AN ARCHITECTURE FOR MHEG OBJECTS

A THESIS SUBMITTED TO

THE UNIVERSITY OF KENT AT CANTERBURY

IN THE SUBJECT OF COMPUTER SCIENCE

FOR THE DEGREE

OF DOCTOR OF PHILOSOPHY.

By

Eduardo Simões de Albuquerque

October 1995

Contents

List of Tables

List of Figures

To my family

Abstract

Hypermedia applications are one of the most recent and most demanding computer uses. It is accepted that one of the main impediments to their widespread use is the lack of standards, and the lack of *Open Systems* with the possibility of having documents interchangeable between different hardware and software platforms.

Several standards are emerging, one of which is the one being developed by the ISO/IEC WG12 known as the Multimedia and Hypermedia Information Coding Expert Group (MHEG).

As desktop systems become more powerful, one of the main users of hypermedia applications is the home market. Therefore it is important to have standards and applications suitable for those platforms.

This work reviews existing proposals for hypermedia architectures and interchange standards. It then assesses the suitability of the MHEG standard for use in open, distributed, and extensible hypermedia systems. An architecture for the implementation of MHEG objects taking into account the limitations imposed by current desktop computers is also proposed.

To assess the suitability of the proposed architecture, a prototype has been implemented. An analysis of the performance obtained in the prototype is presented and conclusions on the requirements for future implementations drawn.

Finally, some suggestions to improve the MHEG standard are made.

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Professor Peter Linington, for his guidance, constructive criticism, constant attention, encouragement and support, and for his example of hard working and discipline which were fundamental to the development of this work.

I would also like to thank Professor Peter Brown for his encouragement and for the privilege of having him as my supervisor in my first year at the University.

I am grateful to Djamel Sadok, Paulo Pinto, Chris Scott and Fred Cole for their valuable help during the early stages of this work.

Many friends and colleagues have made my stay in the Computing Laboratory a constant joy. I would specially like to thank Eduardo Rojas-Vega, Helena Rodrigues, Geraldina Fernandes, Maria Pimentel, Carlos Ferraz and George Justo with whom I shared an office. I would like also to express my gratitude to the Brazilian community in Canterbury (too many to name!) for their comradeship and help.

I acknowledge the financial support of Conselho Nacional de Pesquisas (CNPq) and Universidade Federal de Goiás, Brasil.

And last but not least, I thank Ana, for always being supportive and for sharing all those difficult moments.

Declaration

No portion of the work referred in this thesis has been submitted in support to an application for another degree or qualification at this or any other university or other institution of learning.

Chapter 1

Introduction

1.1 Hypertext, Multimedia and Hypermedia

Although *hypertext, multimedia and hypermedia* have been used for a long time, and are buzzwords, it is still necessary to define their meanings. We could start by defining the components that make up these words:

Text: One of the definitions for *text* in the *Oxford Concise Dictionary* is:

> **[2]** main body of book opp. to notes, pictures, etc.

We can use a broader definition: text can be seen as a *body* of recorded information. Text and document are synonyms and, although they contain basically natural language, they can also have images (contrary to the above definition). Books, recipes, articles, software documentation, for example, are special kinds of text [Rada, 1991].

Medium: *Medium* is defined as:

> **[6]** Means by which something is communicated. Material or form used by artist, musical composer, etc.

The above definition is too broad and, for our purposes, we need to specialise it. The MHEG [MHEG, 1995a] standard gives definitions for several applications of media such as:

- *Presentation Medium:* The means used to reproduce information to a user (output device) or to acquire information from a user (input device).
- *Representation Medium:* The type of interchanged data, which defines the nature of the information as described by its coded form.
- *Storage Medium:* The means used to store information.

Multi: Again from the *Oxford Concise Dictionary*, the prefix *multi-* means:

> *comb. form* many.

Hyper: The prefix *hyper-* means:

> *pref* with senses 'over, beyond, above' *(hypergamy, hyper physical)*, 'exceeding' *(hyperbola, hypersonic)*, 'excessive, above normal' *(hyperbole, hypersensitive)*; opp. HYPO

Using the above definitions, we define:

Hypertext: From the definitions of *hyper* and text, we can conclude that hypertext goes beyond the concept of text. While *text* presents only one dimension (linear), a hypertext has more than one dimension that Contains relationships amongst *texts*. The usual example is an encyclopedia where each keyword is also a link to an encyclopedia entry. In an encyclopedia, we can also see two levels of linking: one is explicit, when the authors make an explicit cross reference; the second one is implicit as the words used to define a term also have an entry in the encyclopedia.

Multimedia: something that is expressed using more than one perceived medium, where "medium" has the sense defined above.

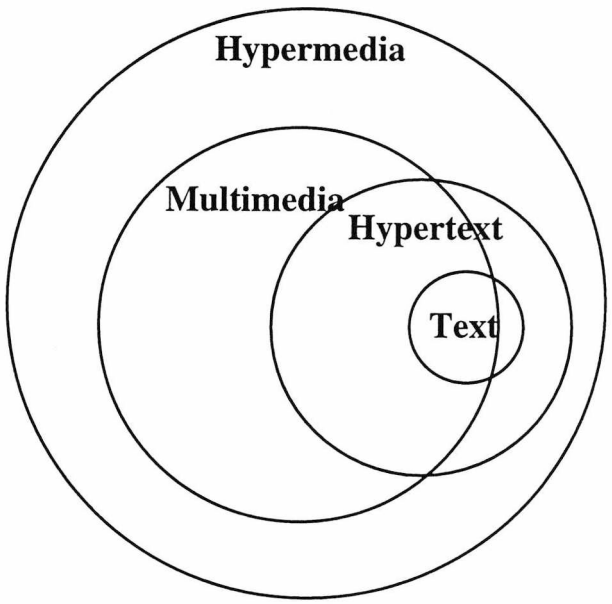

Figure 1.1: *Text, hypertext, multimedia and hypermedia*

Hypermedia: using the same definition we used for *hypertext*, *hypermedia* can be defined as *multimedia* plus an extra dimension that maintains the relationship between the media. *Interactive Multimedia* (another buzzword) is an example of *hypermedia*.

Graphically, we could see the above definitions as in figure 1.1. The kernel of the figure is the *text*; when we add relational structure to text we have *hypertext*; when more than one *perceived medium* is used we have multimedia and finally we can add relations to the media and we have *hypermedia* (or *interactive multimedia*).

1.2 Objectives

The objective of this work is to analyse the practical use of Multimedia Interchange standards, examine their suitability for use in *Multimedia Systems* and to propose an architecture to be implemented in distributed desktop computers. The standard used for the prototype implementation was MHEG which is described in detail in Chapter 3.

1.3 Related Work

Current hypertext/hypermedia systems can be broadly divided into categories depending on their usage [Rada, 1991]:

- *Small volume hypertext:* the document is self contained, with explicit links between components. Most of the early hypertext systems fall in this category (eg. Augmentation System [Howard Rheingold, 1985], ZOG [Akscyn and McCracken, 1984] and Guide described in section 1.3.2);

- *Large volume hypertext:* in this category, systems emphasizes linking between documents and not inside the document itself. Usually many users have documents within the system and there is an institution responsible for maintaining the whole structure. Examples of systems in this category are the pioneer Memex [Bush, 1945, Bush, 1967] and Xanadu [Nelson, 1987].

- *Collaborative hypertext or grouptext:* these are the systems that provide a framework for activities where interaction between users and collaboration are required. Examples of such systems are the Augmentation System and gIbis [Conklin and Begeman, 1987].

- *Intelligent hypertext:* in an intelligent hypertext, expertise is transferred to a knowledge base with an inference mechanism. The transference happens by storing knowledge in links and allowing the links to trigger arbitrary computation. Intelligent systems can fall into any of the previous categories.

Systems being developed today tend to present features that fall into all the categories above, with the ability to deal with more media than just plain text. In the following sections, we present some current hypermedia systems. We also present some attempts to standardise multimedia exchange and separate multimedia data and presentation information. A deeper discussion on multimedia interchange standards is presented in Chapter 2.

1.3.1 MAEstro

The MAEstro Multimedia System [Drapeau and Greenfield, 1991a, Drapeau and Greenfield, 1991b] is a Multimedia Authoring System initially developed at Stanford University, and now produced commercially. The system was designed for extensibility, making it easy to add support for extra media.

The system is made up of four logical components:

1. Media Editors;
2. An authoring application;
3. An inter-application messaging system;
4. The PortManager Application.

Media Editors

The *media editors* are the applications that directly control media. Each application is responsible for one medium.

The system includes the following media editors:

- *QuoteMaker* for working with text and titles. QuoteMaker can work with either ASCII text or scanned material captured through the ImageEdit editor
- *cdEdit* for controlling and editing music on CDs. cdEdit incorporates music and sound from digital audio compact discs (CDs) into a multimedia presentation. It also controls the CD player;
- *VideoEdit* for controlling and editing video disks. It captures and manipulates video information from video disc players;
- *DTR (Digital Tape Recorder)* for recording and editing sound via the workstation's built-in digital audio capabilities. DTR plays back digitised audio using the

built in audio capabilities of the workstation. DTR accepts sound from audio tape, microphones, VCR or videodisk and CDs. It can output audio to the SPARCstation's internal speaker, headphones or an external speaker;

- *ShellEdit* is a tool for selecting, timing and executing UNIX shell commands that are to be incorporated into a time-line presentation;
- *vcrEdit* used to control the NEC PC-VCR, a computer controlled VHS videotape player;
- *vcrDub* used to record video segments from one tape to another, and to re-arrange the order of those segments;
- *ImageEdit* for showing images as part of a multimedia presentation. ImageEdit supports the GIF and TIFF image formats;
- *TimeLine* builds the Multimedia presentation.

The Authoring Application

The TimeLine Editor Application presents documents as a number of "tracks" of time, one track for each medium in the document. The TimeLine does not directly control media but controls the actions of the media editors which do the actual media manipulation.

The Inter-application Messaging System

The MAEstro messaging system is implemented with Sun Remote Procedure Call (RPC). Each application in the MAEstro environment uses the messaging system for communication with other applications. A typical use of the protocol is for an authoring application to request a media editor to open a document, select part of that document, and play that selection.

The Port Manager Application

The PortManager serves as a rendezvous point for applications that wish to communicate with each other. It listens to a TCP/IP protocol port for messages from applications that wish to advertise their services, and keeps an internal list of the TCP/IP port numbers passed in by the registering applications.

All applications in the MAEstro environment use the same set of RPCs, making it easy for applications to communicate with new applications added at any time. Applications can become aware of new services by querying the PortManager.

Problems of the model

The MAEstro environment presents some problems:

- MAEstro does not address the problem of synchronisation of media and guaranteed network delivery of continuous media. The environment is at the mercy of slow media.

- The model does not allow the development of interactive applications. The author defines how a presentation should happen but the model does not provide for objects capable of receiving user input (eg. to follow a hypermedia link). The system is, therefore, *multimedia* and not *hypermedia* and this limits its practical usage.

- Documents in MAEstro tend to have too many components. This occurs because each piece of information to be presented must be stored in a file, and the model does not allow the use of parts of a file. In a presentation where there are, for example, 100 lines of text to be used as captions, there will be 100 small files to be maintained.

- It does not address security issues.

1.3.2 Guide

The Guide system [Brown, 1987c, Brown, 1986, Brown, 1987a], one of the first hypertext systems, started as a research project at the University of Kent at Canterbury in 1982 by Professor Peter Brown. Guide was to be used as a tool for reading documents in a computer. In Guide, there is no difference between author and reader to encourage users also to be authors of the documents they are reading.

Since 1987 there has been a commercial version of Guide [OWL International, Inc, 1988, OWL International, Inc, 1992a] (which is the one described), implemented and sold by *Office Workstations Limited (OWL)* (now InfoAccess Inc.) Guide is available for Macintoshes and IBM-PC compatible machines. There is also an UNIX [Brown, 1987b] version of Guide used at the University of Kent.

A Guide document has an implicit tree structure that is invisible to the user who sees the document as continuous and linear with the possibility of *changing routes* within the document. Guide has a very simple user interface and does not demand a steep learning curve for its new users.

Guide includes a full scripting language, LOGiiX [OWL International, Inc, 1992c, OWL International, Inc, 1992b] that allows access to its hypertext engine. LOGiiX allows, for example, the association of conditions to links providing for the creation of *one-to-many* links. A LOGiiX script can also make use of Windows Dynamic Data Exchange (see section 4.5.2) which allows links to have as destination any DDE-aware application. LOGiiX also includes features for:

- Arithmetic and logical operations;
- Text string functions;
- Access to all Guide menu commands and some dialog box commands;
- Guide document manipulation;
- Looping constructs;

- Access to objects in Guide documents;
- File input and output.

In Guide, "just about anything that can be selected with your mouse can be made into an object" [OWL International, Inc, 1992a].

Guide objects are made up of three components (figure 1.2):

1. *Data component:* the *text* or *graphic* that appears on the screen when the object is displayed. It can also be a LOGiiX program;
2. *Presentation Attributes:* determine how the data is displayed and include text styles and color. The *Presentation Attributes* apply to *text objects.*
3. *Behavioral attributes:* define the events that take place when an object is displayed or activated using the mouse.

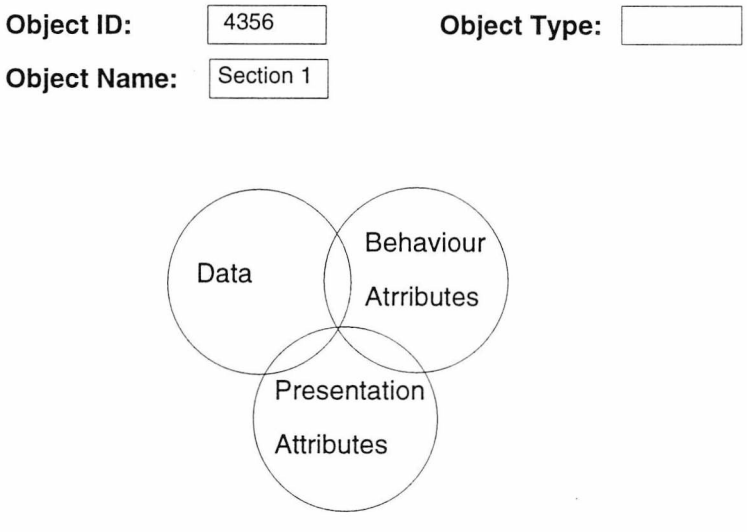

Figure 1.2: Components of a Guide Object

Several tools for converting SGML documents and documents created using other word processors to Guide are also available on the market.

Limitations of Guide

Guide is based on the *book* metaphor, where each document is, to a certain extent, self contained. It therefore does not provide facilities for network access, which is a common problem in most early hypertext/hypermedia systems.

The design of Guide was focused on text and although extensions are available to provide access to the multimedia features existing in the Windows environment through the *Media Control Interface* (MCI), such as video and sound, the author does not have full control over them.

1.3.3 Idex

Idex [Woodhead, 1991], sold by the same company that makes Guide, is its *in-the-large* version. Idex runs on a network of computers and it has facilities for document management, allowing authorship and reading in one or more shared collections of documents. Idex allows processes such as document conversion, indexing and retrieval of information. Idex has a layered structure that allows the substitution of compatible processes for those implemented as standard, such as the retrieval engine [McAleese, 1993]. At a higher level, it also allows the generation of tables of contents, tables of figures, glossaries and citations [William, 1991].

Idex has some of the security features that are usually found in database management systems. Documents are stored in a way similar to a library, and not directly in files/directories; this allows users to focus their attention on the contents rather than the location of the material.

1.3.4 Microcosm

The Microcosm [Fountain *et al.*, 1990, Hill and Hall, 1994, Davis *et al.*, 1994] hypertext system started as a research project at Southampton University and is now available commercially for IBM-PC platforms, and versions for Apple Macintosh and Unix

machines are being developed.

The Microcosm project aimed at developing and creating methods for hypertext authoring on a large scale. Microcosm was designed around the following set of principles:

- *No distinction between author and user:* all users are allowed to build links;
- *Loosely coupled system:* Microcosm is built up from a set of communicating programs (or tasks) with a low level of interdependency in such a way that it is not difficult to couple any other program into it.
- *Modularity:* as a research project, the idea was to have a design where elements (such as a document viewer) could be replaced;
- *Separation of links from data objects:* data objects and information describing their relationships are kept separate from each other, allowing the definition of different levels of abstraction for the same data.

Structure of Microcosm

Microcosm is made up of a set of autonomous processes that communicate with each other using a message passing system. The main task in Microcosm is the *Document Control System* (see figure 1.3) which is responsible for opening documents, routing messages and supporting links. This task is not visible to the user, but in a session using Microcosm there must be at least one visible window, a *Document Viewer* that displays a document on the screen. There is a viewer for each type of document (text, graphics, etc) and they communicate by sending messages that are routed through the Document Control System.

The viewers are divided into three categories depending on how "aware" of the system they are:

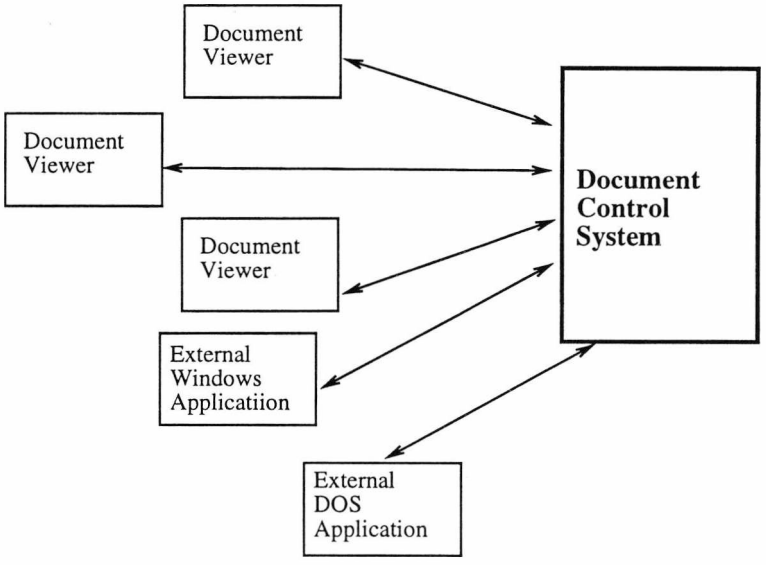

Figure 1.3: Microcosm tasks (from [Fountain *et al.*, 1990])

1. *Fully aware:* a viewer that has been written to communicate with Microcosm providing a bi-directional message channel with Microcosm;

2. *Partially aware:* a viewer that can be customised by the user, such as Word for Windows. In general, any DDE (see section 4.5.2) aware application can be made partially aware of Microcosm.

