second example shows two ReversibleVideoObjects playing side by side. One object rotates and plays at 0.25 the speed of the other. The `Handle` method passes received events to each sub-object, without interpretation. The two objects are labeled differently, so each will reverse itself independently.

```
public class DoubleReversibleVideoObject extends AVObject {
  AVObject obj1 = new ReversibleVideoObject(17);
  AVObject obj2 = new ReversibleVideoObject(23);
public void render(Compositor c) {
    c.pushTransform();
    c.transform.translate( - 80,0);
    c.render(obj1);
    c.transform.speed(0.25);
    c.transform.translate(160,0);
    c.render(obj2);
    c.popTransform();
    }
public void handle(mpeg4.Event e) {
    obj1.handle(e);
    obj2.handle(e);
    }
}
```

3.4.3. AnimatedVideoObject

This final example shows a video object whose DPCM-encoded (x,y) position is parsed and decoded from the input stream. For conceptual simplicity, the `render` method here makes use of a private `decode` method to decode the (x,y) position. The syntactic description can be integrated with the class declaration, as described in Section 5.

```
public class AnimatedVideoObject extends AVObject {
  VideoObject videoObject = new VideoObject();
  int x = 0;
  int y = 0;
public void render(Compositor c) {
  decode(c.inputstream);
  c.pushTransform();
  c.transform.translate(x, y);
  c.render(videoObject);
  c.popTransform();
  }
void decode(InputStream is) {
  if (!is.eos()){
    x+ = is.int(9);
    y+ = is.int(9);
    }
  }
}
```

3.5. Future work

A large amount of work remains to be done in the area of MPEG-4 composition. In particular:

- Harmonization with VRML, the Java™ Media API, etc. A standard should unify the marketplace, not splinter it. None of the existing standards address all the needs of MPEG-4, but there is substantial overlap with some of them. Work needs to be done to harmonize MPEG-4 with existing standards as much as possible.
- Adding higher level constructs to the API. The composition API is low-level, much closer to OpenGL than to VRML, for example. This permits the maximum amount of flexibility to script writers. However, authoring scripts is difficult, and moreover, different authors will build different higher-level functionalities (e.g., layout management) into their audiovisual objects. In that case the audiovisual objects will only work together at a very low level. Work needs to be done to standardize higher level mechanisms in the API.
- Specifying more primitives. Work needs to be done to define standard audiovisual objects and other primitives, such as text, fonts, synthetic audio, and other objects that script authors should be able to count on.
- Refining the audio rendering. To date, audio rendering has not been implemented within the context of the composition API. The design needs to be implemented and refined.
- Refining the non-flexible composition syntax for different Systems profiles.

4. Decompression – reconstruction of AV objects

4.1. Key concepts and requirements

In previous coding standards, such as MPEG-1, MPEG-2, or H.263, a fixed set of algorithms was specified and used to decode synchronized video and audio frames. MPEG-4 intends to cover a much larger scope: it specifies some flexible and configurable decompression algorithms for various types of audiovisual data, not restricted to audio or video frames. Typically, an MPEG-4 scene may consist of a movie coded with MPEG-2, a 3D logo coded with a specific encoder-defined algorithm, and a sound coded with a modified version of G.723. In the following sections, the design of a decoder enabling such functionalities will be detailed.

To achieve the above-mentioned goals, an object-oriented design of the decoder has been adopted. The key concept of this design, as described in Section 2.3, is to separate the audiovisual objects that represent the data to be manipulated in the application (video frames, 3D graphic objects, etc.), from the decompression processes. The decoder will thus rely on a set of classes (in the object-oriented sense) called the standard class library. This library will provide the necessary interfaces for decompressing a pre-defined set of classes of audiovisual data with a pre-defined set of tools. Moreover, the design makes it possible to define and download new audiovisual primitives, or decompression tools, and to configure tools into new algorithms.

In summary, an MPEG-4 terminal provides the following capabilities regarding to decompression:

- the use of pre-defined algorithms, such as MPEG-1 video or audio, MPEG-2 video profiles or MPEG-4 audio or video profiles,
- the configuration of existing tools into new algorithms,
- the definition of new tools and algorithms, and
- the exchange of various messages with the multiplexer and the compositor.

These capabilities are eventually limited by the flexibility supported by the terminal. These limitations will be described in the next sections.

4.2. The standard class library for decompression

The standard class library contains the basic AV objects and process objects needed by a non-flexible application. It also contains all the configurable tools that will be used in a flexible application (see Section 2). In this section, a subset of the standard class library, as it is currently defined, is described. The standard class library is evolving according to the verification model specifications for audio and video coding (both natural and synthetic) in MPEG-4. Each audio and video profile of MPEG-4 may use different subsets of the standard class library.

As mentioned in the previous section, the standard class library for decompression contains two kinds of classes:
- `AVObjects`, which are the elementary audiovisual components that need to be manipulated in the application, and
- `ProcessObjects`, which are the decompression tools used by AV objects to decode themselves.

As discussed in Section 3, AV objects define two methods in particular: `render` and `handle` methods. In addition, many of the AV object classes in the standard class library define specific `decode` methods. The `decode` method of such a standard AV object decodes the attributes of the object itself: it builds and instantiates from a coded representation, all the attributes of the AV object. The method is typically implemented using high level process objects, but it may be overridden in a flexible environment. For instance, the `decode` method of a 3D model may be overridden to just read a VRML description, or the `decode` method of an image may be overridden to use a newly defined decompression algorithm.

Process objects perform processing on AV objects, by using an `apply` method. This `apply` method changes or instantiates the attributes of the AV object being processed.

The MPEG-4 Systems is designed in this way to emphasize the independence of AV objects and the way they have been coded in a particular application. Moreover, designing the architecture in this way promotes reusability of the tools in various contexts. For instance, a DCT coding tool may be used by MPEG-1/2, H.263, JPEG and by any downloaded decompression method.

AV Objects in the class library are likely to include video objects, images, audio waveforms, and 3D face models, for example. Fig. 8 shows an OMT representation of some AV object classes.

Process Objects in the standard class library are organized in two sections:
- high level objects, such as MPEG-4 profiles, MPEG-2 profiles, exemplified for video in Fig. 9, and
- low level decompression tools such as quantizers, exemplified for video in Fig. 10.

4.3. The non-flexible approach to decompression

In this section, we deal with decompression for a non-flexible terminal. This corresponds to a finite set of standardized audio, video, and system algorithms made up of standardized tools. In this section, the concept

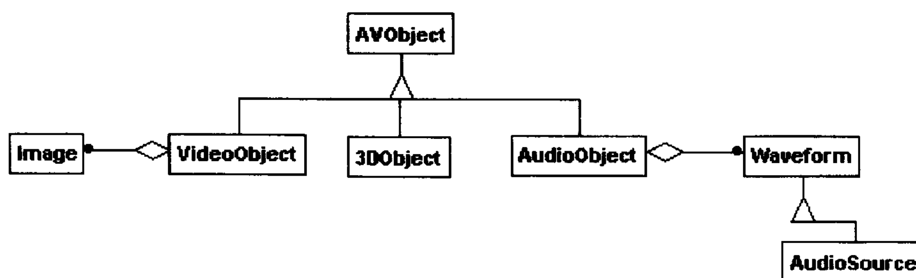


Fig. 8. OMT diagram: examples of AV objects.

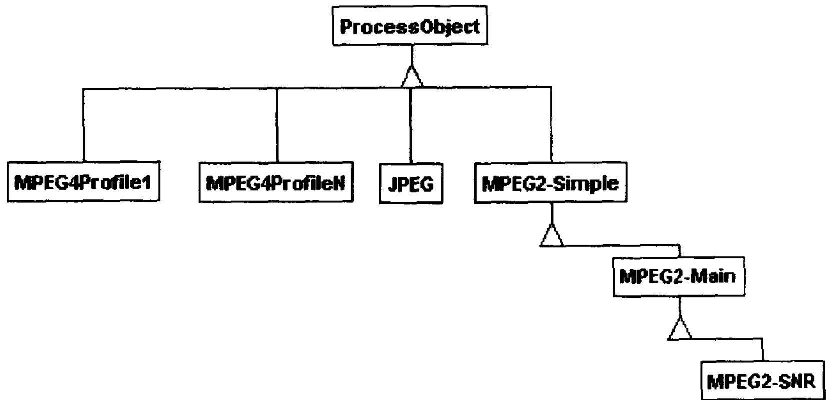


Fig. 9. OMT diagram: example of high level process objects.

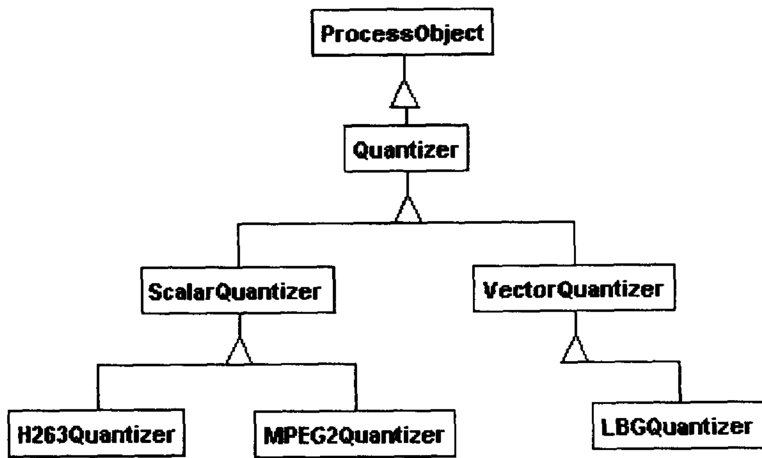


Fig. 10. OMT diagram: example of low level process objects.

of the standard class library is used to enable a consistent description of decoders for all levels of flexibility. However, it should be understood that non-flexible terminals do not require the availability of the class interfaces in any specific object-oriented language. The object-oriented formalism is used here, for the consistency of description between the non-flexible and the flexible approach, just as pseudo-C code is used in MPEG-2 for describing its syntax. Bearing that in mind, relating a non-flexible application with the standard class library implies that:
- only classes of AV objects in the standard class library are used,
- each of these AV Objects has a defined standard decompression method, and
- only decompression tools in the standard process objects library are used.

Let us now take the example of a video object in a multi-standard non-flexible application in which all standard video decompression algorithms can be used. In that case, the `VideoObject` class of the standard library will have a `decode` method that takes into account MPEG-2 and MPEG-4 video profiles, as well as

all the video object decoders listed in the standard class library. A pre-defined switch could enable the choice between these algorithms. The standard decode method of the video object may look like:[3]

```
class VideoObject extends AVObject {
  int algorithmType;
  ProcessObject coder = null;
  Image image = Image();
  public void render(Compositor c) {
    decode(c);
    c.render(image);
}
protected void decode(Compositor c) {
  if (coder = = null) {
    algorithmType = c.inputstream.uint(9);
    switch (algorithmType)  {...                    //Construction of an
    case 0:                                          //MPEG2-High Process
      coder = new MPEG2High (c.inputstream);         //Object, reading data
                                                     //from c.inputstream
    break;
    ...
    case xxx:
      coder = new MPEG4ProfileN(c.inputstream);
      break;
    default:
      Error('Coder not supported in the standard library');
      break;
    }
  }
 coder.apply(image,c.transform.localTime());         //decode input
                                                     //stream into image
                                                     //at presentation time
                                                     //c.transform.localtime()
}
```

Then, the only difference between an MPEG-4 decoder architecture and a classical MPEG-2 one is the 9 bits parameter in the header of the bitstream that specifies the standard algorithm to be used (note that the syntactic decoding operations here are handled differently from what is presented in Section 5 to make the examples self-contained).

The non-flexible approach for decompression is thus very similar to the classical MPEG-2 like standards. The algorithms are selected by a switch sent in the bitstream. The only difference is that a broader set of algorithms is eventually available.

The non-flexible approach is useful for applications that are satisfied with the fixed standardized set of profiles (standardized configuration of tools). However, any time an application needs a new decompression algorithms (non-standardized configuration of tools) to satisfy its special needs, either a new round of standardization is engaged, or the flexible approach is used.

[3]Here we are using a Java-like description of the code. For purposes of clarity and conciseness, we only provide pseudo code in this example, as well as the following ones concerning decoding.

4.4. The flexible approach to decompression

4.4.1. Configuration of existing tools into new algorithms

In this section, we consider that a finite set of standardized audio, video, and system tools and their standardized interfaces are available at the terminal. These tools may be flexibly configured into arbitrary algorithms. In terms of the standard class library this means the following:
- the standard class library can be used as in a non-flexible application,
- decompression methods for standard AV objects can be overridden by configurations of standard tools, and
- standard decompression tools can be used for non-standard (encoder-defined) AV objects.

In order to fulfill the above requirements, the flexible terminal must have the ability to download new classes. Let us give now some concrete examples to illustrate the various possibilities for using the flexible approach to decompression in an MPEG-4 decoder.

In the first example, let us consider a flexible application in which a video object is decoded using the standard decompression method of the non-flexible application given in the previous section. Imagine that due to the low bit-rate used, post-filtering needs to be performed in order to reduce the blocking artifacts. To do so in the flexible approach, the application first transmits the definition of a new class, say `MyVideo Object` class, which simply inherits from the standard `VideoObject` class and overrides its `decode` method in the following way:

```
class MyVideoObject extends VideoObject {
    protected void decode(InputStream is) {VideoObject.decode(is);
    PostFilter.apply(this);
  }
}
```

Note here that it is not necessary to re-define the `render` method of the class, since it is exactly the same as the one of its parent class. This is an example where the independence of the rendering and the decompression processes is used. Moreover, the standard method for decompressing a video object for the particular profile used did not need to be redefined, since the `decode` method of the parent class was called.

The scene (top-most AV Object in the application) makes then use of the `MyVideoObject` class. A typical render method (see Section 3 for more detail) of a scene displaying two videos, one filling the whole screen (using this new decoding process) and one in the lower left corner of the screen (using the standard decompression process) will look like

```
class MyScene extends AVObject {
    MyVideoObject video1;
    VideoObject video2;
    public void render(Compositor c) {
      c.pushTransform();
      c.render(video1);// equivalent to video1.render(c);
      c.transform.scale(0.25);
      c.render(video2); //equivalent to video2.render(c);
      c.popTransform();
  }
}
```

In this example, the `MyVideoObject` class definition needs to be sent before the data. No additional overhead is needed in the bit stream.

A second context for that kind of flexible application is the case in which a new AV object class is created, and its `decode` method just makes use of existing tools in the standard library.

Let us consider a simple example of a two-dimensional polygon object, for which vertices are coded two by two, using a simple LBG quantizer in four dimensions, with a fixed codebook. The class definition with the associated `decode` method may look like

```
class 2DPolygon extends AVObject {
     ListOfPoint vertices = new ListOfPoint();
     Codebook myDictionary = new Codebook(x1, y1, x2, y2, ...);
     // Defines the fixed Codebook for this particular example
     Quantizer quantizer = new LBGQuantizer(myDictionary, ...);
     // Builds an LBG quantizer with a given dictionary and relevant parameters
     EventSource events = new EventSource(4);
     public void render(Compositor c) {
         decode(c.inputstream);
         c.render(vertices);
     }
     protected void decode(InputStream is) {
         int numberOfVertexPairs is.uint(5);    // read number of vertex
                                                // pairs
         while (numberOfVertexPairs-- > 0) {
              quantizer.apply(events,is);
              // The apply method of the LBG quantizer reads the quantized
              // vectors
              // from the inputstream and put the result in the event
              // object
              vertices.InsertPoint(events.elem(0),events.elem(1));
              vertices.InsertPoint(events.elem(2),events.elem(3));
         }
     }
}
```

In the above example, an alternative possibility would be to define a new process object `Polygon-Decoder`, and make an explicit call to this process object in the decode method of the 2DPolygon object. This would be done if the `PolygonDecoder` process object had to be re-used in another context in the application. Note also in the above example that the class definition is using four classes of the standard class library, namely the `LBGQuantizer`, `ListOfPoint`, `EventSource` and `CodeBook` classes. The main advantages for using the standard class library here are the following:
- the overhead of sending class definitions in order to run the application at the terminal is reduced, and
- the implementation of the standard class library is optimized for the terminal, and is usually more efficient computationally than a downloaded version of the same library. For this reason, only the interfaces are standardized.

Concerning the terminal, the following elements are required:
- a locally implemented and optimized version of the class library, and
- the ability to download new classes and link them with the standard class library.

The standard class library will be called any time a computationally expensive tool is required. If the decoder is powerful enough to download full tool definitions or algorithms, decompression may be reconfigured as described in the following section.

This kind of flexible terminal enables downloading complex audiovisual scene definitions, with specific treatment of interactivity (see Section 3 on composition), and defining new AV objects with associated decompression methods.

4.4.2. The definition of new tools and algorithms

We describe in this section a possible extension of the flexible approach defined in MPEG-4. We consider a terminal with a standardized mechanism to describe arbitrary algorithms made of arbitrary tools. From the terminal point of view, the following capabilities are required:
- the standard library can be used and extended as in the previous flexible environment, and
- new process objects can be downloaded and used efficiently to enhance the library of tools.

Compared to the capabilities detailed in the previous section, the terminals have now the ability to download and use efficiently decompression tools. From an architecture point of view, there is no major difference with the previous approach. However, a practical implementation of this approach that will ensure satisfactory interworking between heterogeneous terminals is not yet foreseen.