3. *Unaware:* a unaware viewer is one that has no direct communication with Microcosm, although it can be launched from Microcosm and some communication via the clipboard (see section 4.5.4) is still supported.

The filter layer in the Microcosm architecture (figure 1.4) is composed of processes, possibly chained, that receive messages, take any appropriate action and pass the message on to the next filter [Hill *et al.*, 1992]. Among the existing filters are:

- *Linkbases:* that hold information referring to links. As more than one linkbase may be installed, different views of the document can be provided;

- *Show links:* a filter that shows the links that do not have an anchor in the viewer; this is particularly useful with unaware viewers;

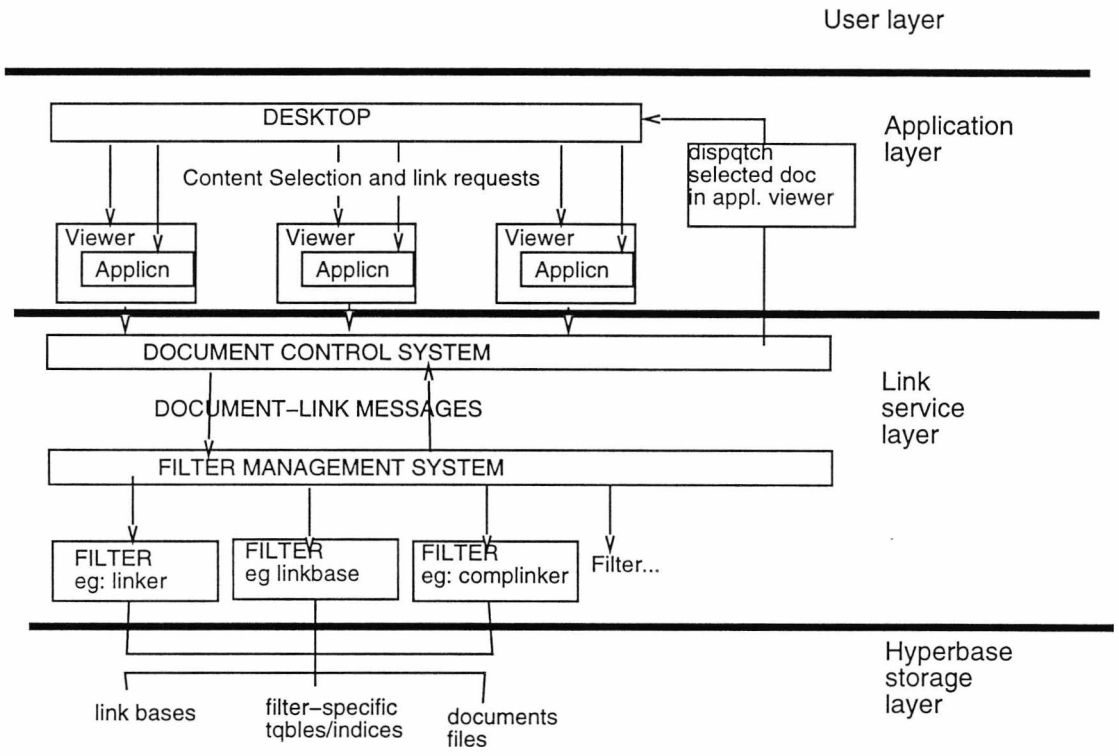

Figure 1.4: The Microcosm Model (from [Multicosm, 1994])

- *Compute links:* that creates links automatically based, for example, on statistical analysis of the content;

- *Navigational aids:* that include a history mechanism and a *mimic* filter that allows the user to follow a pre-defined tour, to deviate from the tour and return to the point that they left previously.

Final remarks

The Microcosm project is very recent and the developers had the opportunity to evaluate existing systems, such as Guide and Idex. Since Microcosm was developed with automatic authorship in mind, it is very powerful in converting existing documents to its format, and it also can make use of existing tools.

However, in spite of its recent development, Microcosm does not include facilities for distribution and the group responsible for its development it is currently researching its integration with the World Wide Web [Carr *et al.*, 1995].

1.4 HyperCard

HyperCard is probably the best known hypermedia system today. Part of its success is due to the fact the Apple has shipped a free copy with every Macintosh sold since 1989 when it was created.

HyperCard was initially projected as a graphical environment, not as a hypermedia system, and many of its applications are not *hyper*media as defined early in this Chapter. HyperCard is based on a *card metaphor* and a document is called a *stack*. Every stack starts with a special card called *home* and the navigation depends on definitions by the author [Danny Goodman, 1987].

HyperCard has an associated scripting language (hyperTalk), that can be used either to generate scripts or to enter direct commands via a window (*MessageBox*). In both cases, the script is interpreted. The whole system can in fact be seen as a high level programming language, structured in blocks that manipulate a hypermedia environment.

HyperCard has a hierarchical structure of object categories:

1. *Card:* the card is the central object in the system. Each card may have an individual lay-out and can perform graphical, computational, textual, QuickTime video (see section 2.10) and audio functions;

2. *Background:* a background is above a set of cards in the hierarchy. Several cards may have a common background which can also have the attributes that are associated with a card;

3. *Stack:* a stack is a collection of cards and backgrounds. It is also possible to define attributes common to all elements in a stack;

4. *Home stack:* the home stack is a special type of stack. It is the first one activated when HyperCard is executed. The system requires the existence of a home stack even if it consists only of an empty card. The home stack may be used as an index to other stacks, in which case it must have anchors to trigger the activation of the

other stacks.

5. *HyperCard:* this is the highest level in the hierarchy. Messages that were not dealt with in the lower levels must be processed by *HyperCard*. If the message is not understood at this level, the system opens a dialog box informing the user.

There are also some more basic objects used in cards: *buttons* and *fields* that can have a personalised layout and can have associated scripts. Buttons are objects that usually deal with mouse events while fields are optimised for keyboard textual input.

All HyperCard anchors are graphical. To create a reference from a text, it is necessary to define a rectangle delimiting the text and this prevents the anchor from being edited in the future.

The main *hyper*-characteristic of HyperCard is the possibility of associating scripts with anchors [Jakob Nielsen, 1990]. Scripts are activated by events such as a mouse click, or when the cursor enters a region or some temporal event is triggered.

HyperCard takes advantage of the Macintosh graphical interface, allowing links between text and images.

Limitations of HyperCard

HyperCard does not provide facilities for converting existing documents into its format. It has been used mainly to create small document. Like Guide, HyperCard was not designed to run on a network or in a distributed environment.

1.5 Pandora

Developed at Olivetti Research Laboratory in Cambridge, UK, Pandora [Hopper, 1990, Tebbutt, 1991, Jones and Hopper, 1993] was created to demonstrate the practicability of adding real-time audio and video to the desktop via a general purpose *Asynchronous Transfer Mode* (ATM) [Handel, 1991] network.

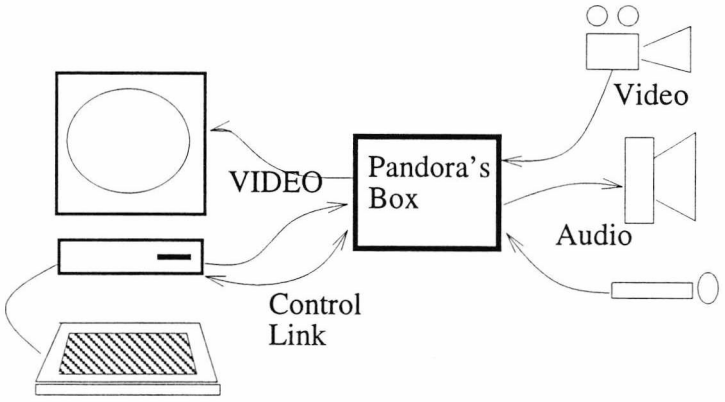

Figure 1.5: Pandora Connections (from [Jones and Hopper, 1993])

Pandora is based on a subsystem that handles the multimedia peripherals (the Pandora's box) as shown in figure 1.5. The Pandora box can be controlled by the host workstation; it can intercept the workstation's video output and add material from its own framestore, digitise video from a camera, accept telephone or microphone inputs, send output to telephones or loudspeakers and transmit and receive streams over a dedicated ATM network connection.

The design of Pandora was aimed at allowing the highest possible loads, and to achieve that goal it was based on the following principles:

- *Outgoing priority:* a box that is overloaded should be the first to notice the degradation. Therefore incoming traffic is degraded first;
- *Àudio priority:* it is better to have a lower quality video than degraded audio;
- *New stream priority:* as users do not have to shutdown one stream before starting a new one, if necessary, older streams are degraded first;
- *Command priority:* commands should be executed even if time-critical activities exceed system capacity;
- *Upstream independence:* when a stream is copied to more than one destination, it is desirable that overload of one of the destinations should not affect the quality of the other ones;

- *Continuity during reconfiguration:* the operation of adding or removing a destination to/from a stream should not disturb the other recipients of the stream (eg. by causing jitter on the video);
- *Minimize delay:* delays in the data streams should be kept to a minimum to keep audio and video synchronised and to avoid echoes in the audio stream;
- *Local adaptation:* decisions related to buffering and discarding data should be made locally depending on conditions such as bandwidth and critical times.

As it stands today, Pandora allows a workstation to be used as a videophone or as a video conferencing station, and supports video e-mail.

Limitations of Pandora

Pandora is a recent project and makes use of state-of-art technology. It provides a framework upon which higher level applications can be built. It does not make use of standards at the document level such as MHEG (see chapter 3), and there are no reported uses to date to assess its viability to integrate large applications involving highly interconnected documents.

Pandora also requires a large investment in terms of dedicated hardware.

1.6 Thesis outline

This chapter presented an overview of the areas influencing this work, examples of related work and the motivation for developing this thesis.

Chapter 2 presents an overview of the state of the art in multimedia interchange and the related standards.

Chapter 3 presents the MHEG standard on which this work is based. It is important to note that during the development of this thesis, MHEG was still being defined. The implementation described in chapter 5 incorporates the concepts present in the versions

published in 1993 [MHEG, 1993] and 1994 [MHEG, 1994a, MHEG, 1994b]. Therefore this work uses many ideas introduced by MHEG but it is not compliant with the 1995 version of the standard, and the description (markup) language used is not the one defined by the standard but one intended to be closer to the LaTeX language, which is more like the English language.

Chapter 4 presents the requirements for a distributed hypermedia system capable of handling portable *documents*.

The requirements presented in chapter 4 are used as guidelines for proposing an architecture for implementing MHEG objects in chapter 5. This chapter also describes the implementation of a prototype of the architecture proposed.

Chapter 6 presents performance measurements obtained from the prototype, and discusses several strategies for improving performance, depending on the types of objects and interaction being processed.

Chapter 7 presents an analysis of the MHEG standard and its suitability as a basis for distributed multimedia systems. Enhancements to MHEG are also proposed in this chapter.

Finally, chapter 8 summarises the conclusions, comments on the decisions taken for the distribution and scheduling of processes and makes some further remarks on the MHEG standard.

Chapter 2

Multimedia Interchange

2.1 Introduction

The Computer industry has been driven by technology rather than by the requirements of its market [Gray, 1991]. Unlike markets such as the hi-fi industry where every product plays the same type of tapes or CDs, and product differentiation is in terms of price, performance and functionality, until very recently computing systems have been proprietary: a product that was written for one system would not run on another one.

The incompatibility was both on the hardware and on the software level. However, the high cost of producing computing systems, and the need to make applications available on a wide variety of machines has led the industry to define standards, in spite of the relative costs involved in terms of performance.

Multimedia, as one of the most recent applications of computing (and also one of the most expensive and demanding) is one of the areas that has suffered from this problem. It is commonly accepted that unless standards for multimedia interchange are defined and widely adopted, the industry will not mature.

The industry has recently seen the development of media compression standards such as JPEG and MPEG. The general acceptance of such standards has made it feasible to have some of these algorithms implemented in hardware at an affordable cost for

desktop computers users, reducing the storage and communication costs for multimedia data. The performance now available at a desktop computer is also leading to the development of sophisticated hypermedia applications that are available on a *standard* home computer.

This chapter presents and discusses some of the problems related to *Multimedia Interchange* and to the standards under development.

2.2 Content and Presentation Information

Most data that we receive carries information at two levels:

- *Contents:* the abstract data being transmitted and,
- *Presentation:* the form in which the information is presented and perceived.

The same *contents* may be presented in a completely different format depending on the context. For example, a speech may be *written* for an audience with hearing problems and may be *spoken* to a different audience. In each case, although the contents is the same, the presentation information is very different. For portability, it is desirable that contents and presentation information are kept in different structures. When format and contents information are kept separately, it is easy to change the way information is presented as we will discuss in the following sections.

Unfortunately computers, like humans, do not usually share the same language. One of the most widely accepted standards is the one defined by ISO Standard 646 which defines the international reference version of the ASCII character set. ASCII is 7-bit coded and it is enough to represent English correctly. Other modern languages, including several European languages, require an 8-bit system. Oriental languages such as Chinese or Japanese require 16-bits or more. It is clear that a system such as ASCII, unable even to represent many languages, is not powerful enough to describe the broad spectrum that multimedia documents represent.

As ASCII is used by virtually all computer systems, it is a good means for exchanging information between computers. One method used to overcome the limitations imposed by the small character set is to add tags within the the textual information. These tags describe *how* the data contents is to be presented, as opposed to directly representing the presentation. A common way of using tags is by the definition of *markup languages* such as SGML.

2.3 Markup languages

```
{\Large 1 Markup languages}
\vspace{1mm}
\hspace{1cm}Here I have the first paragraph
            in this section.
\vspace{1mm}
\hspace{1cm}Finally here comes the second
            paragraph.
\vspace{2mm}
{\Large 2 Standards}
\vspace{1mm}
\vspace{1mm}
\hspace{1cm} Here we have the first paragraph
             of the second section.
```

Figure 2.1: Example of Procedural Markup

Markup is some information that is added to a document to describe its structure or formatting instructions. For example, the text in figure 2.1 contains instructions (using a LaTeX-*like* notation) on how it should be formatted. The markup describes the instructions to format two sections of a document in a very rigid way (*procedural markup*). If the user decides to alter the sections in a document, the section numbering would have to be changed manually; if he decides to change the relative size of font used for the section heading (defined as `\Large`), all sections would have to be edited, and the same applies to the spacing separating sections. In this case, the markup is providing formatting instructions but no structure information.

```
\section{Markup languages}
\label{s_markup}
Here I have the first paragraph in this section.

Finally here comes the second paragraph.

\section{Standards}
Here we have the first paragraph of the second
section with a reference to section \ref{s_markup}
(Markup languages).
```

Figure 2.2: Example of Logical Markup

A more flexible approach to describe the same text is given in figure 2.2, again in LaTeX format. The figure shows the contents and the structure of *sections* that could be part of a book, or an article, etc. In the figure, markup is defined by a keyword prefixed by \ and the marked contents is enclosed by {} (for example, the titles in the sections) or just placed between two markers like the contents of each section which is placed between the `\section{}` markup instructions. There are no tags to mark the beginning and ends of paragraphs; these are delimited by an empty line. The tag `\label{s_markup}` provides information that can be used by a referencing or indexing mechanism to retrieve the position of this piece of the overall structure.

How the sections will ultimately be formatted is not defined by the markup, which only defines the logical structures. Processing functions performed on the logical components that define the actual format of the document would be very different if the sections were part of a book or part of a two column article.

The concept of logical markup can be extented to more abstract structures beyond formatting ones, providing hooks for further processing. For example, a logical component could specify a *price* which may be used for some computation; or a reference such as the one in figure 2.2 or even to generate cross references within a text or a table of contents.

2.4 SGML

The *Standard Generalised Markup Language* (SGML) [ISO, 1986a, Goldfarb, 1990, McArthur, 1995] is the standard ISO 8879. It is based on two postulates:

1. "Markup should describe a document's structure and other attributes rather than specify processing to be performed on it, as descriptive markup need be done only once and will suffice for all future processing."
2. "Markup should be rigorous so that the techniques available for processing rigorously-defined objects like programs and data bases can be used for processing documents as well."

The standard is also *system* and *device independent* in the sense that it deals with virtual storage which can map onto different physical storage provided by different vendors; *language independent* as the standard can be used even in languages that do not use Latin based alphabets such as Greek; *application independent* as it is flexible enough to encode both simple and complex documents, able to deal with documents subject to frequent changes, and can also support the inclusion of data other than text.

It seems unlikely that a completely automatic publication process where human intervention is not required will happen, so another criteria that was taken into account when the standard was being developed was that the markup should be human readable.

SGML is like ODA (see section 2.6) in that it models documents as trees, but unlike ODA, SGML describes only the document's logical structure, not its layout semantics.

Logical items in SGML are called *elements*. Each element is delimited by tags that indicate its beginning and end. The element's contents (which may contain nested elements) occurs between the tags. SGML also allows user defined attributes to be added to elements.

SGML is not a language as the markup itself is not standardised but it is a metalanguage that provides the means to define the markup rules to be used. It defines syntax but no semantics.

2.4.1 A SGML document

A SGML document is made up of three parts:

1. *The SGML declaration;*
2. *The Document Type Definition (DTD);*
3. *The document instance.*

The SGML declaration

The SGML declaration includes information about the usage of characters sets, the concrete syntax, capacity requirements and optional features.

The Document Type Definition (DTD)

The Document Type Definition (DTD) is the centre of SGML. It defines the rules that apply to all documents of that type, including the names of the various elements (generic identifiers) allowed, any attributes they may have, the number of times they may appear, the order they must appear in, whether some markup (such as start- and end- tags) may be omitted and the relationships of the elements.

A DTD does not have information on how to process a document or what it should look like, but it allows users to define their own markup language depending on their requirements. Figure 2.3 show a *Document Type Declaration* (which may include several DTDs) where the items between [and] represent a DTD. The DTD defines a document of class `letter` which must consist of a recipient's name, a recipient's address, a senders's name, a salutation, a date and one or more paragraphs in that order.

```
<!DOCTYPE letter [
<!ELEMENT letter - - (rn, ra, sn, sl, d, p+)>
<!-- letter components:
     rn      = recipient's name
     ra      = recipient's address
     sn      = senders's name
     sl      = salutation
     d       = date
     p       = paragraph -->
<!ELEMENT (rn|ra|sn|sl|d|p) - o (#PCDATA)>
]>
```

Figure 2.3: Example of a Document Type Declaration for a type of document called letter

A DTD defines three types of markup commands:

- *Elements:* which are marked up with symbols called the *start-tag* and *end-tag*. For example, in figure 2.4 `<rn> </rn>` is an element (in the simplest form possible) whose start tag is "`<rn>`", end-tag is "`</rn>`" and with the *generic identifier* (or tag name) "`rn`";

- *Attributes:* which provide extra information about the element that is being specified. Attributes are similar to parameters in programming languages.

- *Entities:* which are character strings to mark locations in the text where external material, such as figures, or mathematical symbols that cannot be entered directly from the keyboard, must be placed.

The document instance

The document instance represents the document itself which is marked up according to the rules established in the SGML declaration and the DTD. Figure 2.4 gives an example of an instance of a SGML document.

```
<!DOCTYPE LETTER SYSTEM "letter.dtd">
<letter>
<rn>The Editor - Summer Magazine</rn>
<ra>London - UK</ra>
<sn>E Albuquerque </sn>
<sl>Dear Sir,</sl>
<d> 10/11/95</d>
<p>Please cancel my subscription to your
   magazine.</p>
</letter>
```

Figure 2.4: Example of an instance of a document of type letter (defined in figure 2.3)

2.4.2 Interchanging SGML documents

SGML documents are interchanged using the SGML Document Interchange Format (SDIF) (ISO 9069 [ISO, 1988]). SDIF was originally defined to provide the hooks to enable an SGML document to be interchanged in the OSI environment.

A SGML document may be made up of several separate parts such as document type definition, external entity definitions, etc, and the standard does not specify how these parts are organised. SDIF defines how to pack the parts into a single data stream with descriptors indicating how the parts are related to each other. The data stream is encoded using ASN 1, allowing SGML to be compatible and used with network standards [van Herwijnen, 1994] defined in terms of the OSI Reference Model.

2.4.3 Binary SGML (SGML-B)

SGML tags clearly add a large overhead to the data being transmitted. SGML-B is an extension to the SGML standard to provide SGML binary encoding with bidirectional convertibility between the text and binary forms. SGML-B uses minimised markup in order to reduce storage.

2.4.4 Limitations of SGML

One of the main limitations of SGML is that it does not actually specify documents. It specifies DTDs, and incompatible DTDs defeat the purpose of universal document exchange. Another shortcoming is that DTDs do not indicate how to process non-text objects. When non-text objects are encountered, DTDs simply specify special markup tags called "escapes" (sequences that suppress markup recognition when code extension is in use) that cause the processing program to jump outside the SGML-defined process to an application that can cope with the non-text object. It does not standardise either how objects are tagged for transfer to these other applications or how these applications will interpret those objects once they receive them.

A partial remedy for this weakness is the Hypermedia/Time-based (HyTime) structuring language, described in the next section.

2.5 HyTime

The HyTime or Hypermedia Time-Based Structuring Language [ISO, 1992, Markey, 1991, Newcomb *et al.*, 1991, Fujitsu and TechnoTeacher, 1995] is the result of a project at the American National Standard Institute aimed at creating a *Standard Music Description Language* (SMDL). HyTime became an ISO standard in April 1992.

HyTime is an application of SGML (and is said to be its future by Goldfarb) that allows markup and DTDs to be used to describe the structure of multimedia documents. The types of contents a document may have include:

- Digital audio recording;
- Digital motion video and dynamic graphics;
- Text fields;
- Musical notation;

- Instrumental control;
- Control signals for external systems.

HyTime overcomes some of the limitations of SGML by providing a standard way to tag text or non-text objects so that they can be rendered as a complete document or processed as independent objects. HyTime does not specify how document objects are encoded or interpreted by computer programs. However, by using standardised linking, alignment, and addressing methods, it ensures that such objects are made available to programs in a standardised way.

In order not to limit its expressive power, HyTime is not itself an SGML DTD, but provides constructs and guidelines ("architectural forms") for making DTDs for describing Hypermedia documents.

HyTime specifies how concepts common to all hypermedia documents can be represented using SGML. These concepts include [Adie, 1994]:

- association of objects within documents with hyperlinks;
- placement and interrelation of objects in space and time;
- logical structure of the document;
- inclusion of non-textual data in the document.

An "object" in HyTime is part of a document, and it is not medium dependent: it may be video, audio, text, a program, graphics, etc.

HyTime consists of six modules (figure 2.5):

- *The Base Module:* provides the architectural form that makes up the document itself (and therefore is required by all others modules), including a lexical model describing element contents; facilities for identifying policies for coping with changes to a document, or traversing a link ("activity tracking"); and the ability to define "container entities" which can hold multiple data objects.

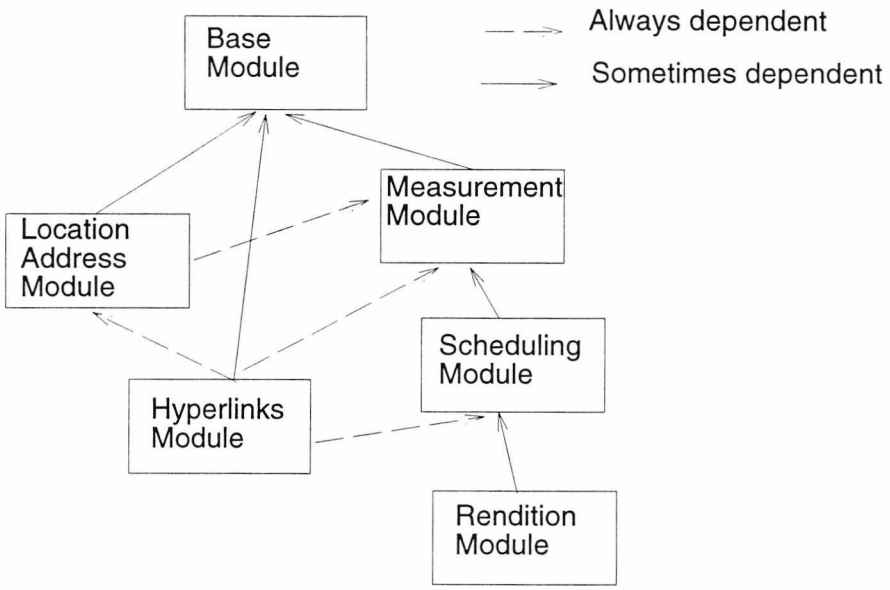

Figure 2.5: HyTime Module Interdependencies

- *The Measurement Module:* provides documents with the ability to represent concepts involving dimension, measurement and counting. It allows an object to be located in time and/or space, or any other domain, which can be represented by a finite coordinate space, within a bounding box called an "event", defined by a set of coordinate points which may be expressed in any units.

- *The Location Address Module:* provides means, in addition to those specified by SGML, to identify and refer to elements. This module provides a special "named location address" architectural form which can be used to refer indirectly to data which spans elements, or which is located in external entities. Data may also be addressed indirectly through the use of "queries", which return addresses of objects within some domain which have properties matching the query. If the measurement module is used, locations can be specified which are numeric addresses or indices along particular dimensions.

  HyTime provides three kinds of addressing:

  - Name-space Addressing (a unique name being provided);
  - Co-ordinate addressing (relative to a given location);

- Semantic addressing (by description).

- *The Hyperlinks Module:* supports the definition of links between parts of documents that can be traversed as a hypertext. The link endpoints may be location addresses, measurement, or scheduling modules.

  Two basic types of hyperlinks are defined: the *contextual link* (clink) which has two anchors, one of which is embedded in a document to explicitly denote the anchor location; and the *independent link* (ilink) which may have more than two anchors, and does not require the anchors to be embedded in the document, allowing structural information (*hyperlinks*) to be maintained outside the contents.

- *The Scheduling Module:* specifies how events in a source Finite Coordinate Space (FCS) are to be mapped onto a target FCS. For instance, events on a time axis could be projected onto a spatial axis for graphical display purposes, or a "virtual" time axis as used in music could be projected onto a physical time axis. In another example, if a map is to be rendered in say, a scale of 1:100000, it means that a dimension of 1 Km in the origin FCS will be mapped to 1 cm in the target FCS. If the event is applied to the whole map, it means that the same emphasis will be given to all its parts. If however it is desired to have some parts with more detail, more than one projector, with different scopes of action (`proscopes`) may be used. A schedule of `proscopes` is called a *baton* and determines the rate at which virtual units are converted into real units.

  An FCS may contain any number of event schedules and each event schedule may contain any number of events.

- *The Rendition Module:* allows for individual objects to be modified before rendition, in an object-specific way. It specifies how events in one finite coordinate space can be mapped to another finite coordinate space. One example is modification of colours in an image so that it can be displayed using the currently-selected

colour map on a graphics terminal, or changing the volume of an audio channel according to a user's requirements.

In this case, objects are modified within events and neither the semantics of objects nor modifiers are defined by HyTime. In a similar way to a *baton* described above, the scope of a modifier is expressed by a `modscope` and a schedule of `modscopes` is called a *wand*.

Applications are not required to support all modules: only the portions of HyTime that are appropriate for a given document have to be supported.

As it was defined as an exchange format, HyTime has great expressive power but is not optimised for run-time efficiency.

HyTime engine

The *HyTime engine* is basically a program that recognises HyTime constructs in a document. The HyTime standard does not specify how the *engine* works. There are still few engines available: there is at least one commercial system: *HyMinder* developed by TechnoTeacher [Adie, 1994]; and one developed by the Interactive Multimedia Group at the University of Massachussets (Lowel) [Koegel *et al.*, 1993].

2.5.1 Interchange in HyTime

As HyTime is an application of SGML, interchange is possible by using SDIF (section 2.4.2).

2.5.2 The Standard Music Description Language (SMDL)

The Standard Music Description Language (SMDL) [Newcomb, 1991] is an application of HyTime with the addition of tools for the representation of music. HyTime was originally developed from a project aimed at representing music.

SMDL defines four domains of information (figure 2.6):

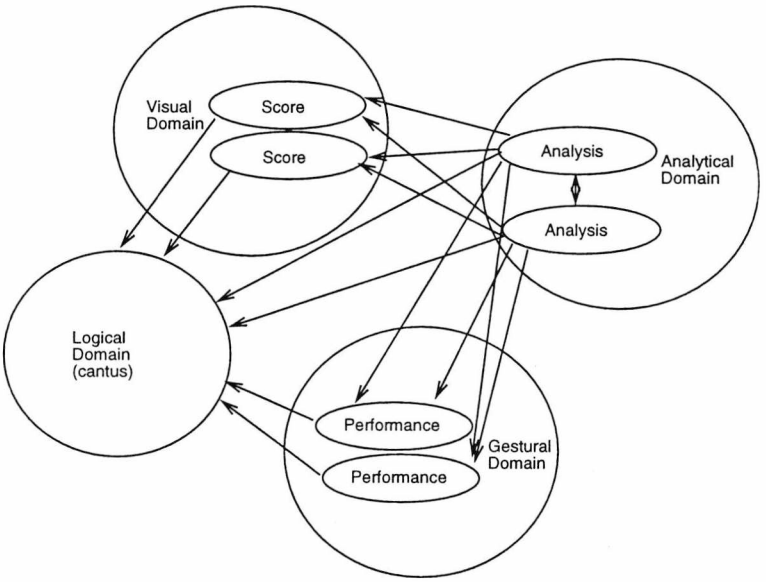

Figure 2.6: Domain of description in SMDL

- *Visual domain:* contains the score information.
- *Gestural domain;* contains the performance information.
- *Analytical domain:* contains theoretical information and facilitates theoretical and musicological discussions of musical structure, thematic transformation, performance and engraving practices and analytic techniques.
- *Logical domain (cantus):* Cantus can be defined as "pitches (frequency) and durations (timings)" and it contains the minimum information necessary for an automaton to generate a printed score and a minimal synthetic performance.

  A cantus can referred to by more than one performance (gestural domain) and more than one score. A document can also contain more than one printable edition of the music (visual domain).

  In SMDL, a piece of music is seen as a combination of events. The events occur in "thread" and "lyric" schedules (figure 2.7) in a finite time space, with a position (start) and extent (duration). Start and duration can be defined by an explicit position or by reference to other events.

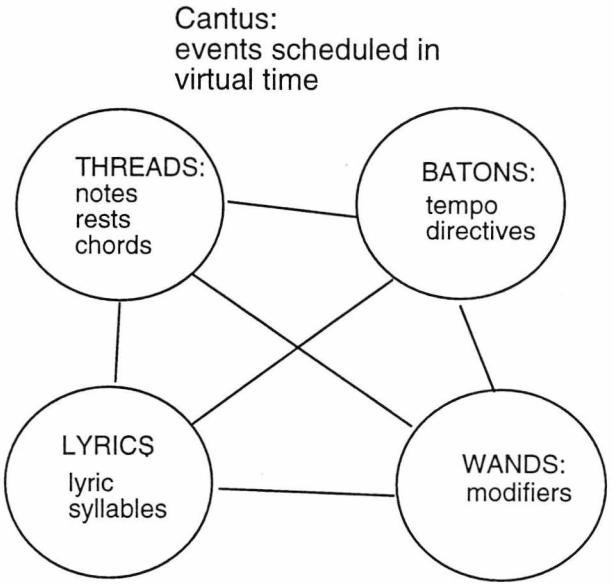

Figure 2.7: Cantus Structure in SMDL

To represent time units, which are not necessarily the same in music, SMDL uses a HyTime *baton*. Other processing instantiations such as modifying and filtering are specified by a HyTime *wand* in which all the modifier semantics are user defined.

2.5.3 Final remarks

HyTime provides very powerful constructs to define complex documents with time and synchronisation dependencies. Part of this strength is derived from the fact that it started as a project intended to represent music. One of the most interesting features of HyTime, which makes it specially suitable for representing hypermedia, is that it allows the definition of hyperlinks based on contents data in addition to the definition of time dependent constraints.

One weakness of HyTime, when used to represent hypermedia, comes directly from its strength. Because it has a very strong expressive power, and allows the definition of documents at a high level of abstraction, it requires a lot of processing and is thus not very efficient in terms of rendering that information.

2.6 ODA

ODA [ISO, 1989, Brown, 1989] is a standard for storing and interchanging documents. It defines a hierarchical and object-oriented document model. An ODA document can be seen as a tree in which the leaves contain the *contents* data and the *shape* defines the document structure. The separation of structure and document contents makes the handling and creation of multimedia documents easier. An object attribute is a property of a document or of a document component. It represents a characteristic of a document or a relation with one or more other documents.

An ODA document is described by two structures:

1. *Logical structure:* a document's logical structure is based on its meaning, i.e. a structure that makes sense to the writer and to the reader of the document. The logical structure defines the relationship between logical components and contents components. Examples of logical components are: chapters, sections, figures, etc. Logical components may also specify specialised items such as dictionary entries.

2. *Layout structure:* the layout structure defines the appearance of a document: how it is formatted. Examples of layout objects are pages, frames (rectangular areas) and blocks.

ODA also defines the concept of document classes which are analogous to DTDs in SGML. A document class is defined by generic structures. Based on the concept of document classes, the above structures can therefore be further divided into:

1. *Generic (Logical and Layout) structures:* which are associated with classes of documents, and

2. *Specific (Logical and Layout) structures:* which are associated with an example of the class.

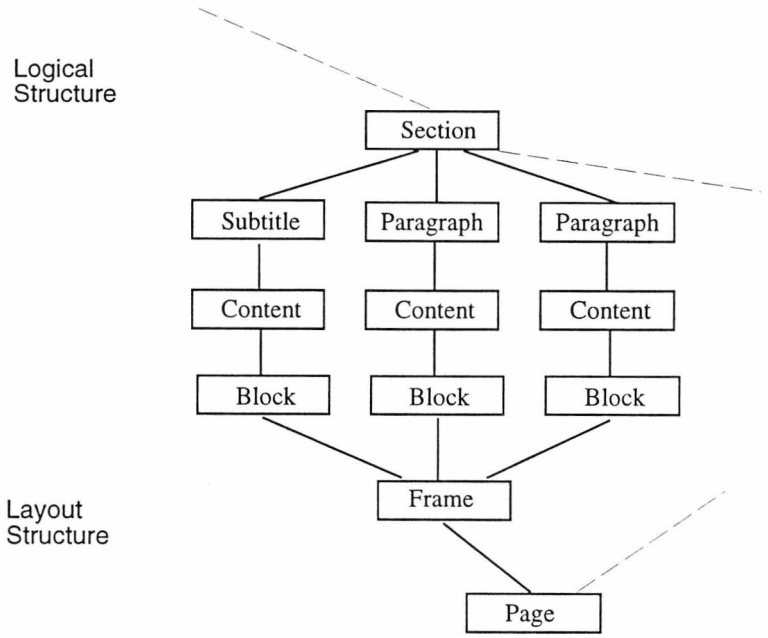

Figure 2.8: Logical and Layout structures in ODA

The separation of the logical and layout structure in ODA can be seen in figure 2.8.