Taking the 2DPolygons example of Section 4.4.1, an MPEG-4 content provider may like to use, for coding efficiency reasons, its own quantizing tool to code 2DPolygons vertices. Let us take the example of a D4 lattice vector quantizer tool, which has not been included in the standard class library. Two classes need to be transmitted:

```
class D4LatticeVectorQuantizer extends VectorQuantizer {
  public void apply(EventSource events,InputStream is) {
      int rho = is.int(8);
      int theta = is.int(8);
    // rho and theta are the result of the D4 lattice VQ
    // here the input parameter of the inverse quantization...
      The code of the D4 Lattice VQ inverse quantization procedure...
    }
  }
```

Then, the new Polygon2D class using this new tool in the decompression procedure is sent:

```
class 2DPolygon extends AVObject {
     ListOfPoint vertices = new ListOfPoint();
     Quantizer quantizer = new D4LatticeVectorQuantizer();
     EventSource events = new EventSource(4);
     public void render(Compositor c) {
           decode(c.inputstream);
           c.render(vertices);
    }
    protected void decode(InputStream is) {
          int numberOfVertexPairs = is.uint(5);     // read number of vertex
                                                   // pairs
          while (numberOfVertexPairs-- > 0) {
                quantizer.apply(events,is);
                vertices.InsertPoint(events.elem(0),events.elem(1));
                vertices.InsertPoint(events.elem(2),events.elem(3));
          }
       }
}
```

Note here that the inverse lattice vector quantization tool may be used in other contexts in the same application, to decode other kinds of AV objects. This is the reason to download it as a separate process object. Note also, that polymorphism has been explicitly used in this example. The `LBGQuantizer` and the `D4LatticeVectorQuantizer` are both derived from the abstract `Quantizer` class. They are all using an apply method with the same signature. The advantage of doing so, is that the code for the `decode` method of the two polygons is exactly the same. Only the declaration of the quantizer has changed.

4.5. Future work

Concerning decompression, the non-flexible approach is feasible today almost without changing the current architectures of audiovisual terminals such as MPEG-2 decoders. But the impact in terms of functionalities and applications is limited, since only pre-defined decompression algorithms may be used, for pre-defined audiovisual objects.

Flexible terminals provide for the ability to download descriptions of new classes of data (AV objects), with associated decompression methods, using locally optimized decompression tools. Such flexible terminals enable content providers to adapt the decompression processes to the specific data they are dealing with, by reconfiguring the algorithms with standardized tools. Depending on the technology and market needs, the flexible terminals may be extended to allow downloading of tools.

Flexibility enables manipulation of much richer content than in previous standards. However, to achieve this, much work remains to be done to completely specify and validate the decompression interfaces for natural and synthetic audiovisual objects.

5. Syntactic decoding – interpreting the bits

5.1. Rationale

The Syntactic Description Language (SDL)[4] part of MSDL addresses the need to disengage the definition of the bitstream syntax of MPEG-4 content from the decoding and rendering tools. This requirement originates from the fact that a given syntax specification may be decoded using different implementations of the relevant algorithms. This certainly has been the central theme in the MPEG-1 and MPEG-2 series of specifications, even though the syntax specification utilized both formal and non-formal techniques (i.e., it included explanatory text without which the definition of the syntax would be incomplete).

The separation of bitstream parsing from the other decoding and rendering steps provides content developers with the capability to create customized bitstream structures to suit their specific application needs. For example, it allows content developers to modify the size of parameters present in the bitstream without the need to perform any modification in the decoding tools. It also allows the introduction of new bitstream parameters without losing backwards compatibility with older tools: the new parameters will be ignored by older tools, but they can be used by their newer versions.

This separation also promotes an open approach in terms of bitstream definition, in which content developers may wish to publish their low-level syntax but not their proprietary processing algorithms. This is especially important for flexible systems, and is in line with the implementation-independent approach of MPEG-1 and MPEG-2. Terminals who are not equipped with the appropriate decoding tool will just parse and discard any corresponding data. Alternatively, different implementations of the decoding tools may be used, developed by a third party.

[4]SDL is unrelated to The ‘Specification and Description Language’ (ITU-T Z.100) which is used in the telecommunications field.

An additional important benefit of separating bitstream parsing from processing is that the task of the bitstream architect (or content developer) is greatly simplified. Focus is drawn on the important tasks of decoding information and preparing it for presentation, and not in the mundane task of obtaining bits from a bitstream. This is an underlying theme in all typed programming languages where a set of standard types (chars, ints, doubles) are directly provided by the language, without requiring direct manipulation of their representation by the programmer. The language compiler or interpreter is responsible for ensuring that data manipulations (including conversions) are consistent with type declarations or their derivatives (extended types). These languages, of course, do not take into account that the data may be obtained from a bitstream, but the same concept applies.

Introducing a formal mechanism to specify bitstream syntax also has significant benefits in terms of the accuracy of the specification. Taking ASN.1 [8, 9] as an example, the specification of a binary encoded structure is performed via a formal language. The binary representation is unambiguously determined by a set of encoding rules [9]. Unfortunately, ASN.1 cannot be used within the context of MPEG-4 because of two drawbacks: (1) it was not designed to address the intricate bitstream structures produced by sophisticated source coding, but rather much simpler packet-like structures, (2) it cannot be integrated with a programming language. The latter is necessary in MPEG-4 as in addition to the bitstream syntax, the standard and content developer will need to also specify a facility to peruse the defined decoding and composition APIs. With a well-defined SDL, the exact text of the syntax specification can be used unchanged to drive software tools that produce the entire parsing/bit generation skeleton of a decoder/encoder, as well as automatically generate compliance testing tools.

Separation of the bitstream syntax from decoding provides for automatic compliance with the overall bitstream architecture that MSDL will define (multiplexing level). The alternative approach would put the burden of complying with bitstream-level object delineation and naming on the content developer, whereas SDL can provide automated facilities that prohibit mistakes or warn about potential conflicts. An example is the process of compilation, which produces a binary file with an appropriate header that identifies it to the operating system as an executable (the 'magic' number).

From an implementation perspective, a formal declarative description of the bitstream syntax provides the utmost flexibility. The description of the syntax should indicate *what* the bitstream contains, and not *how* to obtain it. If the syntax is implicitly defined as a series of processing steps deeply embedded into a decoding tool, that capability is completely lost. The declarative approach allows a very broad spectrum of hardware and software implementations, and provides for market differentiation by allowing creative competitiveness.

From a general architectural perspective, independent programmability of bitstream parsing is a natural extension of the notion of programmable clients. In the software architecture field, the idea of downloading platform independent code and locally executing them in a client is by now well entrenched after the proliferation of Java. Such an architecture, however, assumes to a large extent that the executing program exists isolated from the rest of the world. In the communications-oriented environment that MPEG-4 is addressing, in addition to a downloadable executable program, there is also a notion of a continuously transmitted stream that is received by the client for further processing. It is natural, then, to extend the programmability aspects to the treatment of the received bitstream. As described later on, this is elegantly integrated with the overall programming methodology of MPEG-4.

A potential drawback arising from the separation of parsing is that context information (information that affects the parsing of the coming bits) has to be very close to the bitstream level. If a technique relies on extensive decoding in order to obtain the value of a particular condition, then in order to parse the bitstream extensive decoding has to take place. In such situations, the condition has to be expressed in the bitstream, rather than be inferred from decoding. Examination of current state-of-the-art specifications does not reveal instances where such high-level context is used. In addition, the potential overhead for adding bitstream-level information in these cases should be expected to be rather small.

SDL has been designed so that it can describe existing audiovisual coding standards. SDL is also an easy-to-read way of defining syntax specifications, since it is based on a set of well-defined elements with unambiguous semantics. It has been successfully used to describe MPEG-2 Video profiles, while work is underway so that all the MPEG-4 specifications, including the current Verification Models, are converted to use SDL.

5.2. SDL and the MSDL environment

MSDL provides the overall programming environment in which MPEG-4 content is developed. In addition to the class structures, or APIs, that are defined for decoding and compositing, it will also include a programming facility through which these APIs will be exercised (Java is currently being used). SDL is being designed as an orthogonal component of this language. Orthogonality here implies that the two are independent: the specification of SDL does not affect the programming constructs and vice versa. There are, of course, some common basic principles assumed, at the level of the capability to define data structures and object hierarchies.

In its current form, SDL (in its textual version) assumes a C++/Java-like approach as the central theme of the MSDL programming language. It then proceeds to extend the typing system by providing facilities for defining bitstream-level quantities, and how they should be parsed. SDL can be seen as generalizing the concept of declaring constants with hard-coded values, to that of declaring constants that obtain their values from a bitstream. Similarly to traditional constants where a programmer is not concerned how the initialization is performed, but needs to assume that it is performed before the variable is accessed, an SDL programmer can assume that a constant is parsed from the bitstream before it is accessed. As described later on, parameters are parsed only once, and hence the behavior is indeed very similar to traditional constants.

The interface between SDL and the overall MSDL architecture is therefore well-defined. In addition, it poses no restriction to the individual structures of SDL and other MSDL components. The rule of guaranteeing that a variable is parsed before it is accessed is the most general one. Simpler rules could also be adopted; an example would be to mandate that a variable is parsed when an object is instantiated. Alternatively, parsing can be triggered by a specific API method, although this may significantly limit the implementation flexibility of the decoder. The optimal methodology is currently investigated by experimental implementation that can accurately expose the benefits and drawbacks of each approach. Note that regardless of the method used, the SDL specification is not affected in any way.

The key characteristics of SDL within the MSDL environment can be seen by considering the following simple example of the definition of a trivial object in C++/Java.

```
// C++ /Java
class simple {
  int alpha;
  void decode()  {
    alpha = in.getint(3);      // explicit parsing
    alpha+ = 15;               // decoding
  }
};
```

Here the bitstream (in) contains the value of the parameter `alpha` in 3 bits, which is then subsequently decoded by a simple offset adjustment. The same object in SDL would be defined as:

```
// MSDL
```

```
class simple {
  int(3) alpha;          // declarative parsing
  void decode() {
    alpha+ =15;          // decoding
  }
};
```

The construct `int(3)` indicates that the parameter `alpha` will be parsed as 3 bits and then converted to an `int`. The fundamental programming constructs remain unchanged, but with the added benefit of clearly separating the parsing information. Moreover, all the necessary information (data types, parsing information, and object methods) is conveniently defined in a single place.

With respect to flexibility, and since SDL addresses the definition of the fundamental bitstream syntax, it can potentially be applied in both flexible and non-flexible terminals. Clearly, incorporating a programmable syntax parser even of medium complexity has a non-trivial impact on the overall cost of the terminal. As a result, due to the desire to keep non-flexible implementations at low complexity and cost, syntax programmability is currently considered primarily for flexible terminals.

With respect to the communication of the syntax to the MPEG-4 terminal, a binary format will be used. Taking the parallel of Java, for downloadable object definitions the syntax specification becomes an additional part of the downloadable class information. In MPEG-4, however, this information can also be downloaded independently from the other components. This allows redefinition of the syntax without modifying in any way the implementation of the methods of the particular class (tools). As will become evident in the next section, this is easily achievable as long as existing data types and interfaces of the particular class remain the same. The binary format for SDL is currently being investigated and will not be described here.

Note that the complexity of parsing MSDL code is no more complicated than a regular C++ or Java parser. For simulation purposes, the MSDL code can be processed by a translator that generates equivalent C++/Java code (an approach which is currently pursued). The translator can directly generate parsing code, e.g., as part of the constructor, or in the future generate the binary parsing information that would be transmitted to a terminal.

The following describes an overview of the features provided from the current SDL specification. Due to space limitations, several details are omitted; the reader is referred to the text of the specification for a more thorough description [5]. More information can also be obtained from the MSDL Web site: `http://www-elec.enst.fr/msdl/`.

5.3. Overview of SDL features

SDL directly extends the C-like syntax used in the MPEG-1 and MPEG-2 Technical Reports into a well-defined framework that lends itself to object-oriented data representations and machine translation. The bitstream syntax definition features of SDL are described in the form of formal grammar rules. Elementary constructs are first described, moving to composite syntactic constructs, arithmetic and logical expressions, and finally address syntactic flow control and functions. Syntactic flow control is needed to take into account context-sensitive data. Several examples are used to clarify the structure, primarily based on the MPEG-2 Video International Standard.

5.3.1. Elementary data types

SDL identifies the following elementary syntactic elements:

1. Constant-length direct representation bit fields or fixed length codes (e.g., temporal_reference). These include the encoded value as it is to be used by the decoder.

2. Variable length direct representation bit fields. These are codes for which the length is determined by the context of the bitstream (e.g., a syntactic field whose length is determined by the value of a previously parsed syntactic field).
3. Constant-length indirect representation bit fields (e.g., chroma_format or coded_block_pattern). These require an extra lookup into an appropriate table or variable, or some algorithmic processing to obtain the desired value (e.g., coded_block_pattern).
4. Variable-length indirect representation bit fields (e.g., Huffman codes for DCT coefficient run-lengths).

5.3.1.1. Constant-length direct representation bit fields. These can be simply represented as:

Rule 1

[aligned] *type*[(*length*)] *element_name* [=*value*];
 // C++-style comments allowed

The *type* is any of the familiar C/C++ fundamental data types (signed/unsigned int, char, etc.), with the addition of 'bit' for raw data. '*length*' indicates the length of the element in bits, as it is stored in the bitstream. This signals the bitstream parser to read the specified number of bits, interpret them according to the specified type, and place them in the memory area associated with the particular variable name.

The *value* attribute is only present when the value is fixed (e.g., start codes or object IDs), and it may also indicate a range of values (i.e., '0 × 01..0 × AF'). The *type* and the optional *length* are always present, except if the data are non-parsable, i.e., they are not included in the bitstream. The attribute 'aligned' means that the data are aligned on a byte boundary. As an example, a start code would be represented as

```
aligned bit(32) picture_start_code = 0x00000100;
```

An optional numeric modifier, e.g. aligned(32), can be used to signify alignment on other than byte boundary. For example, an entity such as temporal reference would be represented as

```
unsigned int(5) temporal_reference;
```

where 'unsigned int(5)' indicates that the element should be interpreted as a 5-bit unsigned integer (by default with the most significant bit first).
Note that constants are defined using the 'const' attribute:

```
const int SOME_VALUE = 255;
const bit(3) BIT_PATTERN = 1;
```

 // this is equivalent to the bitstring '001'
To designate binary values, the '0b' prefix is used, similar to the '0x' prefix for hexadecimal numbers, and a period ('.') can be optionally placed every four digits for readability. Hence 'bit(8) 0×0F' is equivalent to 0b0000.1111.

5.3.1.2. Variable length direct representation bit fields. This case is covered by Rule 1, by allowing the 'length' field to be a variable included in the bitstream, or an expression involving such a variable. For example,

```
unsigned int(3) precision;
int (precision) DC;
```

5.3.1.3. Constant-length indirect representation bit fields. Here, in addition to the actual element, one needs to define how it is mapped to obtain the actual values that the decoder will use. This can be accomplished by defining the map itself:

Rule 2

```
map MapName (output_type) {
  index, {value_1, ... value_M}, ...
};
```

The input type of such a table is always "bit". These tables are used to translate or map bits from the bitstream into a set of one or more values. The *output_type* entry is either a pre-defined type or a defined class. The map is initialized with pairs of keys and values. Keys are binary string constants while values are *output_type* constants. Values are specified as aggregates surrounded by curly braces, similar to C or C++ structures.

Here follows an example, noting that the precise definition of classes is given in the next section:

```
class YUVblocks {
    unsigned int Yblocks;
    unsigned int Ublocks;
    unsigned int Vblocks;
}
    // a table that relates the chroma format with the
    // number of blocks per signal component
map blocks_per_component (YUVblocks) {
    0b00,{4,1,1},  // 4:2:0
    0b01, {4,2,2}, // 4:2:2
    0b10, {4,4,4}  // 4:4:4
};
```

The next rule describes the use of such a map in the declaration of a variable.

Rule 3

type (*MapName*) *name*;

The *type* here is the output type defined in the map *MapName*. Example:

```
YUVblocks(blocks_per_component) chroma_format;
```

Using the above declaration, one can access a particular value of the map using the construct `chroma_format.Ublocks`.

5.3.1.4. Variable Length Indirect representation bit fields. For a variable length element utilizing a Huffman table, a similar declaration is used:

```
int(table) ac_dct_coefficient;
```

The definition of the table is done in exactly the same way as described before, but here the key entries have variable lengths. For example:

```
class val {
  unsigned int foo;
  int bar;
}
map sample_vlc_map (val) {
  0b0000.001,    {0,5},
  0b0000.0001,   {1, -14}
};
```

Very often, VLC tables are incomplete: due to the large number of possible entries, it is inefficient to keep using variable length codewords for all possible values. This necessitates the use of escape codes that signal the subsequent use of a fixed-length (or even variable length) representation. To allow for such exceptions, parsable type declarations are allowed for map values. This is illustrated in the following example:

```
map sample_map_with_esc (vlc, val) {
  0000.001, {0, 5},
  0000.0001, {1, −14},
  0000.0000.1, {5, int(32)},
  0000.0000.0, {0, −20}
};
```

As written above, when the codeword 0b0000.0000.1 is encountered in the bitstream, then the value '5' is assigned to the value of the first element (val.foo), while the following 32 bits will be parsed and assigned as the value of the second element (val.bar). Note that the order is significant. Using this construct, the complete behavior of the VLC mapping is described in a concise manner in a single place.