The leaf objects contain an attribute that defines the type of the associated contents. Each type of contents is handled differently by rules called *content architectures.* Currently, ODA defines three content architectures:

- *Character content architecture:* made up of control and graphics characters. Positioning and format of characters are controlled by a set of attributes and characteristics. By default, characters are defined vertically, in lines from the top of the block and progressing from left to right and top down. It is possible to change attributes such as character and line progression direction, produce super and subscripts, etc.

- *Raster graphics content architecture: raster graphics* contents represents a two dimension image formed from a two dimension array of *image elements* (PEL, *Picture ELements*). Several attributes can be changed, including: progression direction of PELs within a line, line progression direction, starting point from which the PELs are placed within a line, image size, etc.

- *Geometric graphics content architecture:* This architecture is based on the CGM (*Computer Graphics Metafile*) standard. ODA attributes are used to provide initial CGM values, such as line style, width and colour, size of final image, etc.

Each one of the above architectures define one or more classes similar to document classes. These classes, in turn, may contain other classes offering different levels of facilities, for example, to change the direction in which characters are placed in a document.

2.6.1 Document Processing in ODA

In ODA, document processing is accomplished by three processes:

- *The Editing Process:* involves the process of creating and revising a document. This process includes the structural and logical editing of a document.

- *The Formatting Process:* defines the position where each item in a document will be placed. This process uses the logical and layout structures either generic or specific.

- *The Rendering Process:* is concerned with presenting the document in a suitable format to the user. This process *displays* a visual version of the document on paper or on a screen, for example.

Although there may be some overlap in the two last processes, the *Formatting process* is concerned with *placing items* while the *Rendering process* is concerned with providing a visible presentation of the items.

2.6.2 ODA document interchange

ODA documents an be exchanged in several formats:

- *ODIF format:* ODIF is an abstract syntax where components and attributes of a document are represented by a hierarchy of data structures that appear in a certain order.

- *Office Document Language (ODL) format:* ODL is a particular use of SGML to represent ODA documents.

2.6.3 Extensions to ODA

Several extensions to ODA have been proposed to incorporate multimedia. The extensions include:

- New content architecture features which:
  - Allow voice messages and other forms of audio content to be attached to ODA documents;
  - Allow CCITT H.261, JPEG, MPEG-1 and MPEG-2 objects to be incorporated into ODA documents;
  - Add attributes to ODA documents to allow the control of how images are rendered such as control of contrast, brightness, saturation and hue;
  - Allow the definition of markers within a video clip;
  - Allow video sequences to be cropped both spatially and temporally to show only part of a clip, for example.

- HyperODA: The *Hypermedia Extensions to the Open Document Architecture* are aimed at incorporating audiovisual information into ODA documents. The main characteristics of HyperODA are that it:
  - Supports references to data held externally to the document;
  - Supports non-linear structures, using contextual and independent hyperlinks based on the HyTime model;

- Supports temporal relationships between document components (e.g., sequential, parallel, cyclic, duration, start delay);

HyperODA has not yet become an International Standard.

2.7 Dexter

The *Dexter Hypertext Reference Model* [Halasz and Schwartz, 1990, Halasz and Schwartz, 1994] is an attempt to unify the abstractions met in the various hypertext systems and it is also a basis for the development of standards for interchange among hypertext systems.

The model is the result of two workshops on hypertext. The first one happened at the *Dexter Inn*, New Hampshire (USA), which is where its name comes from.

Dexter divides a hypertext system into three layers (figure 2.9):

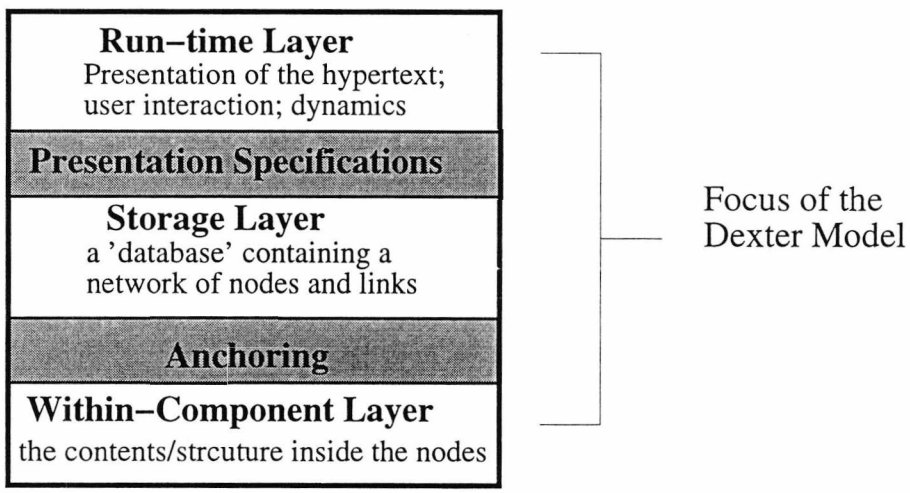

Figure 2.9: Dexter Model

- *Run-time layer:* this layer provides functionality for the use, visualisation and manipulation of the hypertext web. Dexter defines only a basic model and does not cover user interaction. This is because tools to implement these facilities are too varied.

The main function of the run-time layer is the presentation (instantiation) of a component to the user. When the component is instantiated, a copy is created and all user operations are performed on this copy, which is later stored by the storage layer.

It is possible to have more than one instance of the same component and to keep track of these instances; the run time layer maintains a *session* that maintains a mapping between instances and components.

- *Storage layer:* this layer is where the model is focused. The storage layer describes a data base made up of a hierarchy of components interconnected by referential links. Components are regarded as generic data storage with no distinction related to the medium used by the component.

  Access to the components is performed by two functions: one is a *resolver* responsible for resolving a component specification to give the *Unique IDentifier* (UID) which is associated with each component; and one is to *access* the component based on its UID.

- *Within-component layer:* components correspond to hypertext nodes. This layer deals with the contents and internal structure of components within a hypertext web. As a component can contain an unlimited range of contents and structure, the model does not cover this part. Models for specific applications such as ODA and SGML must be used to complete the definition of a hypertext as a whole.

The interface between the within-component layer and the storage layer is called *anchoring*. It provides the means to address items within a component.

The interface between the run-time and the storage layers is called the *presentation specification* and it is a mechanism that provides for the storage of information on how a component should be presented to the user.

2.7.1 Problems with the model

Dexter was one of the first attempts to standardise hypertext systems and although it is relatively recent, it has not given enough attention to the additional problems *hypermedia* adds to *hypertext*:

- Dexter has no notion of time which makes it unsuitable for expressing hypermedia in general;
- It does not allow *across hypertext* links. It does not distinguish between contents that are managed within the scope of the system and those managed by third party applications;
- Composites are also limited as discussed in [Leggett and Schnase, 1994], in particular with respect to composition, which is by copy rather than by reference, preventing the reuse of content objects.

2.8 CWI Multimedia Interchange Format

The *CWI Multimedia Interchange Format* (CMIF) [Bulterman *et al.*, 1991] is used to describe the temporal and structural relationships existing in multimedia documents.

The CMIF project had two goals:

1. To define a structure that separates the temporal, spatial, and content-based aspects of multimedia documents;
2. To investigate ways of using the document description rather than the data contents to control the interrogation and synchronisation of one or more document sets.

The project addresses three problems existing in multimedia applications:

- Elements manipulated within multimedia systems consist of raw data rather than structured information, with little inherent meaning;

- The representation and manipulation of such data is highly machine and/or device dependent;

- The synchronisation within multimedia applications is often implicitly encoded as a function of the speed of a particular system and interface, limiting the ways interaction among elements can be expressed and implemented.

A CMIF document is a collection of several media and a set of structure and synchronisation relationships describing how to present and manipulate these components (figure 2.10).

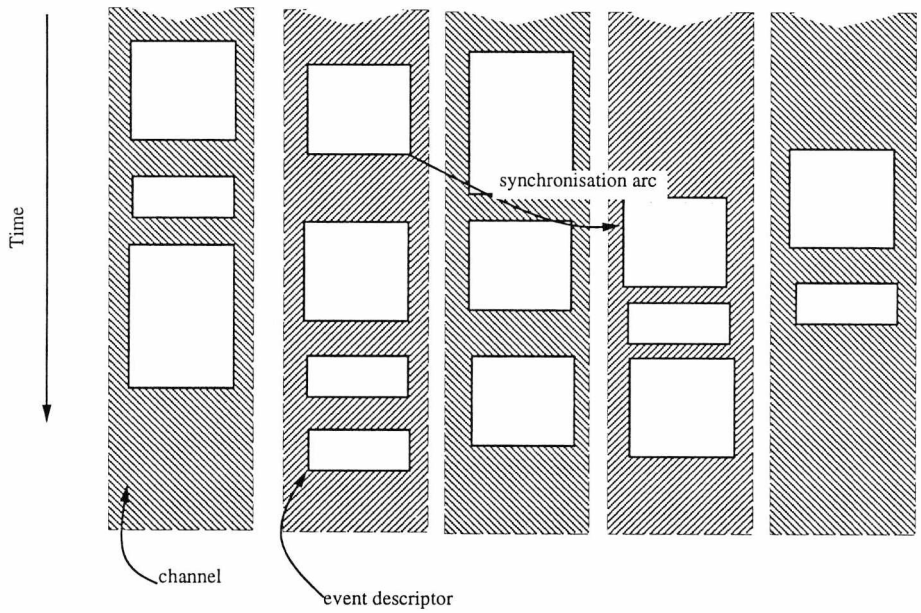

Figure 2.10: Document Structure Components in CMIF (from [Bulterman *et al.*, 1991])

CMIF provides a powerful infrastructure to build hypermedia applications, in particular when expressing timing and synchronisation relationships. However, the model does not provide a direct way to represent "hypermedia" as it lacks a general mechanism to represent hyperlinks.

2.9 The Amsterdam Hypermedia Model (AHM)

The Amsterdam Hypermedia Model (AHM) [Hardman *et al.*, 1993, Hardman *et al.*, 1994] is a framework that can be used to describe hypermedia systems. The AHM extends the Dexter hypertext model (see section 2.7) by adding to it notions of time as defined by CMIF (described in the previous section).

AHM groups temporal relationships among data items in two groups:

1. *Collection:* the class of items related to the identification of components that are to be presented together;

2. *Synchronisation:* the class of items that specify the relative order in which the components are to be presented.

This contrasts with the Dexter model which provides support for collection via the hierarchical definition of components. The set of atomic components can be modeled by Dexter's composite but it does not provide for specifying relative timing relationships among components.

AHM separates *contents data* and presentation information by defining atomic and composite components as shown in figure 2.11.

The atomic component contains information related to a specific data block including duration, presentation specification and anchor identification while a composite contains presentation information related to a collection of atomic or composite blocks. A composite contains no contents data; such data must be either included or referenced in an atomic component.

There are two types of composites:

1. *Choice composite:* at most one child is displayed;

2. *Parallel composite:* all components are displayed.

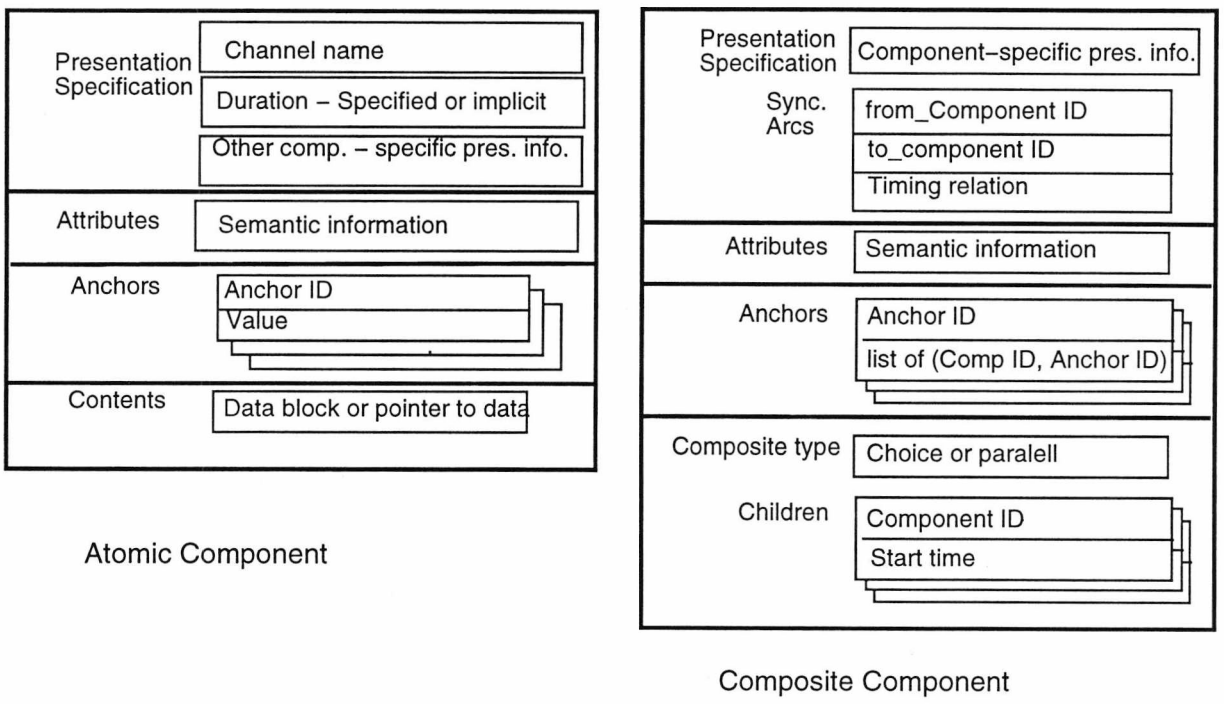

Figure 2.11: Amsterdam Hypermedia Model

2.9.1 Synchronisation in AHM

AHM supports two levels of synchronisation as shown in figure 2.12:

- *Coarse-grained synchronisation:* defines the constraints between the children in a composite such as relative starting time of each child in a composite and this information must be given explicitly with the child definition;
- *Fine-grained synchronisation:* defines constraints among children (which can be nested) within a composite component and these constraints are specified using synchronisation arcs.

2.9.2 Link context

AHM defines the concept of a link context to specify how components should behave when a link is followed. A link context is a composite that contains a collection of components affected by the activation of a link. A source context for a link is the part of

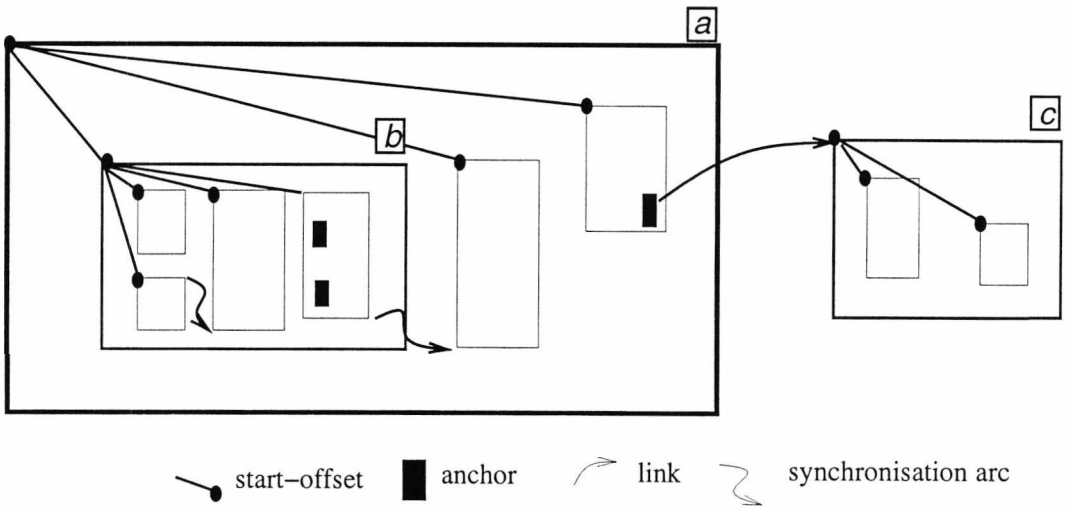

Figure 2.12: Timing relations in AHM

the presentation affected by initiating a link and a destination context is the part affected at the link arrival.

2.9.3 Channels

Channels provide for specifying global output attributes of documents. A channel can specify, for example, the font and style for a text channel.

The concept of channel can also be used to specify, for example, the language to be used to output speech or text in a multi-lingual document.

2.9.4 Limitations of AHM

The main current limitations of AHM are derived from the fact that the underlying model (CMIF) does not support composite "anchors" therefore it is not possible to define complex link conditions.

2.10 QuickTime

The *QuickTime Movie File* (QMF) format [Apple, 1993], was developed by Apple Computers originally for Apple Macintoshes, as an extension to System 7; it has now been ported to MS-Windows.

QuickTime is a container for time-based data. It presents a model for the storage and interchange of time-related media that is independent of a system's built-in timing and synchronisation capabilities.

A QuickTime *movie* is actually any dynamic data such as a *movie*, a slide show, an animation, etc. A movie can have several tracks with different types of information but it can only have one track with a specific medium (such as audio). A movie atom is made up of track atoms which are made up of media atoms as described in figure 2.13, from [Buford, 1994].

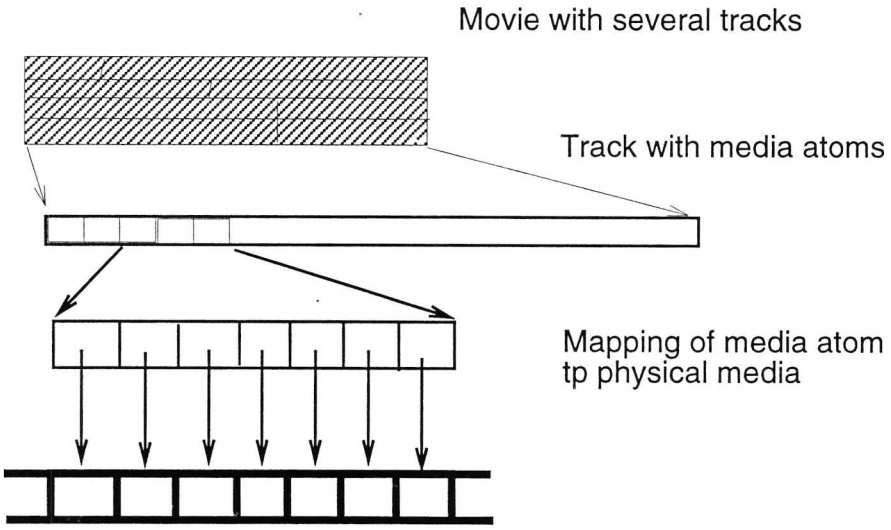

Figure 2.13: QuickTime components

A movie also includes the specification for a *poster* which is a single image to represent a movie, such as an icon, and a *preview* which is a short segment of a QuickTime movie also used to identify a movie.

Components of QuickTime

A full version of QuickTime includes these components:

- *Movie toolbox* which is the authoring application for creating, editing and displaying movies;
- *Image compression manager* that controls compression and decompression of images in a movie. QuickTime uses four compression algorithms:
  1. *Apple Photo - JPEG* for static images;
  2. *Apple Video and Apple Compact Video* for video;
  3. *Apple graphics* for graphics;
  4. *Apple Animations* for animations.
- *Component manager* that provides an interface for adding compression/decompression methods and device drivers.
- *"Scrapbook"* which is capable of storing movie clips to be pasted into applications which use QuickTime.

Limitations of QuickTime

Although QuickTime provides for adding time based media to existing applications, it does not provide mechanisms for creating *hypermedia* as it does not provide for links being triggered from the media it handles.

2.11 Adobe Acrobat

Although Adobe Acrobat [Adobe, 1995] is not a standard for document interchange, it was invented by the creators of the PDL (page-description language) PostScript which is the *de facto* standard of the printer world. Adobe aims at achieving the same level

of acceptance for electronic documents. Acrobat accomplishes this by using proven technologies such as EPS (Encapsulated PostScript) and Multiple Masters fonts.

Acrobat's PDF (Portable Document Format) uses a page description language (PDL) based on PostScript to describe the text, graphics, and images in a file with additional facilities for links and annotations. Because it is based on PostScript, a PDF file is device and resolution-independent, so it will reproduce at the highest resolution that the output device supports.

Adobe has published PDF as an open standard, allowing developers to support the format in third-party applications.

Adobe Acrobat comprises a set of packages with specific objectives:

- *Acrobat Exchange* is the package to read and write portable documents;
- *Acrobat Reader* is only capable of reading documents created using the *Exchange*;
- *Acrobat Distiller* also used to create documents. It provides an easier way of creating documents than the *Exchange* as it can take any level 2 PostScript encapsulated PostScript created by another package and convert it to PDF format.

Acrobat provides for adding links, notes and book marks to documents although it does not provide facilities to identify who added the marks.

Limitations of Acrobat

Although very powerful when converting existing documents in PostScript format, Acrobat is not a general hypermedia tool, since it was not designed to take advantage of distribution, and is not directly extensible to accommodate new media. Another important limitation of Acrobat is that it is page based.

It seems that the natural environment for its usage will be the office, where it provides a good solution to reduce the volume of paper, and not as a hypermedia tool in general, because of its current lack of extensibility.

2.12 The World Wide Web

The World Wide Web (WWW or W3 or just "the web") is a wide-area client-server architecture for retrieving hypermedia documents over the Internet. It started as a project at CERN as a large scale distributed multimedia system to provide for the distribution of documents related to high energy physics research. The web has since spread to other areas and is probably the most well known and heavily used distributed multimedia system available now, with browsers available for virtually all operating systems and configurations from dumb terminals to high performance graphics systems.

The user sees the web as a collections of nodes (documents) and links between them. Navigation is usually initiated by clicking with a mouse on an anchor that triggers the associated link and causes the destination document to be retrieved. It also supports a means of searching remote information sources, for example bibliographies, phone directories and instruction manuals.

The web provide transparent interfaces to Gopher, Wais, or anonymous ftp, providing access to virtually all resources present in the Internet. Identification of resources in the web is accomplished by *Uniform Resource Locators* which are described in the next section.

2.12.1 Uniform Resource Locators (URL)

The World Wide Web uses a naming scheme called *Uniform Resource Locators* (URL) [Berners-Lee, 1995] to represent hypermedia links and links to shared resources.

The URL syntax identifies documents in terms of the protocol to retrieve them, their Internet host and path name (figure 2.14). Among the protocols supported are http, telnet, (anonymous) ftp, NNTP, wais and gopher. One drawback of URLs is that they generally depend on particular servers. Work is still in progress to provide widespread support for lifetime identifiers that are location independent. This will make it possible to provide automated directory services similar to X.500 [CCITT, 1988] for locating

the nearest copy of a resource [Raggett, 1994a].

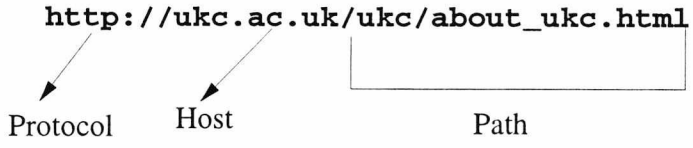

Figure 2.14: Example of a URL

Universal Resource Numbers (URN)

Universal Resource Numbers (URN) are a proposed system for unique timeless identifiers of network-accessible files being developed by IETF Working Groups. URNs, unlike URLs, do not contain information to retrieve nodes, and may be allocated to nodes and represented in source anchors.

The objective of URN is to reduce network traffic and provide a more robust structure where the location of a host is not encoded. The reader's system does not need to retrieve a node if it already has it. Implementation of caching mechanisms is facilitated; because the identification does not change, any server with the object identified by the URN can satisfy a request for it. On the other hand, if the node is changed, all links pointing to it must be updated invalidating existing caches. This scheme is therefore useful only for very large nodes, that impose a heavy transmission cost for their retrieval, and that are unlikely to be updated.

2.12.2 Hypertext Markup Language (HTML)

The *Hypertext Markup Language* (HTML) [Berners-Lee, 1993], which is the language used by the Web browsers, describes the organisation of documents so that structural elements can be identified and accessed over the Internet.

An HTML document is an ASCII file marked up with tags, with a syntax based on SGML (see example in figure 2.15), that provide a hierarchical structure to the text [Barry, 1994]. HTML includes markup elements for:

```
<TITLE> Here comes the documents' title </TITLE>
<H1>A heading with level 1 is here></H1>
<H2>A sub-heading is here</H2>
<P> And here comes the first paragraph.</P>
<H2>Another sub-heading is here</H2>
<P> Now we have the second paragraph...</P>
<P>This is a reference to the WWW
  <A HREF="http://www.w3.org/hypertext/WWW/TheProject">
  (World Wide Web)</A>
</P>
<P>This link will connect you to the Library
<A HREF="telnet://cats.ukc.ac.uk/">
   (UKC Library Catalogue: CATS)</A>
</P>
```

Figure 2.15: Example of an HTML document

- *Headers:* six levels of header are supported and they are tagged from `H1` (the most significant) to `H6` (the least significant). Usually the level of significance will define the font and size the header is displayed with;

- *Paragraphs:* normal texts automatically wrapped by the browser, and a paragraph has in most cases one tag to define its beginning and one to mark its end. The ending tag can be implicit;

- *Various types of character highlighting;*

- *Character-like in-line images;*

- *Hypertext links:* an HTML link defined by an URL (see section 2.12.1). In the example above, the reference `<A HREF="http://www.w3.org/hypertext/WWW/TheProject"> (World Wide Web)</A>` will connect the user to the home page at `www.w3.org` using the `http` protocol. The only part of the reference that the browsers make visible to the reader is `(World Wide Web)` which can be selected (e.g. with a mouse) to activate the link.

- *Lists;*

- *Preformatted text;*
- *Simple search facility;*

HTML+

HTML+ [Raggett, 1993b, Raggett, 1993a, Raggett, 1994a] is a superset of HTML adding extra features such as figures, tables and forms. It also generalises structures present in HTML to facilitate the process of converting between HTML+ and other formats. HTML+ formalises the concept of nested lists providing various list styles. It also defines *unordered lists* that can be used to implement menu elements.

In addition to the elements supported by HTML, an HTML+ document supports the following elements:

- *Nested lists;*
- *Figures;*
- *Tables;*
- *Forms;*
- *Literal or Preformatted text;*
- *Mathematical formulae;*

2.12.3 Virtual reality and the Web

Extensions to provide virtual reality via the Web are being investigated. Examples include the *Virtual Reality Markup Language* (VRML) discussed in [Raggett, 1994b] and [Pichler *et al.*, 1995]. The virtual reality extensions would also provide support for a *virtual teleconference*. The first web browsers with virtual reality features, such as VRweb [Pichler *et al.*, 1995], were made available in 1995.

2.13 Presentation Environment for Multimedia Objects (PREMO)

The *Presentation Environment for Multimedia Objects* (PREMO) is being developed by ISO/IEC JTC1/SC24 [ISO, 1994b, ISO, 1994a, Stenzel *et al.*, 1994] which is the group responsible for computer graphics and image processing. The main aim of the project is to add presentation and interaction with more than one medium. PREMO should therefore make use of standards for single-media already developed within ISO. As PREMO deals primarily with presentation aspects of multimedia, it is distinguished from and should be used in conjunction with, other ISO/IEC standards such as ODA (described in section 2.6), HyTime (section 2.5) and MHEG (chapter 3) [Herman *et al.*, 1994, Stenzel *et al.*, 1994].

PREMO is being designed to cope with the requirements of new multimedia applications and of new functionalities added to workstations. It is an open architecture that provides facilities for customisation, extensions and configurations depending on the needs of applications, and should satisfy the needs of various areas of application, ranging from CAD/CAM to virtual reality.

PREMO is object oriented and based on the model proposed by the Object Management Group (OMG) [Digital Equipment *et al.*, 1993]. This should make it portable and usable in a distributed heterogeneous environment.

The work on PREMO is still in its early stages and it is not expected that PREMO will become an international standard before 1997.

2.14 Final remarks

This chapter has given an overview of a number of existing and emerging standards for structuring hypermedia applications. Of these, the most important are SGML, HyTime, ODA, PREMO and Acrobat. Except for the last one, they are all "de jure" standards;

Acrobat is a commercial product which is being proposed as a "de facto" standard.

Interest in SGML and HyTime applications seems to be booming, especially in the publishing industry (where SGML was originated), while there seems to be a decrease in interest in ODA.

PREMO is still not well known and it is not clear that a new standard that overlaps with MHEG, HyTime and other graphics standards is required in the short term.

Although Acrobat is still a relatively new product, the fact that it is based on well known technologies makes it a strong candidate to become a standard as Adobe intends.

Another important emerging "de jure" standard, not discussed in this chapter, is MHEG, which is described in detail in the next chapter.

Chapter 3

MHEG

The Multimedia and Hypermedia information coding Expert Group (MHEG) is the ISO Working Group WG12 of SC29. The standard being defined by the group is "the base coded representation of final form multimedia and hypermedia information objects" that will be interchanged as a whole within or across services and applications.

This chapter presents an overview of the MHEG standard as defined in [MHEG, 1994b]. An introduction to MHEG can be found in [Colaitis and Bertrand, 1994], [Meyer-Boudnik and Effelsberg, 1995] and [Casey, 1994]. A critical analysis of MHEG is presented in chapter 7 and an overview of MHEG classes is shown in Appendix A.

3.1 Introduction

The work being developed by MHEG, under the general title of *Information Technology – Coding of Multimedia and Hypermedia Information*, aims at producing the following documents:

- *13522-1 MHEG Object Representation, Base notation (ASN.1)*: this is the most advanced part of the work. The description of MHEG in this chapter is based on this document. In the text below, unless explicitly stated, the term MHEG will refer to this document;

- *13522-2 Alternate notation (SGML)*: this document will be developed further once the base notation work is concluded;
- *13522-3 MHEG extensions for Scripting Language Support*: this is still in its early stages of development, as initially MHEG did not aim at providing support for scripting languages;
- *13522-4 Registration Procedure for Format Identifiers*: this is also still in an early stage of development;
- *13522-5 MHEG Subset for Base Level Implementation* : which is a new work item added to the MHEG work in November 1994. MHEG 5 defines a subset of MHEG to be applied to "simple applications" such as video on demand and browsing systems. It is discussed in more detail in section 7.1.1

3.1.1 Standard Objectives

The standard focuses on the generic structuring aspects of the objects and takes into account the following requirements:

- Use in systems with minimal resources;
- Interactivity and multimedia synchronisation;
- Real time presentation;
- Real time interchange;
- Final form representation.

3.1.2 Suitability of MHEG

MHEG defines objects in a non revisable form which makes it unsuitable for highly interactive authoring applications. However, it is well suited for *reading* or *browsing*

systems. For example, an MHEG system is suitable for presenting a collection of multimedia objects stored on a CD-ROM.

3.2 Object Interchange

Figure 3.1 shows the scope of MHEG. The figure shows levels where multimedia interchange occur:

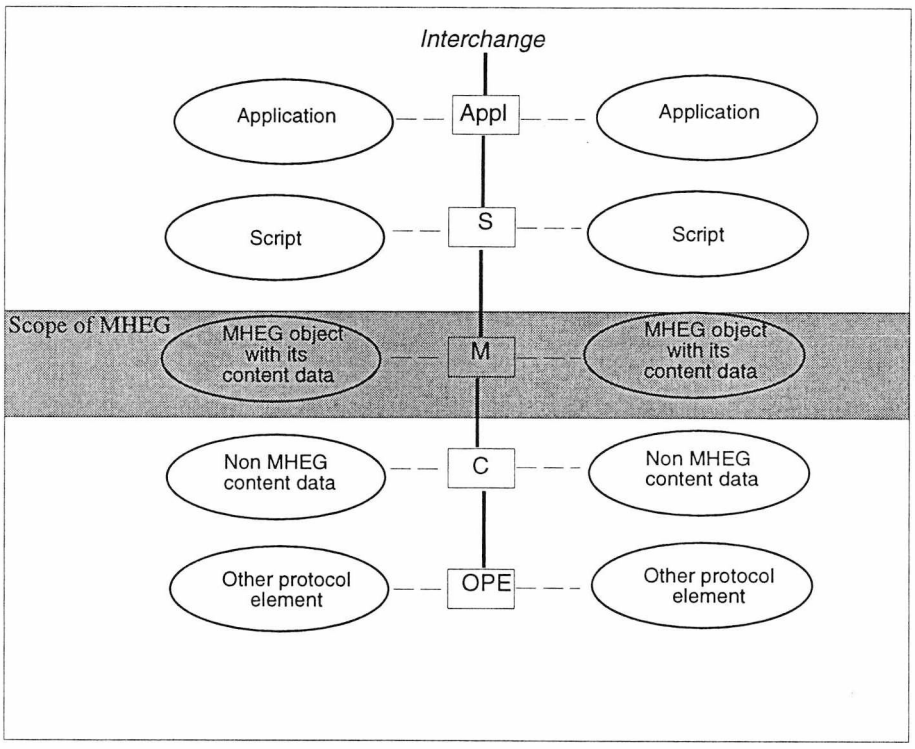

Figure 3.1: Scope of MHEG

1. *Application level:* the infinite varieties that can exist at the application level makes it unsuitable for standardisation. An application may, however, use the script level to exchange objects.

2. *Script level:* MHEG currently makes no attempt to standardise scripting languages. Scripting languages should use the MHEG level below to interchange objects.

3. *MHEG object level:* this is the scope of MHEG.

4. *Non-MHEG object level:* this the level where standards for monomedia interchange are used. MHEG objects make use of standards at this level.

5. *Other protocol element level:* this is the lowest level of interchange. Protocols for messages and acknowledgements are included in this level.

6. *Object Representation:* Objects are coded using ASN.1 and an alternative representation in SGML is being developed.

3.3 Structure of MHEG

MHEG is based on three concepts:

1. *MHEG classes, MHEG objects:* MHEG classes represent the objects that are actually interchanged. The run time system may instantiate any number of objects from a given object class;

2. *Run-time objects (rt-objects):* Run-time objects are not interchanged between applications, but their existence is triggered dynamically by the run-time system. Objects created from subclasses of Model (figure 3.2) may be reused in different contexts. Each time such an object is created, a *run-time object* is instantiated. For example, an image may be exchanged once but be used many times with different attributes such as size and colour. Elements in a run-time composite are called *sockets*.

3. *Channels:* channels define logical spaces in which run-time objects are presented

The standard defines an object oriented representation of multimedia entities, with a single inheritance tree as shown in figure 3.2.