5.3.2. Composite data types

5.3.2.1. Classes. Equipped with the above definitions for fundamental types, the definition of composite types or objects is now examined. A very useful feature is to be able to immediately identify the type of object dealt with; object identifiers are then a particularly attractive feature. In several cases, the desire for bit efficiency precludes their use (this is the case in MPEG-2 below the slice level). The definition of a composite object can then be expressed as:

Rule 4

```
[aligned] class object_name [extends parent_class] [:
                              bit(length) [id_name] = object_id | id_range] {
  [element; ...]                         // zero or more elements
};
```

The different elements are definitions of elementary bitstream components as described in the previous section, or flow control that is discussed later on. The *object_id* is optional, and if present is the key multiplexing entity for individual objects. The *id_range* is specified as *start_id.. end_id*, inclusive of both bounds, to express that the object can have a range of possible IDs.[5] The optional 'is *parent_class*' indicates inheritance.

Example

```
class slice: bit(32) slice_start_code = 0x00000101.. 0x000001AF {
  ... // here vertical_size_extension is get, if present
  if (scalable_mode = = DATA_PARTITIONING) {
    unsigned int(7) priority_breakpoint;
  }
  ...
};
```

The order of declaration of bitstream components is important: it is the same order in which the elements appear in the bitstream. Objects can also be encapsulated within other objects. In this case, the *element* mentioned at the beginning of this section is an object itself.

[5] Note that high level objects become visible at the multiplex layer. At this time, it has not been decided if the object ID syntax described here will be used for the multiplex layer as well.

5.3.2.2. Parameter types. A parameter type defines a class with parameters. This is to address cases where the data structure of the class depends on variables of one or more other objects. The concept of parameter types is consistent with the key design principle in SDL, i.e. a declarative approach that allows conditional parsing in specifying syntax data structures. In fact, parameter types are a very intuitive facility, and, once defined, can be seen to be widely needed in resolving cross-references between objects. The syntax of a class with parameters is:

Rule 5

```
    [aligned] class object_name [(parameter list)] [extends parent_class] [:
                                        bit(length) [id_name] = object_id | id_range] {
      [element; ...]                    // zero or more elements
};
```

The parameter list is a list of type name and variable name pairs separated by commas. A class that uses parameter types is dependent on the objects in its parameter list. When instantiating such a class into an object, the parameters have to be instantiated objects of their corresponding classes or types.

Example

```
class A {
  // class body
  ...
  uint(4) format;
};
class B(A a, int i) {// B uses parameter types
  uint(i) bar;
  ...
};
class C
  {int(2)I;
  A a;
  B(a, I) foo;// instantiated parameters are required
};
```

5.3.3. Arrays

Arrays are defined in a similar way as in C/C++, i.e., using square brackets. Their length, however, can depend on run-time parameters such as other bitstream values or expressions that involve such values. The array declaration is applicable to both elementary as well as composite objects.

Example

```
unsigned int(3) length;
unsigned int(5) elements;
int(length) array[elements];
```

Here, the length of each element is obtained, as well as the number of elements from the bitstream, and then all the elements are read into `array`. SDL also allows incremental declaration of arrays; for more details, we refer the reader to the MSDL Working Draft [5].

5.3.4. Arithmetic and logical expressions

All standard arithmetic and logical operators of C/C++ are used as defined in these languages, including their precedence rules.

5.3.5. Non-parsable variables

In order to accommodate complex syntactic constructs, in which context information cannot be directly obtained from the bitstream but is the result of a non-trivial computation, non-parsable variables are allowed. These naturally follow the regular C++/Java scoping rules, and have the scope of the class in which they are defined. Note that non-parsable variables are naturally needed in flexible implementations, in which new object and method (tool) definitions are allowed.

5.3.6. Syntactic flow control

Syntactic flow control provides constructs that allow conditional parsing, depending on context, as well as repetitive parsing. The familiar C++/Java if-then-else construct is used for testing conditions. Note that since syntactic flow control occurs at the scope of class declarations, it can be easily distinguished by a software translator as referring to the syntax specification. Such structures are only allowed within method definitions in regular programming languages such as C++ and Java.
The following example illustrates the procedure.

```
map some_vlc_table(unsigned int) {
  0b0, 32,
  0b10, 33,
  . . .
};
class conditional_object {
  unsigned int(3) foo;
  bit(1) bar_flag;
  if (bar_flag) {
    unsigned int(8) bar;
  } else {
    vlc(some_vlc_table) bar;
    }
    unsigned int(32) more_foo;
};
```

Here two different representations are allowed for bar, depending on the value of `bar_flag` (note that the VLC in this case must return entries of type `unsigned int`, so that there is no conflict with the previous definition of bar). There could equally well be another entity instead of the second version (the variable length one) of `bar` (another object, or another variable).

In the same category of context-sensitive objects are the so-called repetitive objects. These simply imply the repetitive use of the same syntax to parse the bitstream, until some condition is met (it is the conditional repetition that implies context, but fixed repetitions are obviously treated the same way). The familiar structures of 'for', 'while', and 'do' loops can be used for this purpose.

To facilitate bitstream-level conditional tests, the following notation is used.

Rule 6

The construct '[bitstring]' is a test condition that is true (non-zero) if the next bits present in the input bitstream are equal to bitstring. The construct '[bitstring*]' performs the same operation, but if the string is found, the bits are removed from the bitstream.

The SDL specification defines several other minor constructs, which are omitted for brevity.

5.4. Future work

In order to assert the power and completeness of SDL, descriptions of several different audiovisual coding standards are pursued. Furthermore, work is underway to convert the entire MPEG-4 series of specification to utilize SDL for bitstream representation, thus helping to further refine the language. This process will ensure that all the necessary components are available for the codec designers arsenal. In addition, software translators are being developed that produce C++/Java code from SDL code. These tools are necessary in order to experiment with the overall SDL architecture, ensure integration with the overall MSDL environment and APIs, and solidify the syntax and grammar rules. Finally, reconfiguration of syntactic decoding based on SDL is also being developed in order to assess its potential utility within the context of MPEG-4.

6. Synchronization and multiplexing – managing time

The previous parts of this paper developed an object-oriented model of audiovisual information and its processing. A coded representation of a scene formed from a set of AV objects that is generated according to this architecture finally has to be prepared for transmission or storage in a manner suitable for synchronized, in most cases real-time, delivery and decoding. This representation of the scene is called a session, emphasizing its potentially interactive character and its finite duration. This part states the requirements on synchronization and multiplexing followed by proposals for a system decoder model, an elementary stream interface, and a two layer multiplex approach.

6.1. Requirements on synchronization and multiplexing

A wide range of requirements has been identified by the MPEG-4 project. Many of them are related to the achievable Quality of Service (QoS). They include
- support for a large and time varying number of concurrent data streams,
- support for variable bit-rate data streams,
- synchronization of data streams,
- prioritization of data streams,
- low and deterministic end to end delay,
- bounded amount of multiplex jitter,
- reliable, low bandwidth control channels,
- low residual error data channels, and
- reliable transmission error detection.

Different AV objects may have different requirements while belonging to the same application, e.g., if real-time communication and data base access are combined. The data for these concurrent sessions may be stored or transmitted jointly. Even when considering only a single AV object, it is desirable to have better error protection for crucial header information than for the remainder of the data.

MPEG-4 can be used in rather reliable communication channels like LANs as well as in error prone environments where reliable re-synchronization and other error protection features are very important. This suggests having such features configurable according to the application or even according to the needs of individual data streams. In general, it should be possible to exploit the relevant capabilities of the underlying network.

The amount of overhead must be adaptable to the bit-rate of the application. A low bit-rate mobile multimedia communication application may not tolerate 1 kbps for multiplex and stream-related information, while this may be no problem for a similar, LAN-based application.

Furthermore it is desirable for an application to have a similar interface to a broadcast and a point to point transmission channel, obviously without a return channel in the former case. Bitstream editing to extract some AV object data must be possible to some extent without parsing into an elementary stream. This requires easily accessible descriptive information about the content of the multiplexed bitstream. Additional requirements include conditional access support, feasibility of re-multiplexing, and easy interworking in heterogeneous networks.

6.2. System decoder model

The purpose of the System Decoder Model (SDM) is to provide a simplified description of the location and behavior of buffers in the system and the definition of timing information. With this model an encoder is able to monitor the decoder buffer state and knows how synchronization of data streams is achieved. The model is an adaptation of the System Target Decoder described in MPEG-2/Systems [7]. An additional model of the execution time for the various functions of an MPEG-4 decoder system would be desirable, however, it is out of scope of this paper.

6.2.1. System buffer model

Bits corresponding to coded entities that are to be presented simultaneously do not necessarily have to be in close neighborhood within a multiplexed data stream. A system buffer model is used to quantify the allowed range of this multiplex jitter as well as the induced delay.

The SDM includes the demultiplexer, adaptation layer (AL) decoders, buffers for encoded data for each elementary stream (EB), the AV object decoders, buffers for decoded data for each AV object (PB) and the compositor, as outlined in Fig. 11. Note that in general more than one elementary stream may be connected to a single AV object decoder, e.g. for scaleable coding, as indicated with AVO-Dec2, while there is always exactly one PB associated to each AV object decoder.

Processing times in all SDM elements are assumed to be zero. EB buffers are filled at the true rate of the incoming data for this stream. All incoming data are partitioned in access units (AU). AUs must have implicit or explicit decoding and presentation times. At its decoding time an AU is instantaneously removed from EB, decoded and put as a presentation unit (PU) in the associated presentation buffer. The PU may be accessed multiple times by the compositor as long as it remains in PB. The PU is removed from PB at the presentation time of the temporally subsequent PU or, if this does not exist, at the end of lifetime of the AV object.

With these model assumptions the encoder may freely use the space in the buffers. For example it may transfer data for several access units of a non-real time stream to the decoder and pre-store them in the EB some time before they have to be decoded. Then the full multiplex channel bandwidth may be used to transfer data of a real time stream just in time afterwards.

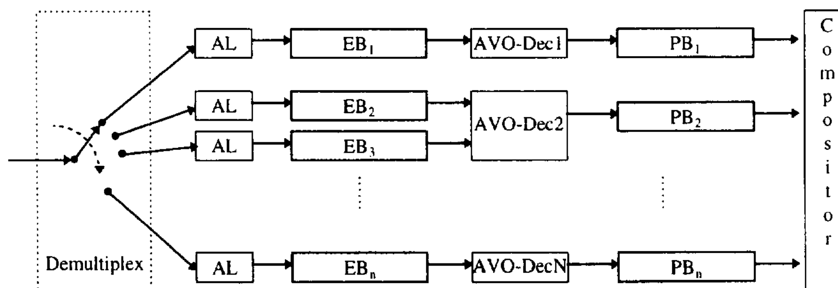


Fig. 11. System decoder model.

The PB may be used, e.g., as reordering buffer for early decoded P-frames which are needed by the video decoder for decoding of intermediate B-frames before the presentation time of the P-frame arrives. PB can also be used for any other decoded AV data that needs to be available for some longer period of time (static background images, synthetic model descriptions, etc.).

The encoder has to signal the required buffer resources to the decoder before starting the transmission. This can either be done explicitly by requesting buffer sizes for individual elementary streams or AV objects or for the whole session. It can also be done implicitly by specification of an MPEG-4 profile and level.

6.2.2. System timing model

AV objects being transmitted and presented in real time, have a timing model in which the end-to-end delay from the signal input to an encoder to the signal output from a decoder is a constant. This delay is the sum of all processing and buffering delays. Encoded data contain implicit or explicit timing information to convey the interval between successive access units. Implicit timing information could be, e.g., a constant coded frame rate.

The AV objects in one session may be encoded by different encoder systems which, in general, run at slightly different clock speeds. So, each AV object has its own Object Time Base (OTB). The speed of the System Time Base (STB) of an MPEG-4 decoder system will not be synchronized a priori with any of these OTB. Therefore, Object Clock Reference (OCR) time stamps are introduced in the bitstream to convey the speed of the encoders time base(s) to the decoder. They must be transmitted frequently enough so that the decoder can track the OTB.

Decoding Time Stamps (DTS) and Presentation Time Stamps (PTS) are used to indicate decoding and presentation time of a given access unit, measured in units of the OTB. To ensure synchronized decoding and presentation of these AV objects, all explicit or implicit timing information has to be mapped to the STB, yielding a system presentation time t_{SPT} from the known object presentation time t_{OPT} by the following formula:

$$t_{SPT} = \frac{\Delta t_{STB}}{\Delta t_{OTB}} t_{OPT} - \frac{\Delta t_{STB}}{\Delta t_{OTB}} t_{OTB\text{-}START} + t_{STB\text{-}START},$$

with

t_{SPT}	decoder's system presentation time measured in units of t_{STB},
t_{STB}	decoder's System Time Base,
t_{OPT}	AV object presentation time measured in units of t_{OTB},
t_{OTB}	AV object encoder's time base,
$t_{STB\text{-}START}$	value of decoder's STB when the first OCR time stamp of the AV object is encountered,
$t_{OTB\text{-}START}$	value of the first OCR time stamp of the AV object.

Similarly, a system decoding time t_{SDT} can be computed from the known object decoding time t_{ODT}.

Note that a system may be operated without transmission of timing information. In that case the system decoder model cannot be applied, i.e., the encoder cannot know how buffers are used and when access units are processed by the decoder. Tight synchronization is therefore not possible.

6.3. Elementary stream interface

It is desirable to define an interface for MPEG-4 data to the underlying transport layer, that is independent of this layer itself and, more precisely, hides the specifics of any transport layer from the MPEG-4 system implementation.

This interface is called the Elementary Stream (ES) Interface. All data arriving at or leaving an MPEG-4 terminal are transferred by method calls to `InElemStream` or `OutElemStream` objects, respectively. Additionally, the API for these classes has to be designed so as to provide all the necessary configuration parameters for OutElemStreams and to return all required status information in case of InElemStreams.

The precise parameters of this interface are still under investigation. Two major issues need to be resolved: how are quality of service requirements communicated and what is the minimum interface to be standardized.

A certain quality of service may be either provided by explicitly inserting adapters that perform a specific task, e.g., forward error correction, or by a more generic interface that requests the QoS in terms of, e.g., the permissible residual error rate, jitter, delay, stream priority, etc. This leaves the task to the API implementation to ensure compliance to these QoS requirements.

The OutElemStream interface may include the following methods:
- setBufferSize – sets the size of the buffer associated with this stream instance,
- put – output a number of bits or bytes to the stream,
- accessUnitStart – label this bitstream position as start of an access unit,
- decodingTimestamp – assign a decoding time to the current access unit,
- presentationTimeStamp – assign a presentation time to the current access unit,
- openChild – establish a new OutElemStream as child of the current stream,
- child – get a handle to a (already open) child of the current stream.

Calls to the put method transparently write user data while calls to the accessUnitStart and timestamp methods lead to the generation of an access unit header that is inserted in the bitstream by the multiplexer.

The InElemStream class is conceptually simpler since its main function is to allow retrieval of data. Its methods include:
- get – returns a number of bits or bytes from this elementary stream instance,
- accessUnitIndex – returns the index of the current access unit,
- decodingTimestamp – returns the decoding time of the current access unit,
- presentationTimeStamp – returns the presentation time of the current access unit,
- child – returns a handle to a child of the current stream (null if child does not exist).

Further API methods may include alignment of the read pointer to a datum boundary, methods to return the current stream buffer fullness and the state of the stream.

Elementary streams have a hierarchical relationship that is determined by the MPEG-4 content designer. The ES interface ensures that streams can be accessed by their unique hierarchical name, hiding the name spaces for logical transport channels from the MPEG-4 system. The correspondence between both name spaces must of course be conveyed from multiplexer to demultiplexer. The number and properties of the elementary streams depend on the desired number of concurrent data channels and transport characteristics. Data for one AV object may be distributed on more than one ES with different QoS, e.g., if this AV object is compressed with an algorithm that allows for scalability.

From the systems perspective, elementary streams are information entities that are structured into access units, consisting of some visible attributes, like their decoding and presentation time, followed by a number of consecutive data bits. All coded information that is intended to be manipulated or edited on a system level ('bitstream editing'), without need to parse or decode this information, has to be delivered in elementary streams. Of course, it is at the discretion of the elementary stream user to apply a proprietary multiplexing of some data streams if no system visibility of these streams is required. Compression efficiency may be a reason for this choice, however, this is not in the spirit of MPEG-4 which aims to provide easy AV object accessibility. The low overhead multiplex described next tries to support this spirit.

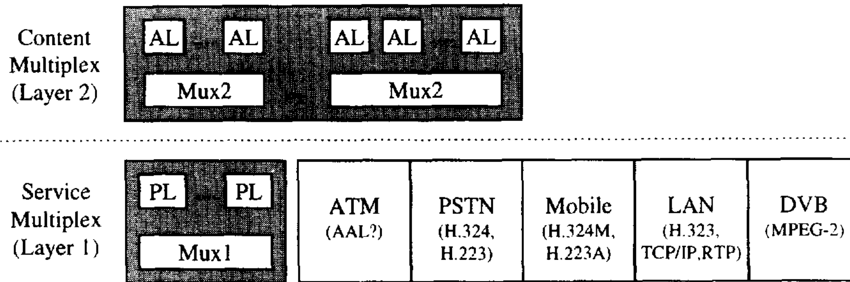


Fig. 12. Multiplex layers, including envisaged interface to other protocol stacks.

6.4. A two-layer multiplex approach

The list of requirements shows that, in terms of the OSI layer model, the multiplexing and synchronization part of MPEG-4 is centered around the transport layer (multiplexing, QoS choice) with additional session layer (synchronization) and network layer (retransmission) functionalities.