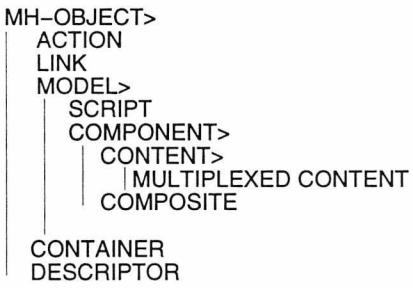

Figure 3.2: MHEG Classes

The hierarchy defines data inheritance but the standard does not define class methods. Neither does MHEG enforce or define an object-oriented approach for a MHEG system; it makes no assumption on the internal representation of systems.

3.4 Object Identification

3.4.1 Naming

The standard provides three identification mechanisms:

1. External identification;
2. Internal identification;
3. Symbolic identification.

External identification

External identifiers are defined by the standards ISO 8879 *Formal Public Identifiers* [ISO, 1986a] and *ISO 9070* [ISO, 1986b] *Registration procedures for public text owner identifiers*. They are not defined by the standard and are not encoded within the MHEG object.

An external identification can be decoded without decoding the object. It is made up of two parts: *i)* a public identifier which is a character string defined by ISO 8879;

and *ii)* a system identifier which is system dependent and is used to identify information within a system and to identify the system itself.

The external identification is the only way to identify data not included in a content object or in a script object.

Internal identification

Internal identifiers are encoded within the object. They cannot be retrieved without decoding the object. An internal identification can be an *identifier* (used to identify an MHEG Object, an rt-object, a channel, a multiplexed stream or null objects), or *index* (used to identify a composition element, a container element or a socket).

- *MHEG identifier:* an optional identifier that can be assigned to each MHEG object. It is made up of an *Application Identifier* which is a list of numerics provided by the application designer and an *Object Number.*

  It is the object designer's responsibility to ensure that MHEG identifiers are unique within the application.

- *Rt-object identifier:* a mandatory identifier assigned to each rt-object. The identifier is composed of a *model object identification* that identifies the model object, and an *rt-object number* provided by the author (again, the author must ensure that the number is unique).

- *Channel identifier:* a mandatory numeric identifier assigned to a channel. The author must ensure that the number is unique within the composite.

- *Stream identifier:* a mandatory identifier assigned to each stream within a multiplexed content object. A stream identifier defines a list of numerics that give the path from an outer stream to inner stream within the multiplexed data.

- *Indexing:* An index provides an identification for an element within a constructed entity. In container objects an index can be created by the engine (they are

sequential) or by the author, in which case they do not need to be sequential. In composite objects, indexes are provided by the author and do not need to be sequential. In rt-composite objects, sockets are indexed from one, sequentially and all indexes must be used.

Symbolic identification

Symbolic identification (*aliasing*) can be used to replace any other external or internal identification. It is recommended that such an alias should conform to sub-clause 9.3 of ISO 8879 SGML.

3.4.2 Referencing

Referencing in MHEG is realised by the use of a *Generic Reference* which addresses MHEG entities. Generic references can be constants or the result of a get action (see section A.2).

Table 3.1 summarises the referencing that can be used in each context. Chapter 5 gives details of MHEG referencing.

Reference Type	Alias	Exter id.	Cont id.	MHEG id.	Null	Obj + tail	Model + rt-obj	rt-comp + socket tail
Data	•	•						
MHEG Object	•	•	•	•	•			
Container element	•					•		
Rt-Object	•						•	•
Socket	•							
Channel	•			•	•			
Stream	•			•				

Table 3.1: MHEG Referencing Summary

3.4.3 Tail referencing

Tail referencing is used to identify an element within a container object or rt-composite. It can be:

- *Single tail:* which describes the path from an outer entity to the desired inner one;
- *Child tail:* where the reference is made to the set of all child elements in the first generation only;
- *Descendant tail:* where the reference is made to all descendants elements in all generations.

3.5 Representation of time and space

The MHEG generic space

MHEG defines a generic space as being composed of four axes:

- *The temporal axis (T):* this axis is measured using a *Generic Temporal Unit* (GTU). For rt-components, the range is `[0,original_duration]` where `original_duration` is, for example, the length of a video clip.

  The granularity of T is defined by a *Generic Temporal Ratio* (GTR) which defines the number of consecutive unitary *Generic Temporal Units* intervals there are in one second.

- *Three spatial axes (X, Y, Z):* which have the usual right handed mathematical sense. Measures in these axes are expressed by *Generic Spatial Units* (GSU)

  The granularity of each axis is defined by a *Generic Space Ratio* (GSR) which defines the number of unitary *Generic Spatial Units* intervals there are in one generic space.

Composition and projection in the generic space

The standard provides two independent mechanisms for the manipulation of time and space:

1. The composition of MHEG objects that defines position points at which a presentation process will attach the reference points of the projectable objects; and

2. The projection of an MHEG object that specifies how content data is related to its reference point.

The user process is responsible for converting MHEG generic space to real units. The rate at which time units are mapped into real units can be changed by setting different values of *speed*. Size can be changed by setting different *aspect ratios* (they can be different for each axis).

MHEG does not have a generic mechanism such as HyTime's wands that allows the user to define the semantics of the mapping from one generic space to another generic space or to real units.

3.5.1 Synchronisations relations

Six levels of synchronisations are provided:

1. *Atomic Serial Synchronisation Relation* (figure 3.3 left)

   Object `01` is presented immediately after the activation time of a composite object and object `02` after the end of presentation of object `01`.

2. *Atomic Parallel Synchronisation Relation* (figure 3.3 right)

   Objects `01` and `02` are presented both starting at the reference time.

3. *Elementary Synchronisation Relation*

   Two types are defined:

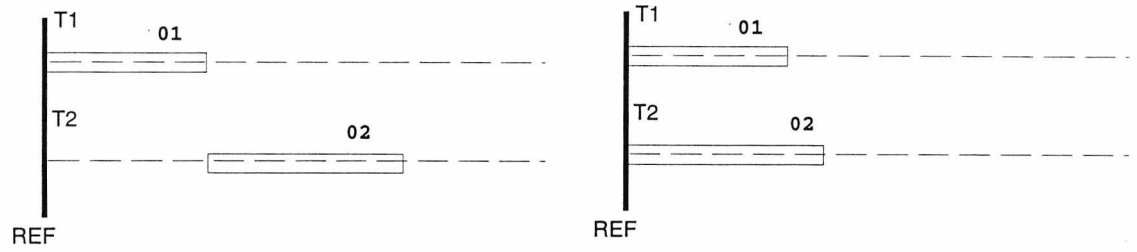

Figure 3.3: Atomic Serial (l) and Parallel (r) Synchronisation

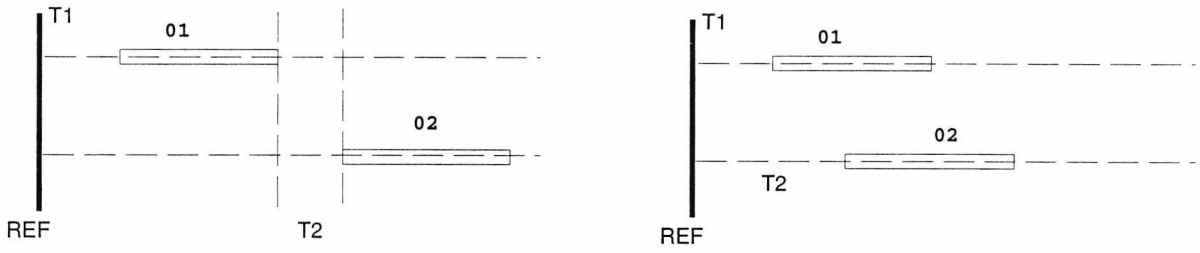

Figure 3.4: Sequential (l) and Parallel (r)Mode Synchronisation

(a) Sequential mode (figure 3.4 left) where object `01` is presented at time `T1` after the reference presentation time and object `02` is presented at time `T2` after the end of `01`;

(b) Parallel mode (figure 3.4 right) where object `01` is presented at time `T1` after the reference presentation time and object `02` is presented at time `T2` after the reference presentation time.

4. *Conditional Synchronisation Relation*

   The presentation of an object is linked to the satisfaction of a condition;

5. *Chained Synchronisation Relation* (figure 3.5)

   The synchronisation occurs at marks.

6. *Cyclic Synchronisation Relation* (figure 3.6)

   This is the type of synchronisation required by a chronometer for example.

The types of synchronisation provided are powerful enough to model most hypermedia usages. Synchronisations that depend on the occurrence of complex conditions

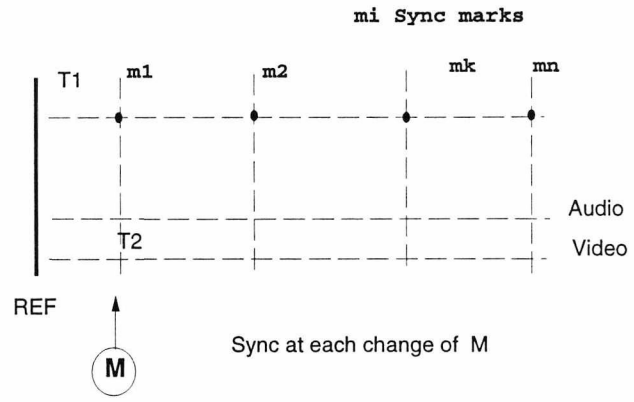

Figure 3.5: Chained Synchronisation

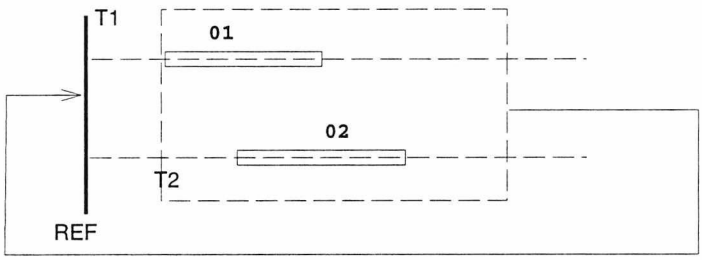

Figure 3.6: Cyclic Synchronisation

cannot be defined directly and must be handled by scripting languages that will generate the actions defining synchronisation types supported by MHEG.

3.6 Extensibility of the model

MHEG allows applications to extend the model via three mechanisms:

1. *Extension of elementary actions:* Elementary actions may be extended by *i)* adding a parameter to an existing action; *ii)* extending actions by allowing the inclusion of more powerful functionalities; *iii)* using dedicated implementations that extend the elementary action — the actions are then tagged as 'PRIVATE' and do not conflict with elementary actions defined by the standard.

   Basic MHEG engines ignore the extended actions.

2. *Extension of attributes of an MHEG object:* applications may create new classes derived from MHEG classes (they are not considered MHEG classes), or new elementary actions may be defined as described above.

3. *Extension of data types and classifications:* MHEG proposes the existence of an "MHEG data type registration authority" that is responsible for keeping the "MHEG catalogue" to which new data types or classifications may be added.

   Applications may also define a "proprietary catalogue" that will keep private data types and classifications.

3.7 Final Remarks

This chapter provided a flavour of the MHEG standard. An analysis of its main features in provided in chapter 7. However, as a standard that deals mainly with the presentation of multimedia objects, one of the main points that distinguishes an MHEG implementation from other standards, such as HyTime, SGML or ODA, is the requirement for efficiency. When a choice must be made between expressive power and run-time efficiency, the latter must be emphasised.

Chapter 4

Requirements and Constraints

4.1 Introduction

Most hypermedia systems designed to date are centralised monolithic systems. These systems provide built-in support for a limited set of media. The environment usually has a poorer supporting interface to each medium than that of an application solely dedicated to that medium.

As new media are constantly introduced to the market, monolithic systems are no longer feasible. The introduction of new media, hardware and software tend to make these centralised systems obsolete soon after they are released.

Multimedia environments should be easily extensible to accommodate technological developments. The interoperation and interconnection of heterogeneous systems and systems components should be easily and coherently extendible by the addition of new components, defining an Open System model. The development of concepts independent of the applications is needed for the handling of information in an Open System environment.

Distributed processing of multimedia information will play a key role in applications areas [Eckhard Moeller and Angela Scheller and Gerd Schürmann, 1990] such as office systems, publishing, health care, CAD, CAI, multimedia information bases, and

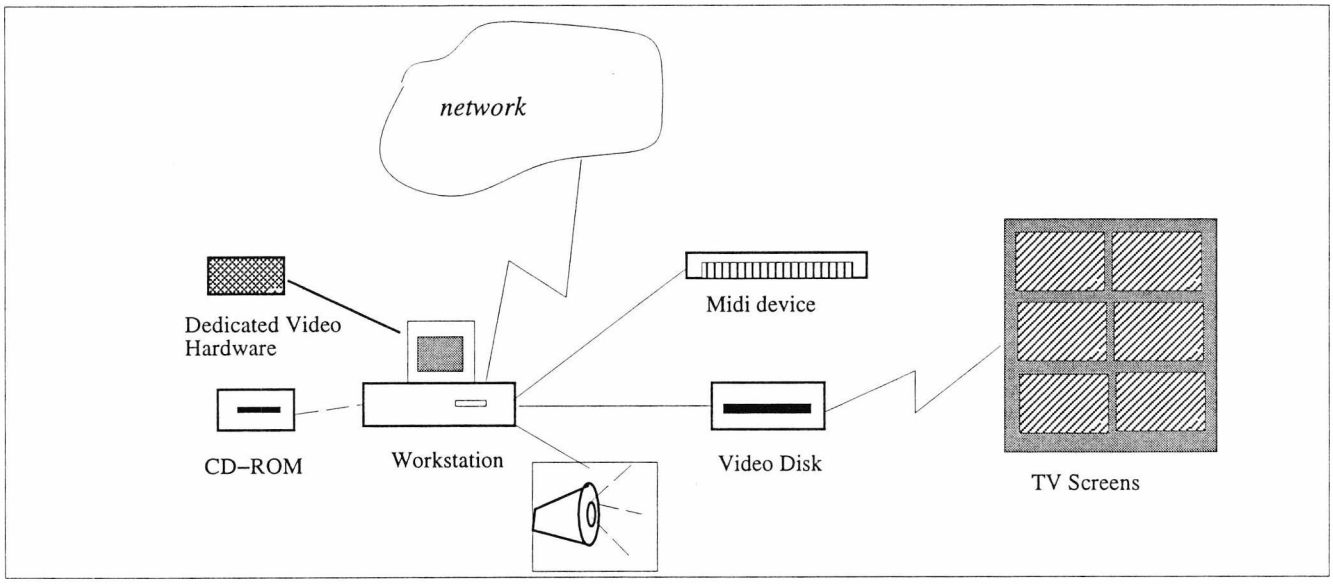

Figure 4.1: A Multimedia System

advertising. This will permit the development of new areas of communications and information processing.

Most existing Hypermedia and Multimedia Communication Systems do not provide an adequate solution for all applications. In these systems, the multimedia information is integrated into a single multimedia stream where time and events are not involved.

Figure 4.1 shows a typical multimedia system. A central workstation controls several special dedicated devices for presenting the different media. In the figure, devices like a video disk can be used to display video on TV screens asynchronously. The same video could be presented on the host's monitor using a video card.

The workstation only has to start playing the video and all further processing can proceed with no control from the central system. In the case of images stored on the CD-ROM, the workstation must keep a closer control, retrieving the contents data and controlling the display of the window that frames the images.

From figure 4.1, it can be seen that a multimedia system is inherently distributed and that distribution transparency should be carefully planned: a file system can (usually) be transparent to the user but although it may be desired to have the possibility of presenting a content object on more than one device, it is required that the user defines

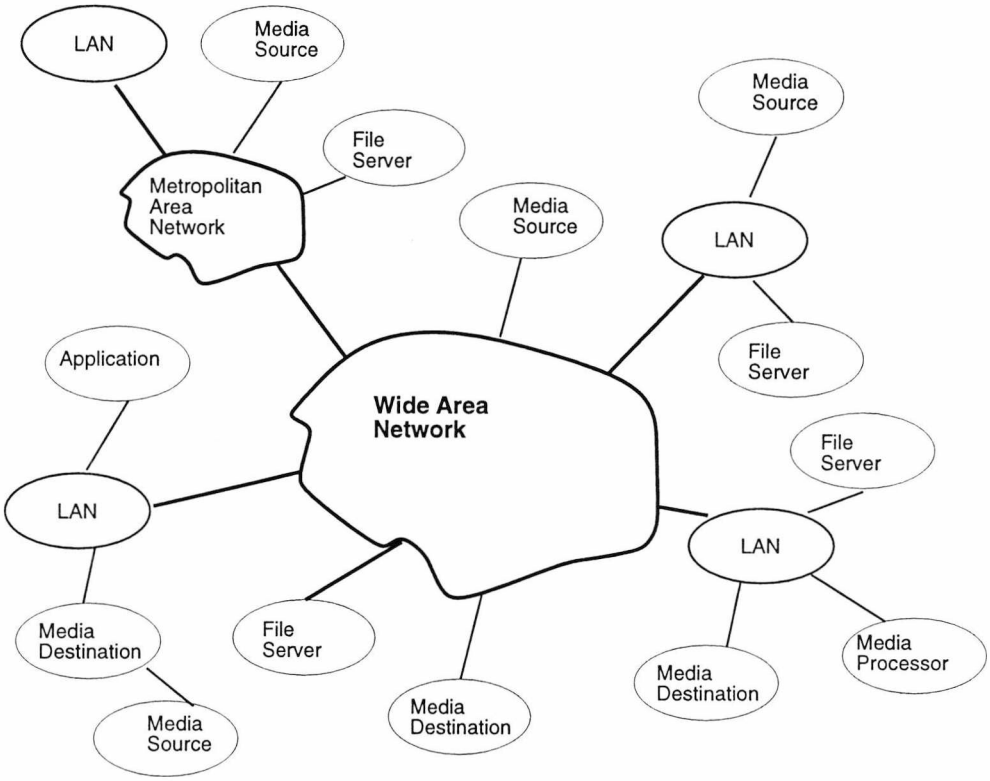

Figure 4.2: A Distributed Multimedia Environment

where the device is physically located. Figure 4.2 gives an idea of how a distributed system may be structured.

It is also impossible to predict the availability of new media in the future, and a multimedia model should be able to accommodate a new medium as seamlessly as possible.

This chapter discusses some of the requirements that should be met by the implementation of such a system. The chapter is structured as follows:

- Initially the general requirements that all hypermedia systems should meet are presented;
- Then the desirable features for the operating system are presented;
- The next section presents an overview of the operating system chosen (Windows 3.1) for the prototype implementation.

- Finally, a summary of available naming schemes and existing naming servers is presented.

4.2 General requirements

There is effectively a consensus in the scientific community that all hypermedia systems should provide:

- *Separation of data and structure:* raw data should be completely separated from the structure that binds it together. When using mark up languages, the mark up should not be embedded in the data. This principle should be applied not only for logical structures such as hypertext links but also to presentation information.

  The separation of content and structure allows users to have their personal view of the contents by changing how documents are created, but it also has potential commercial implications as there is the possibility of purchasing different structures for the same data. This is what happens today, for example with directory services that sell *structure services* over freely accessible information such as telephone numbers, or even companies that sell directories of free software accessible by anonymous ftp without actually providing the software themselves.

  The requirement for separation of data and structure has serious implications for the software that will manipulate this information, especially with regards to software integration; if the content information is changed, all structure that relates to it should also be updated.

- *Reuse of existing tools:* every organisation has a considerable investment in software which represents not only the cost of the software itself but also the cost of training employees in its use and the value of the data represented by output from existing (usually proprietary) systems. A new multimedia system should be capable of making use of existing tools for manipulating individual media.

- *Distribution:* distributed applications now are used not only at scientific and military sites but even in medium sized companies. The widespread use of the Internet is also adding extra requirements for *"every day"* applications.

  Although some success can be achieved in adding distribution to existing systems, as in [Brown, 1994], distribution and network access should be planned for explicitly.

  Distribution happens at the level of contents data, where a document may include some contents stored anywhere in the world, and also at the presentation level when a document may be *run* using several machines; for example, the audio may be handled by a special midi device, video by another device and the whole presentation process may be happening at more that one location simultaneously.

- *Support for heterogeneity of architecture:* this is a side effect of the distribution requirement. As most organisations make use of several different machine architectures and it is desirable to have all machines interconnected, a multimedia system should be planned to cope with different architectures, communication protocols and network services.

- *Extensibility:* new services, hardware and even media are constantly being introduced into the market. A new system should be flexible enough to be able to accommodate the extra facilities without a great impact on existing documents and software. As an example, access to the World Wide Web (described in section 2.12) is today almost a mandatory requirement for a hypermedia system. Existing systems, such as Microcosm (described in section 1.3.4) had to be adapted to integrate to the Web.

4.2.1 Adding new media and devices

Any Open Hypermedia system must be able to accommodate new media and devices as they appear. When new media or devices are added to an existing environment, some

general requirements must be analysed:

- *The processing the device performs on its data:* this determines the timing requirements and constraints imposed/required by the device;
- *The types of data that can be processed:* this determines the format of interfaces to devices or software controlling the new medium;
- *The stream position and control vocabulary required:* this determines if any stream interface or position keys will be needed.

To be extensible, a system must be able to accept a wide range of devices and media with different answers to the above points without requiring changes in its basic structure.

4.3 Operating System

The implementation of run time support for a multimedia system with real time constraints and distribution sometimes makes the border between operating system and application blurred.

In this section, the requirements of the application which affect the operating system used are discussed.

4.3.1 Cooperative *vs* Preemptive Operating Systems

One of the main tasks of an operating system is to decide which process to run when more than one is runnable. Several strategies for scheduling could be used [Tanenbaum, 1992a], including fairness, efficiency, response time, turnaround or throughput.

Two opposite strategies are:

- *Preemptive scheduling:* The scheduler gives a time slice of the CPU in turn to all competing runnable processes. Priorities can be defined and the scheduler

can take advantage of events such as a process having to wait for I/O to optimise CPU usage. An example of a preemptive operating system is the Unix operating system.

- *Run to completion scheduling:* in this case, only one process is run until completion. This algorithm is simpler than a preemptive one and is nowadays restricted to dedicated systems.

One scheduling strategy that falls in between these two is *cooperative scheduling*. Under a cooperative operating system, more than one process can run, sharing the computer resources. Unlike a preemptive system, in a cooperative scheduler, the operating system does not enforce a division of CPU time between the processes but relies on the *behaviour* of each process to yield control at certain points. A badly behaved process can take over all resources until its completion. An example of a *cooperative operating system* is MS-Windows 3.1, which is described in more detail in section 4.5.

The choice of a cooperative or a preemptive operating system depends on the application's timing requirements and leads to distinct programming disciplines. The non-preemptive approach is better in some cases; to use it we must be able to guarantee an upper bound on the execution times of all threads (between points where a dispatching system call is made), and this must be less than the minimum required response time for any thread (the time from an interrupt occurring to the thread running). In many cases this can be ensured for longer running threads by having them call a "reschedule" system call every so often (that is, a call that permits the kernel to reschedule if a higher priority thread is ready).

If these conditions are true, then non-preemptive scheduling offers some advantages because:

- Scheduling issues are simplified as non-preemption eliminates most of the need for explicit synchronisation and critical sections as a task can only lose the processor

at discrete points in its code.

Consequently it:

- reduces design and debug time as the number of critical sections are reduced because atomicity of most operations on shared data is inherently ensured.
- improves code quality (less potential for nasty bugs).
- reduces OS run-time overhead as context changes due to task switching are less frequent.

- Saves memory as the operating system needs to store less contextual information for the running process when it is halted.

On the other hand, the programmer must be aware that each application must be well behaved otherwise the feeling of multitasking is lost. Under certain conditions, this may cause the code to be more complex as time consuming operations must be interrupted frequently to give a chance for other applications to run, and the application programmer is responsible for providing these interruptions. In the case of operations such as reading or writing a large file, the I/O routine must be careful not to take control for too long as I/O operations can be time demanding if there is no hardware support for asynchronous operations [Silberschatz and Galvin, 1994]; this is commonly the case with desktop computers.

4.4 Future Operating Systems

Open distributed Multimedia use is one of the forces driving the development of future operating systems. Resource sharing needs to be widespread with machines required to be both client and server at the same time, breaking the current model where a desktop machine is usually the client of a larger remote machine.

Operating systems must be able to protect applications from each other. Because the *run to completion* requirements are often not met by applications, an operating

system should also be able to provide a preemptive scheduling mechanism where all applications are treated fairly. As processor costs are reduced, the operating system should also be capable of supporting symmetric multiprocessing (SMP) hardware.

The world wide level of interconnection required by future applications places heavier requirements on operating systems in the following areas [Orfali *et al.*, 1995] [Linington, 1992]:

- *Location transparency:* users, servers and resources should be able to move from one place in the network to another without disruption;
- *Name space transparency:* names must resolve uniquely within a given context or naming authority, but the operating system should provide support for a set of federated name spaces;
- *Administrative transparency:* the operating system should be responsible for synchronising clocks, and handling updates when replicated services are being provided;
- *Secured-access transparency:* users should be able to access the resources they are entitled to use from anywhere, even when using insecure telephone lines. Authentication, using mechanisms such as Kerberos [Kohl *et al.*, 1994], should regulate users' access;
- *Communications transparency:* users should not be aware of the protocols involved in the communication. Mechanisms such as Remote Procedure Call (RPC) provide transparency so that, from the client point of view, a call to a procedure running on a machine thousands of miles away can be handled just like a call to a local procedure;
- *Consistency checking:* the infrastructure should provide means to check consistency between the human interface, programming interface and communication protocols.

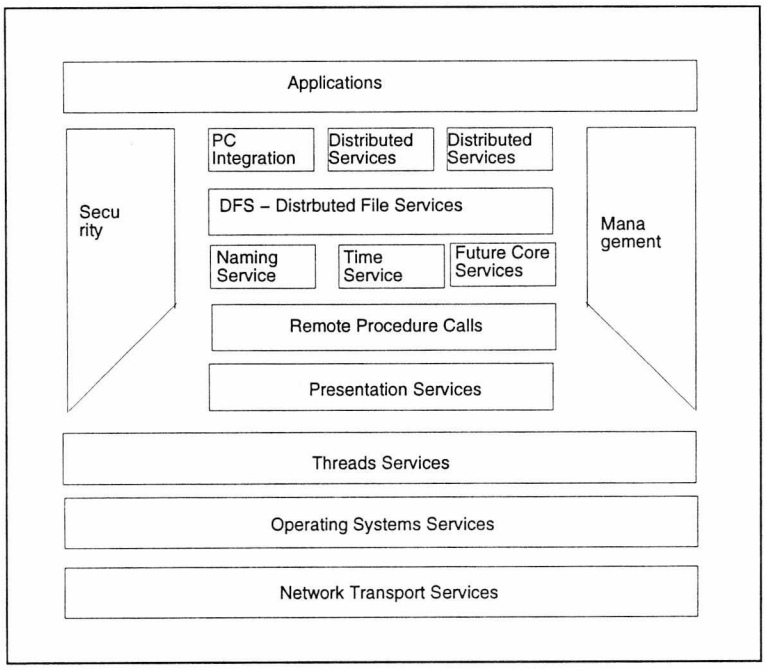

Figure 4.3: DCE architecture (from [Berson, 1992])

The tendency, therefore, is for the development of an open distributed system. Among the providers of such technologies today, the Open Software Foundation (OSF) provides a moderately complete approach with the *Distributed Computing Environment* (DCE) (figure 4.3) which consists of the following components [Berson, 1992]:

- Distributed file system;
- Directory service;
- Remote procedure call;
- Threads services;
- Time services.

Other components such as distributed file systems (e.g. the Andrew file system) also provide desirable features to a distributed environment.

4.4.1 CORBA

The most complete approach to open distributed systems today is offered by the *Object Management Group* in the form of the *Common Object Request Broker* (CORBA) [Digital Equipment *et al.*, 1993, Soley, 1990]. OMG is a non profit international trade association, composed of more than 400 members, including several large corporations such as DEC, HP and NCR. Their objective is to define an open software architecture in which object components written by different vendors can inter-operate across networks and operating systems.

The Object Management Architecture provides an infrastructure for distributed objects. It consists of the following components:

- *The Object Request Broker (ORB):* which provides the infrastructure for object communication;
- *The Object Services:* which control the life cycle of objects, including functions to create objects, and to control access to objects;
- *The Common Facilities:* which provide a set of configurable generic applications such as printing facilities, electronic mail, etc;
- *The Application Objects:* which represent application objects. An application object is usually derived from a set of basic object classes by using inheritance.

The latest version of CORBA (CORBA 2.0) includes the description of the *Object Request Broker* and the definition of the *Interface Definition Language* that allows interaction between objects in the same ORB. It also defines protocols for the interoperation of several ORBs, with two versions: a lightweight one based directly on IP and, optionally, another based on DCE [Orfali *et al.*, 1995].

4.5 Windows 3.1 Operating System

Although MS-Windows 3.1 provides few of the desirable features discussed above, it was chosen as the platform on which to implement the prototype because of the size of its installed base, and because one of the aims of this work is to define an architecture for MHEG objects usable on desktop computers. The existence of several dedicated pieces of hardware for multimedia, supported by MS-Windows 3.1, was another point taken into account when the operating system was chosen.

MS-Windows 3.1 is not a full-fledged operating system, as it runs on top of MS-DOS. However it provides the user with the feeling of a cooperative multitasking operating system. The file system is still managed by MS-DOS while MS-Windows 3.1 handles the other pieces of hardware and is responsible for memory management, program execution and scheduling [Charles Petzold, 1992].

Processing under windows is based on a *Window Procedure* and each process under MS-Windows 3.1 is a window although not necessarily a *visible window*. The Window Procedure is called by MS-Windows 3.1 and the process retains the CPU until it yields control. A window, therefore, should not take control for a long time to give chance for other processes to run. A program usually runs around the loop:

```
while (GetMessage((LPMSG) &msg, NULL, 0, 0))
      {
       TranslateMessage((LPMSG) &msg);
       DispatchMessage((LPMSG) &msg);
      }
```

that retrieves any messages to the Window that are to be processed by the *Window Procedure*. At each `GetMessage` cycle, the window yields control to the operating system which can then schedule a new process. The MS-Windows 3.1 environment can, however, call the main window procedure directly, which makes the environment a mixture of the message and call back paradigms.

Windows messages

The loop above shows that an application will usually retrieve messages from a queue managed by Windows, and will process each message internally. Typically, the process will hold the CPU during the whole time it takes to process a message. The call to `GetMessage` is an indication to the operating system that this process may be suspended and another application scheduled. If the process is performing a very demanding routine, it should frequently suspend itself by informing the operating system, using a mechanism such as the above, so that another process may be run.

A window message is a simple structure defined as follows:

```
typedef struct tagMSG
{
    HWND    hwnd;
    UINT    message;
    WPARAM  wParam;
    LPARAM  lParam;
    DWORD   time;
    POINT   pt;
} MSG;
```

where `WPARAM` (an `unsigned int`) and `LPARAM` (a `LONG`) are used to transfer parameters to the process. Typically the application receives messages directly from the operating system (such as those concerned with mouse or keyboard events, or window positioning) and not from other applications. From this structure, it can be seen that the amount of information a message can carry is very limited. To provide communications between processes, MS-Windows 3.1 defines other mechanisms, discussed below.

4.5.1 Interprocess Communication under Windows 3.1

Interprocess communication under MS-Windows 3.1 is based primarily around shared memory. Typically applications communicate by three mechanisms:

1. Dynamic Data Exchange (DDE);

2. Object Linking and Embedding (OLE);

3. The Clipboard;

The following sections provide an overview of each of these mechanisms.

4.5.2 Dynamic Data Exchange (DDE)

Dynamic Data Exchange (DDE) is an interprocess communication method that uses shared memory to exchange data between applications and implements a protocol to synchronise the passing of data. DDE applications fall into four categories [Clark, 1992]:

1. *Client:* an application that requests data or services from another application;

2. *Server:* an application that responds to a client application with data or services;

3. *Client/server:* an application that is both a client and a server;

4. *Monitor:* a monitor is an application that can intercept DDE messages from other DDE applications but cannot act on them.

A DDE application can have multiple concurrent conversations. Within a conversation messages are handled synchronously but the application may switch between applications asynchronously.

DDE applications must uniquely define all conversations. The conversation is defined by the server and client applications windows' handlers and each conversation is managed by a hidden window. If a client application needs to have more than one conversation with a server, a new (hidden) window must be created for each conversation.

The DDE protocol defines a hierarchy to identify the desired data. DDE defines a three-tiered identification scheme:

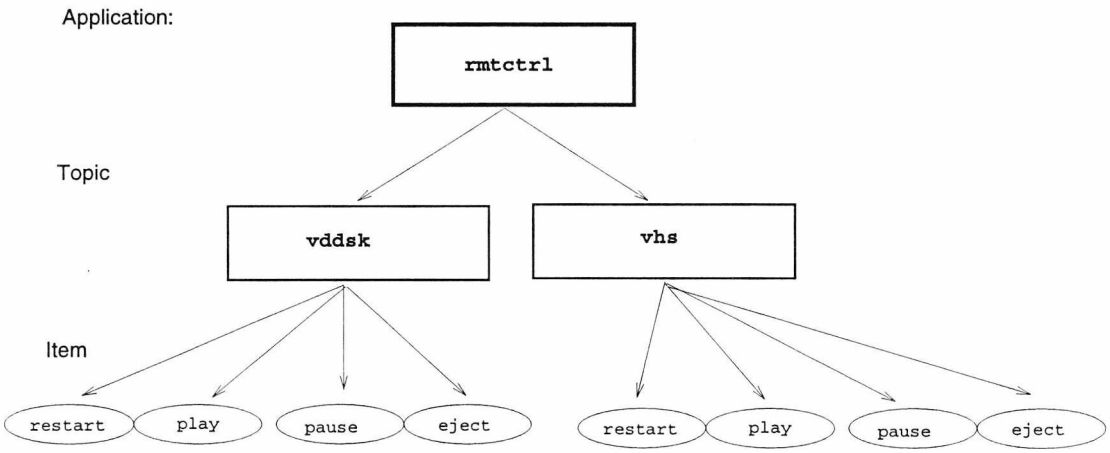

Figure 4.4: Example of a DDE Server

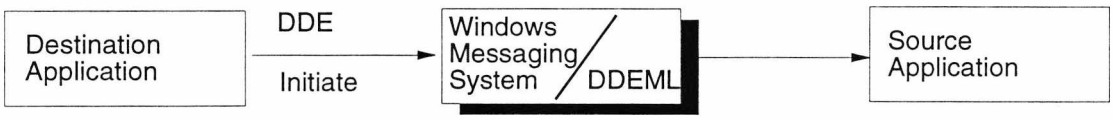

Figure 4.5: Example of a DDE Conversation

1. *Application:* this is the top level of the hierarchy and identifies the program, i.e. the *application* that provides the service (the server);

2. *Topic:* each DDE server must provide at least one topic and it may provide several topics.

3. *Item:* the application may provide several items within a topic.

For example, if we had an application that worked as a remote control for a Videodisk and a VHS player, we could define the services it provides (figure 4.4) as:

- *Application:* `rmtctrl` (the name of the program);
- *Topic:* there would be two topics: `vddsk` and `vhs`;
- *Item:* each topic would have the items `restart, play, pause, eject`.

There are three basic types of DDE conversations (figure 4.5) which are briefly described:

1. *The cold link:* A cold link is established when a client application broadcasts a message identifying the application and topic it requires. A server that provides the topic acknowledges the message and then the client requests a specific item which is acknowledged if the server supports it. If the server does not support the item, it posts a negative acknowledgement. The conversation continues with the client requesting items from the server. Either side of the conversation may end the connection.