Some of these functionalities are also provided by existing transport protocol stacks for LANs, ATM, ISDN or PSTN, digital video (DVB) or digital audio (DAB) broadcast, to name a few. Some protocols take into account real time issues like synchronization (H.225 [13] for TCP/IP LANs), bandwidth resource reservation (ATM) or error resilience (H.223 Annex A [12]). All provide some kind of temporal interleaving of data streams, albeit with vastly different sizes for the data cells or packets to be multiplexed.

A two-layer multiplex approach, as depicted in the gray shaded part of Fig. 12, has been designed that separates the functionalities in order to facilitate the interface to all these transport environments. The goal is to exploit their characteristics while adding functionalities that these environments lack, and always preserving the homogeneous interface towards the MPEG-4 system as described in the previous section.

Data streams with similar quality of service requirements are first multiplexed on a content multiplex layer (Section 6.4.1) that efficiently interleaves data from a variable number of variable bit-rate streams, frames access units and contains synchronization information. A service multiplex layer (Section 6.4.2) supplies channels with a variety of quality of service and provides reliable framing of its content (the content multiplexed data) and error detection.

Appropriate protocol layers of specific networks, as shown on the lower right part of Fig. 12, may substitute this service multiplex. The protection sublayer (PL) can be seen as an example implementation of the desired quality of service functionalities that may be implemented if the underlying network protocol does not provide them.

6.4.1. Content multiplex layer

The purpose of the content multiplex layer is an efficient temporal interleaving of a varying number of variable bit-rate elementary streams with a low multiplex delay. Each elementary stream is transported in a separate logical channel.

The content multiplex packet is of variable length and consists of an index, I, the length of the payload, L, and the packet payload, as indicated in Fig. 13. In the direct addressing mode ($I \geqslant N$) the logical channel (LC) to which the complete packet payload belongs can be computed directly from the index I, as shown in Fig. 13(a). In the multiplex table mode ($I < N$) I is a pointer to a multiplex table entry $T(I)$, that defines how a payload is shared between m logical channels (Fig. 13(b)). Currently $N = 16$ values are reserved for the multiplex table mode.

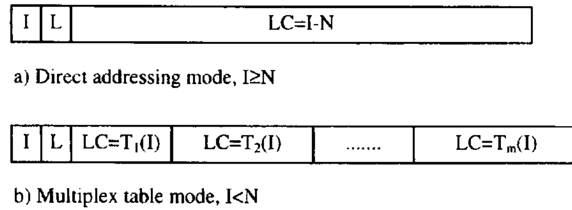


Fig. 13. Content multiplex PDU (I = index, L = length, $T_j(I)$ = multiplex table entry).

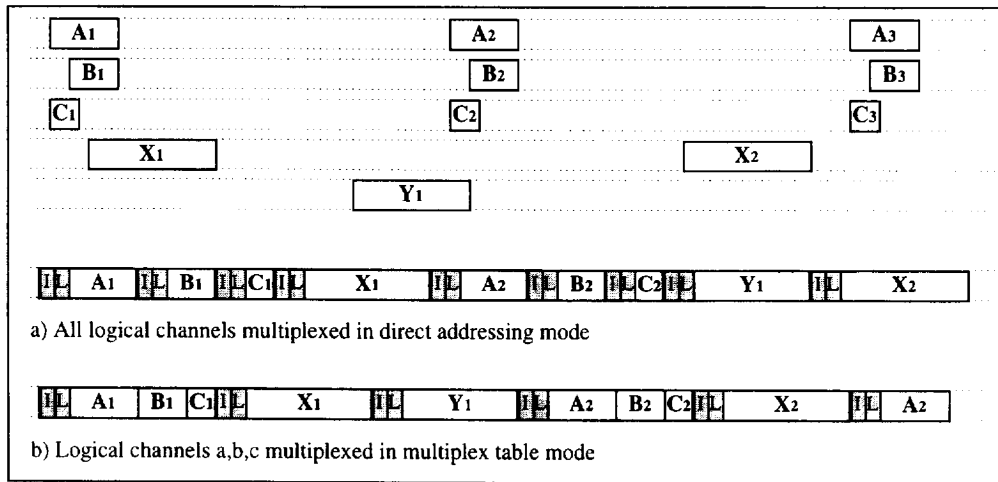


Fig. 14. Application of multiplex table mode.

A multiplex table entry $T_j(I)$, which describes how a packet is shared, has to be defined with a suitable configuration protocol before it is used. This mode, which is adapted from ITU-T multiplex recommendations H.223 [11] and H.223A [12], allows to further reduce the multiplex overhead under some conditions. This is illustrated in Fig. 14 that assumes an application with several concurrent streams, e.g., with animation parameters for synthetic objects. Each set of animation parameters may only be a few byte at a time, but to be updated regularly, e.g., every n milliseconds. Data from other streams may be interspersed. In multiplex table mode a table entry $T(I)$ may be defined to multiplex parameters for all objects in one content multiplex packet, as indicated in Fig. 14(b) for logical channels A, B and C, saving some packet headers compared to Fig. 14(a). Data in logical channels X and Y are always multiplexed in direct addressing mode.

Savings in multiplex overhead, and possibly delay, occur if data rates are sufficiently predictable. If the multiplex table entries need to be redefined often, the advantage may disappear.

Each portion of the content multiplex packet payload belonging to a distinct logical channel (A_i, B_i, etc.) may have a further adaptation sublayer header. Functionalities to frame access units, to carry the object clock reference timestamps and sequence numbering to detect lost adaptation layer packets can be configured here.

6.4.2. Service multiplex layer

The currently specified service multiplex layer is in many respects a generalization of features found in ITU-T H.223A [12]. This multiplexer has been designed for error prone channels and therefore features robust packet synchronization and constant packet length. On the protection sublayer (PL), framing, error detection and forward error correction tools using convolutional coding are included. Furthermore automatic resend requests and convolutional interleaving are available. Even more than in the original H.223A, all these functionalities are configurable.

Recent discussions show that it has to be studied for which transmission or storage media this service multiplex layer can be implemented as is. Knowing that the number of multiplexing layers should be kept small, it has to be taken into account that many networks (see introductory remarks in Section 6.4) already provide some of the functionalities suggested here. Furthermore each network has its own error characteristics and therefore needs specific tools for error protection. Therefore it can be assumed that this layer will evolve to a set of interface specifications for all those networks that are of relevance to MPEG-4.

6.5. Content management

Content management does not appear to be a multiplexer task; however, some information about the content of the multiplexed elementary streams is obviously necessary to allow object-oriented functionalities, like bitstream editing, to be performed on the MPEG-4 system level.

In MPEG-2 this task is addressed from two sides. Program Specific Information (PSI) is defined to convey a minimum of information to be able to assign elementary streams to their respective recipients. The unique assignment between elementary streams and AV objects is implemented in MPEG-4 by the convention how to access elementary streams via the ES interface in conjunction with a transmission of the necessary correspondence information, which is much alike MPEG-2 PSI, except that no semantic information about the content of the elementary streams ('descriptors') is conveyed here. These descriptors in MPEG-2 are seen as a potential source of ambiguity, as they mostly duplicate information that is already present in the elementary streams themselves.

In MPEG-2, as a second step, a more exhaustive content management is realized via an additional protocol, DSM-CC [10], that allows for, e.g., the selection of programs in a Video on Demand application. While this functionality is only useful for part of the digital TV broadcast applications that make use of MPEG-2, it is much more central for MPEG-4 that is supposed to have a focus on interactivity. Significant activity is still needed to establish a framework, probably based on DSM-CC, to describe the content of multiplexed MPEG-4 streams as well as to embed protocols to configure MPEG-4 communication sessions and to implement user interactivity.

6.6. Future work

The proposed synchronization and multiplexing framework is based on a homogeneous elementary stream interface towards the MPEG-4 system, combined with a system decoder model, comparable to MPEG-2/Systems but modified according to the MPEG-4 requirements. Underneath the elementary stream interface a two-layer multiplex approach is implemented, separating a flexible content multiplex from a service multiplex layer.

The work on this part of MPEG-4 Systems is still very new, so that this outline, even more than the other parts, can only be seen as a snapshot of on-going work. Future work has to address implementation and testing of the layered multiplex specification and a refinement of its functionalities. Adaptation to relevant

networks has to be defined. A major task, not only in the multiplex area, will be the definition of a description and signaling framework, that could also be used for multiplexer and session configuration. In this context also the homogeneous integration of broadcast and point-to-point applications will be further pursued.

7. Conclusion

The MPEG-4 Systems and Description Languages is based on contemporary object-based technology. Unlike previous digital or analog video systems, the MPEG-4 data representation is based on natural audiovisual objects, not the (unnatural) audio and video frames used for presentation. So MPEG-4 Systems defines AV Objects that contain data and the methods to render and composite the objects. AV Objects are hierarchical in general. The top object in a given audiovisual hierarchy is the scene object. The generic MPEG-4 scene is a 3D space of arbitrary size changing dynamically over time. Presentation of this scene consists of defining the viewport into the scene, rendering each object and compositing the objects onto this viewport. MPEG-4 Systems provides the structures and interfaces to support this model.

In addition, efficient coding of the AV objects is required, so process objects are defined to perform this function. The coded objects are converted to a bitstream format that can be configured by the Syntactic Description Language, and finally, the objects and associated control data are multiplexed together for storage or transmission.

As a result, MPEG-4 Systems provides an environment with considerable flexibility. This makes it adaptable to many applications, and allows modification of individual components such that they may be tailored to specific operating conditions. The environment is also highly interactive. Separation of the presentation from the coding structure allows interactive selection of the presentation viewport. The natural object structure allows interactive manipulation of the audiovisual objects.

MPEG-4 Systems attempts to provide a solution where the minimum is specified. It defines just the underlying data structures, and the APIs for each component of the complete solution. In this way, maximum freedom is provided to the algorithm, architecture and system designers.

Acknowledgements

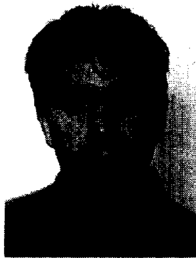
The authors would like to express their deep appreciation to the MPEG-4 Systems sub-group. The major part of the presented material has been discussed and constructed from the various contributions made within the context of this group, including many fruitful discussions with MPEG-4 experts. The authors hope that this paper faithfully represents the work and the spirit of this group, founded on cooperation and the open exchange of ideas, and that it will serve as a good vehicle for promoting the understanding and further development of MSDL.

References

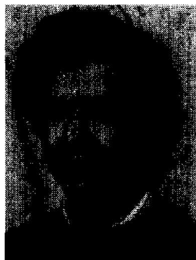
[1] ISO/IEC JTC1/SC29/WG11 N1495, MPEG-4 Requirements Version 1.2, Maceiò, Brasil, November 1996.
[2] ISO/IEC JTC1/SC29/WG11 N1378, MPEG-4 Audio Verification Model 2.0, Chicago, USA, September–October 1996.
[3] ISO/IEC JTC1/SC29/WG11 N1469, MPEG-4 Video Verification Model 5.0, Maceiò, Brasil, November 1996.
[4] ISO/IEC JTC1/SC29/WG11 N1454, MPEG-4 SNHC Verification Model 5.0, Maceiò, Brasil, November 1996.
[5] ISO/IEC JTC1/SC29/WG11 N1483, MPEG-4 Systems Working Draft Version 2.0, Maceiò, Brasil, November 1996.
[6] ISO/IEC JTC1/SC29/WG11 N1484, MPEG-4 Systems Verification Model Version 2.0, Maceiò, Brasil, November 1996.
[7] ISO International Standard 13818-1, MPEG-2 Systems, 1994.
[8] ISO International Standard 8824, Information Processing Systems – Open Systems Interconnection – Specification of Abstract Syntax Notation One (ASN.1), 1990.

[9] ISO International Standard 8825, Information Processing Systems – Open Systems Interconnection – Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1), 1990.
[10] ISO International Standard 13818-6, MPEG-2 Extension for Digital Storage Media Command and Control (DSM-CC), 1996.
[11] ITU-T Recommendation H.223, Multiplexing Protocol For Low Bitrate Multimedia Communication, 1996.
[12] ITU-T Draft Recommendation H.223 Annex A, Multiplexing Protocol for Low Bitrate Mobile Multimedia Communication, 1996.
[13] ITU-T Draft Recommendation H.225, Media Stream Packetization and Synchronization on Non-Guaranteed Quality of Service LANs, 1996.
[14] Rumbaugh et al., Object-oriented Modeling and Design, Prentice-Hall, Englewood Cliffs, NJ, 1991.

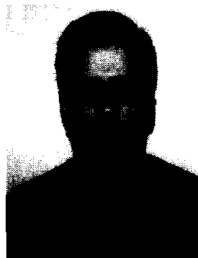
Biographies



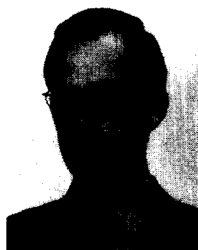
Olivier Avaro was born in Provence, France on 27 July 1968. He received his Dipl. Ing. degree in telecommunication engineering in 1992 from the Higher School of Telecommunication of Brittany. After joining France Telecom-CNET department on image communication in 1992, he worked on image representation techniques and analysis. His research areas cover video compression algorithms, error resiliency of video compression algorithms, invariant representation of images and shapes and model based representation for interpersonnal communications. He has been early involved in the ISO/MPEG-4 project, in particular through the European platform MAVT and its successor MoMuSys. He is currently chairman of the MPEG-4 Systems subgroup.



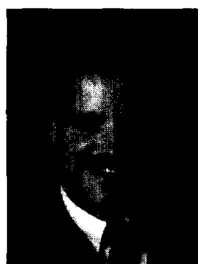
Philip A. Chou was born in Stamford, CT on 17 April 1958. He received the B.S.E. degree from Princeton University in 1980 and the M.S. degree from the University of California, Berkeley, in 1983, both in electrical engineering and computer science, and the Ph.D. degree in electrical engineering from Stanford University in 1988. From 1977 through the present, he worked for IBM, AT&T Bell Laboratories, Princeton Plasma Physics Lab, Telesensory Systems, Speech Plus, Hughes, and Xerox, where he was involved variously in office automation, motion estimation in television, optical character recognition, LPC speech compression and synthesis, text-to-speech synthesis by rule, compression of digitized terrain, speech and document recognition, and image and video compression. His research interests are pattern recognition, data compression, and speech, image, and video processing. He is the recipient, along with Tom Lookabaugh and Robert M. Gray, of the 1993 IEEE Signal Processing Society paper award. He has served as a guest editor of the IEEE Transactions on Image Processing, and as a consulting associate professor for Stanford University in 1995. Currently, he is with the Xerox Palo Alto Research Center in Palo Alto, CA. Dr. Chou is a member of Phi Beta Kappa, Tau Beta Pi, Sigma Xi, and IEEE Computer, Information Theory, and Signal Processing societies.



Alexandros Eleftheriadis was born in Athens, Greece, in 1967. He received the Diploma in Electrical Engineering and Computer Science from the National Technical University of Athens, Greece, in 1990, and the M.S., M.Phil. and Ph.D. degrees in Electrical Engineering from Columbia University, New York, in 1992, 1994 and 1995 respectively. Since 1995 he has been an Assistant Professor in the Department of Electrical Engineering at Columbia University, where he is leading a research team working in the areas of video signal processing and compression, video communication systems (including video-on-demand and Internet video), distributed multimedia systems, and the fundamentals of compression. During the summers of 1993 and 1994, he was with AT&T Bell Laboratories, Murray Hill, NJ, developing low bit-rate model-assisted video coding techniques for videoconferencing applications. From 1990 until 1995, he was a Graduate Research Assistant in the Department of Electrical Engineering at Columbia University. Prof. Eleftheriadis has served as a guest editor, committee member, and organizer for several international journals and conferences. He is a member of the ANSI X3L3.1 Committee, and is participating in the ISO/IEC JTC1/SC29/WG11 (MPEG) standardization activity as well as DAVIC. Prof. Eleftheriadis is a member of the IEEE, the ACM, and the Technical Chamber of Greece.



Carsten Herpel was born in Cologne, Germany on 28 September 1962. He received his Dipl. Ing. degree in electrical engineering in 1988 from the University of Aachen. After joining THOMSONs research facility in Hannover, Germany, he worked on video compression algorithms, co-developing an algorithm proposal both for ISO/IEC's MPEG-1 and MPEG-2 standardization projects. This work was embedded in the European COMIS and VADIS projects as well as in the German HDTVT project, where his MPEG-2 work focussed on development and comparison of hierarchical coding approaches for terrestrial HDTV broadcast systems. He currently works on Systems issues in the ISO/MPEG-4 project.



Cliff Reader has over 20 years experience in digital video coding, image processing, and real-time digital video systems design. His research explored the use of transform coding for video in the early '70s. He has developed products for various imaging markets, most recently concentrating in the emerging field of Consumer Digital Video. Since 1990 he has been an active member of the ISO/IEC MPEG community, including being Head of the US Delegation for two years, and leading the MPEG-4 activity from inception. He is also the technical expert on MPEG intellectual property, assisting in the establishment of the MPEG Patent Pool. Dr. Reader has been with Samsung Semiconductor for the past three years as Associate Director for Strategic Marketing, with responsibility for the MSP product.

Julien Signès received his Engineer Degree from Ecole Polytechnique in Paris in 1992, then he joined Ecole Nationale des Telecommunications in Paris where he obtained a Telecommunication Engineer degree. He has been working at CCETT (Joint research center for broadcast and telecommunications) in Rennes as a member of the Corps Interministeriel des Télécommunications since 1994. His research areas cover Image and Video coding, as well as object oriented design for image processing and multimedia applications, in the context of European projects and ISO/MPEG-4 standardization group.