   What characterises a cold link is that data is only transferred in response to clients requests. If some data changes in the server, the client will only receive the updated information if it make a new request.

2. *The hot link:* a hot link allows servers to inform clients that some data has changed. When an item changes, the server posts a message notifying the client of the update and transfers the new information.

3. *The warm link:* a warm link combines elements of hot and cold links. In this case, the server notifies the client that some data has been updated but unlike the hot link, the server does not automatically transfer the new information. The client must then request the item if it so desires.

The protocol also allows a client to send unsolicited data to the server, and to send a command string to be executed by the server.

The DDEML library

The DDEML provides an application programming interface (API) that simplifies the task of adding DDE capabilities to a Windows application. Instead of sending, posting, and processing DDE messages directly, an application uses the functions provided by the DDEML to manage DDE conversations. The DDEML also provides a facility for managing the strings and data that are shared among DDE applications. DDEML provides a service that makes it possible for a server application to register the service

names that it supports. The names are broadcast to other applications in the system, which can then use these names to connect to the server. The DDEML also ensures compatibility among DDE applications by forcing them to implement the DDE protocol in a consistent manner.

4.5.3 Object Linking and Embedding (OLE)

Object Linking and Embedding (OLE) is a mechanism for inter application communication. OLE introduced the idea of a document-centered approach instead of an application-centered approach. In the document-centered approach, if there is a spreadsheet embedded in a text document, and if the author wants to change its contents, he will be able to activate a spreadsheet processor directly from the word processor to do the processing instead of having independently to activate a spreadsheet, perform the changes and then import the updated information into the word processor again.

The first version of OLE used the DDE protocol as the communication infrastructure. A second version used a flavour of Remote Procedure Call (RPC) that Microsoft called *Light Remote Procedure Call (LRPC)* [Microsoft, 1994] in the sense that it is based on shared memory access and therefore only provides for communication between processes running on the same machine. LRPC is not a communication protocol, as it does not require a conversation to be established between the processes involved.

Future versions of OLE, Distributed OLE, should allow access to services such as visual controls, multimedia services, data-access services, name services and distributed security [Pleas, 1994].

4.5.4 Clipboard

The Clipboard is the standard Windows method of transferring data between a source and a destination application. Data is transferred via the Clipboard by direct user interaction, for example, when data is copied from one application (eg. a spreadsheet)

to another one (eg. a word processor).

There are several standard data formats used to transfer data via the clipboard. These include metafiles, text, bitmaps, and others. An application can also define its own data format.

Under *Windows for Work Groups* (discussed below), the clipboard may also be shared allowing users to communicate data across the network.

4.5.5 Windows for Work Groups

Windows for Workgroups is a version of Windows 3.1 with added network capabilities. It includes peer-to-peer file, printer, and clipboard sharing, mail capabilities, group scheduling, chatting functions, and simple network-monitoring tools. The security provided by Windows for Work Groups is very limited.

Network DDE

Network DDE (Net DDE) is an extension to DDE available for Windows for Workgroups [Matthews and Dobson, 1993]. It works on top of the existing DDE system and monitors DDE conversations. If a conversation takes place with a remote system, Net DDE routes the data over the network as shown in figure 4.6. Net DDE intercepts DDE initiated messages targeted to a remote application; the message is then routed out to the network where it is picked up by Net DDE on the server's computer which then translates it into a standard DDE initiate message.

Under Net DDE, client applications need to know the server's name to establish a conversation, which is still a problem for many applications running under Windows where the server application name is hard coded. Net DDE also provides very little protection for shared resources.

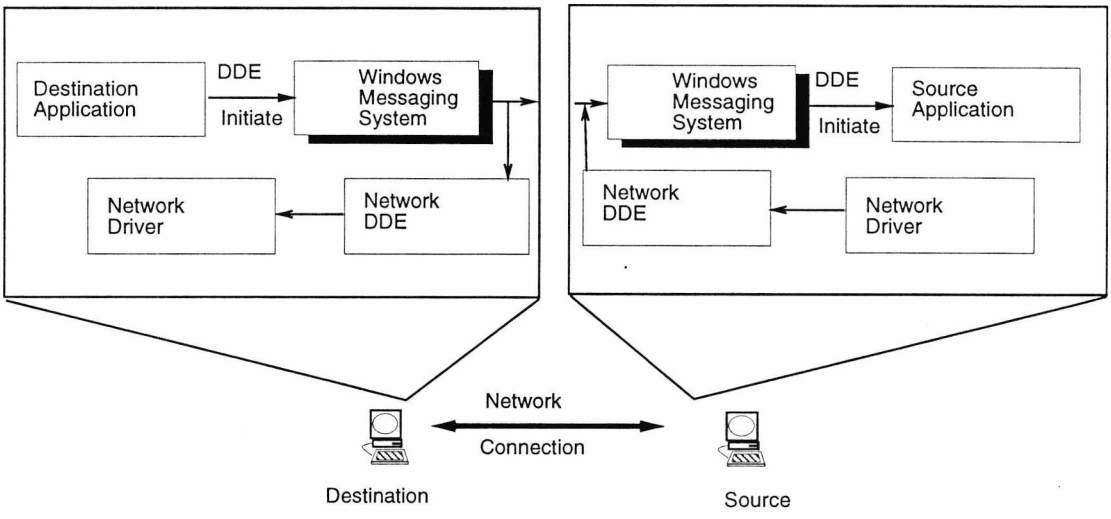

Figure 4.6: Example of a NetDDE Conversation

4.5.6 Dynamic Link Libraries (DLL)

MS-Windows 3.1 uses *Dynamic Link Libraries* (DLL) to optimise resource usage. In a DLL, memory is shared by all processes using it. There are two types of DLL:

1. *Code libraries* that contain executable code. Exported functions from these libraries can be used by several processes. One example is a DLL to emulate the floating point co-processor.

2. *Resource-only libraries* that may store, for example, bitmaps or fonts.

MS-Windows 3.1 system itself is basically built around three DLLs:

- *KRNL386* responsible for memory management, loading and executing programs and scheduling;

- *User* that manages the user interface and windowing;

- *GDI* that is responsible for the graphics.

4.6 Naming

This is probably the most crucial aspect of design and standardisation in an open hypermedia system. It is concerned with the syntax of a name by which a document or part of a document is referenced from anywhere else in the world.

Since many protocols are currently used for information retrieval, the address must be capable of encompassing many protocols, access methods or, indeed, naming schemes. For example, the WWW scheme uses a prefix to give the addressing sub-scheme, and then a syntax dependent on the prefix used, in order to be open to any new naming systems.

4.6.1 Name or Address, or Identifier?

Conventionally, a "name" has tended to mean a logical way of referring to an object in some abstract name space, while the term "address" has been used for something which specifies the physical location. The term "unique identifier" generally referred to a name which was guaranteed to be unique but had little significance as regards the logical name or physical address. A name server was used to convert names or unique identifiers into addresses.

With wide-area distributed systems, this distinction is blurred. Locally, things which at first look like physical addresses develop more and more levels of translation, so that they cease to give the actual location of the object. At the same time, a logical name or a unique identifier must contain some information which allows the name server to know where to start looking. In a global context, for example `"123715924234624423423234234242346876234236 8"` might well be unique, but it contains insufficient (apparent) structure for a name server to look it up. The name `"info.cern.ch"` has a structure which allows a search to be made in several stages. In fact, practical systems using unique identifiers generally hide within them some clues for the name server, such as a node name.

A hypertext link to a document ought to be specified using the most abstract name possible, as opposed to a physical address. This is (almost) the only way of getting over the problem of documents being physically moved. As the naming scheme becomes more abstract, resolving the name becomes less of a simple look-up and more of a search.

It is expected in practice that the translation of a document name will take several stages as the name becomes less abstract and more physical.

Some document reference formats contain "hints" to the reader about the document, such as server availability, copyright status, last known physical address and data formats. It is very important not to confuse these with the document's name, as the hints have a shorter lifetime than the document.

If this direction is chosen for naming, it still leaves open the question of the format of the address into which a document name will be translated. This must also be left as open-ended as the set of protocols.

4.6.2 The Global Name Service

The *The Global Name Service* (GNS) was designed and implemented at DEC Systems Research Center [Lampson, 1986]. It was a descendent of Grapevine [Birrel *et al.*, 1982] which was one of the first extensible, multi-domain name services.

Some of the goals to be achieved in the GNS project were:

- *Large size:* the system should be capable of handling an essentially unlimited number of names and to serve an arbitrary number of administrative organisations. Its predecessor, Grapevine, was designed to be scalable over at least two orders of magnitude in the size of the name space and the load of requests that it could handle;

- *Long lifetime:* many changes will occur in the organisation of the name space and in the components that implement the service during its lifetime;

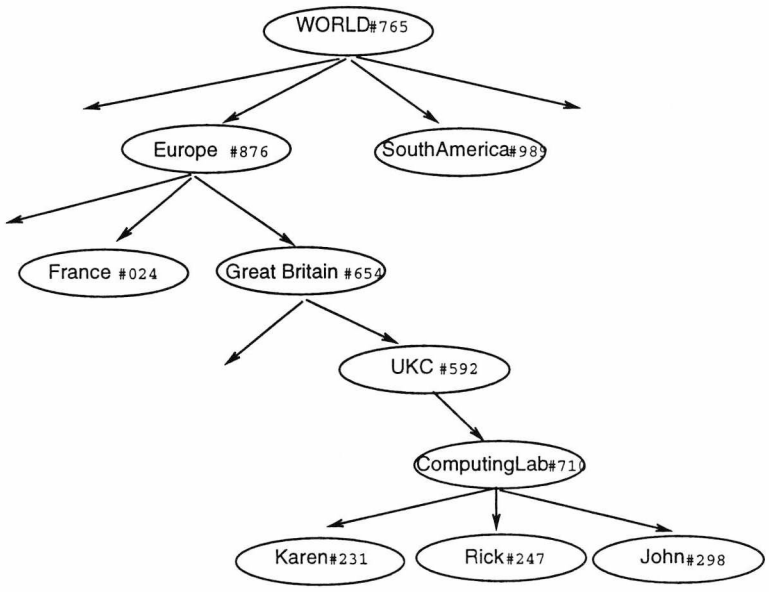

Figure 4.7: Example of a GNS Directory Tree

- *High availability:* most other systems depend upon the name service; they cannot work when it is broken;
- *Fault isolation:* local failures must not cause the entire service to fail;
- *Tolerance of mistrust:* a large open system cannot have any component that must be trusted by all the clients in the system.

In GNS, the user sees a hierarchy, like a file directory in an operating system such as Unix. Figure 4.7 gives an example of a directory tree. Directories have unique identifiers (DI) issued by a central host, and an arc labeled with a DI is a *Directory Reference* (DR). In the figure, UKC is directory `WORLD/Europe/GreatBritain/UKC` or DI `#592`. The Computing Lab can therefore be identified as either `WORLD/Europe/-GreatBritain/UKC/ComputingLab` or `#592/ComputingLab`. The use of DIs provides flexibility for change and growth of the naming structure, as a DI is always correctly resolved in any environment and may be moved within a hierarchy. A detailed explanation of the scheme can be found in [Needham, 1989].

This structure provides flexibility for change. If part of the tree moves to a different place, it is only necessary to keep a "symbolic link" (i.e. an *alias*) pointing from where

it originally was to where it has moved to.

4.6.3 The X.500 directory

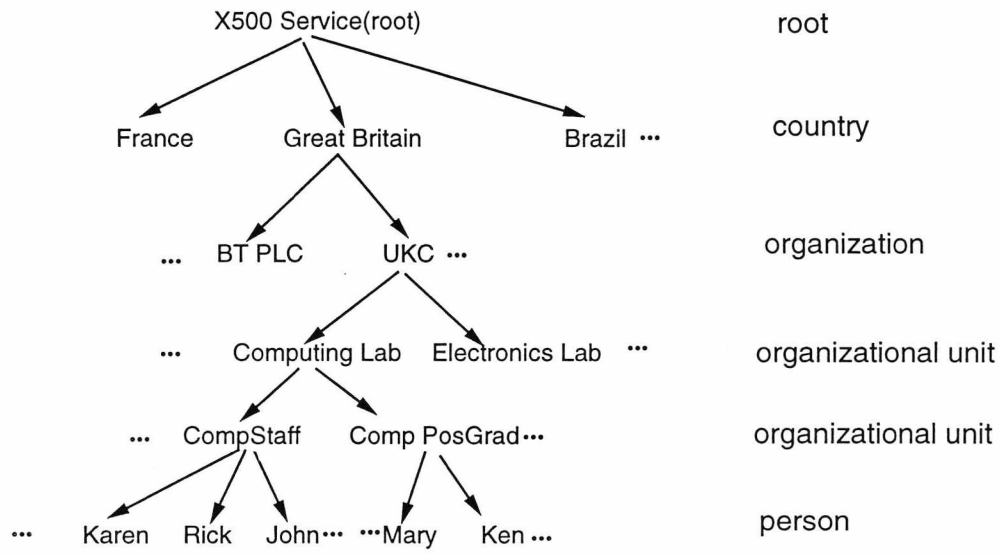

Figure 4.8: X.500 Directory Information Tree

The X.500 [CCITT, 1988] directory service defines an abstract attribute-based name space which is hierarchical as seen in figure 4.8 (adapted from [Coulouris *et al.*, 1994]). The whole tree is called the *Directory Information Tree* (DIT) and the associated directory structure including the data is called the *Directory Information Base*.

X.500 allows objects such as organisations, people, and documents to be arranged in a tree. Whereas the hierarchical structure might make it difficult to decide in which of two locations to put an object (it's not hypertext), this does allow a unique name to be given for anything in the tree. X500 functionally seems to meet the needs of the logical name space in a wide-area hypertext system. Implementations are still somewhat rare, so it cannot be assumed as a general infrastructure. As of 1992 there were 177 servers, holding a total of about 300,000 entries and serving 370 organisations connected to the Internet [Coulouris *et al.*, 1994].

As a directory service, X500 can be used to resolve resource names that are known precisely and also to resolve imprecise queries such as to retrieve the names of users in

a given company, since nodes in the tree store a wide range of attributes.

A server in X.500 is called a *Directory Service Agent* (DSA) and a client is a *Directory User Agent* (DUA). A client usually interacts with a server that in turn may have to interact with other DSAs to resolve a query or redirect the client to another server.

There are two methods of access to the directory:

1. *Read:* an absolute or relative name is given together with the attributes to be read. The DSA locates the entry by navigating in the DIT, passing requests to other DSAs if required and returning the requested attributes to the client;

2. *Search:* in this case, a filter is passed along with a base name that specifies the node in the DIT where the search is to start. The filter (a boolean expression) is evaluated by every node below the base node.

   Additional attributes may be specified to limit the scope of the search in order to reduce resource usage.

4.7 Final remarks

This Chapter has provided an overview of the requirements imposed on the design of an open multimedia system and has also given an overview on the current available technology.

The main points that should be taken into account are:

- The system must be *open* in the sense that it must make use of a heterogeneous environment, and the design should be as independent of applications as possible;

- The system must be *extensible*, accommodating the addition of new technology such as new media and new devices;

- It must be explicitly designed to make use of distribution, as later changes to the

system to accommodate distribution and make full use of networks are usually difficult, and their results not perfect.

Chapter 5

Architecture and implementation

This chapter proposes an architecture for MHEG objects which meets the *Requirements and Constraints* from chapter 4. The architecture is suitable for use in an environment composed of desktop computers, running MS-Windows 3.1. The implementation of a prototype of the architecture is also discussed.

5.1 Introduction

As discussed in chapter 3, there is a clear separation (for exchange purposes) in MHEG between behaviour, content, interaction and composition of objects. Exchanged objects are not themselves presented directly. Run time copies, adding dynamic behaviour, are created from the exchanged objects. During rendering of the run time objects, the contents and dynamic information are put together and managed by one process. This process will also take care of user input, scrolling, etc. Therefore, we will always be dealing with *composite* objects.

The environment used for the implementation is MS-Windows 3.1, and consequently the design will be influenced by it. In MS-Windows 3.1, a process is always a window (although not necessarily visible); therefore we have to define *windows* and the objects they will handle. An overview of the MS-Windows 3.1 operating system was presented

in section 4.5.

5.2 Architecture Overview

At a high level, the system architecture proposed can be seen (figure 5.1) as being made up of a *system kernel*[1] (described in the next section); and several *processes* (see section 5.4) (windows) that use the resources provided by the kernel for synchronisation and communication.

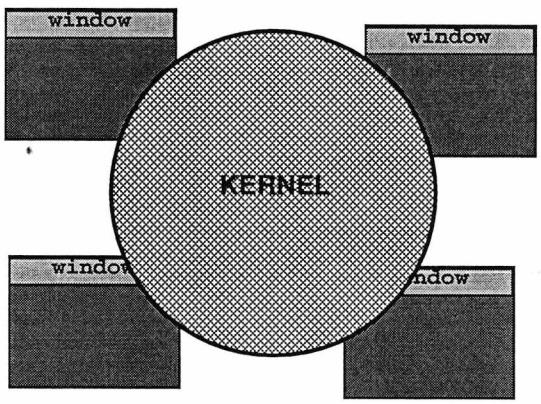

Figure 5.1: High level system overview

Figure 5.1 represents the structure when only one host is being used. In the generic case, more that one host will process the presentation and one kernel will be present in each of them. All communication across hosts' boundaries will be performed between the kernels, as shown in figure 5.2. Typically, kernel processes running on more than one host will exchange messages for clock synchronisation, name resolution and to request remote execution of actions. The next sections will discuss the main components in the system.

[1] *System kernel* here means the main modules in the system, and not a *kernel* as used in Operating Systems

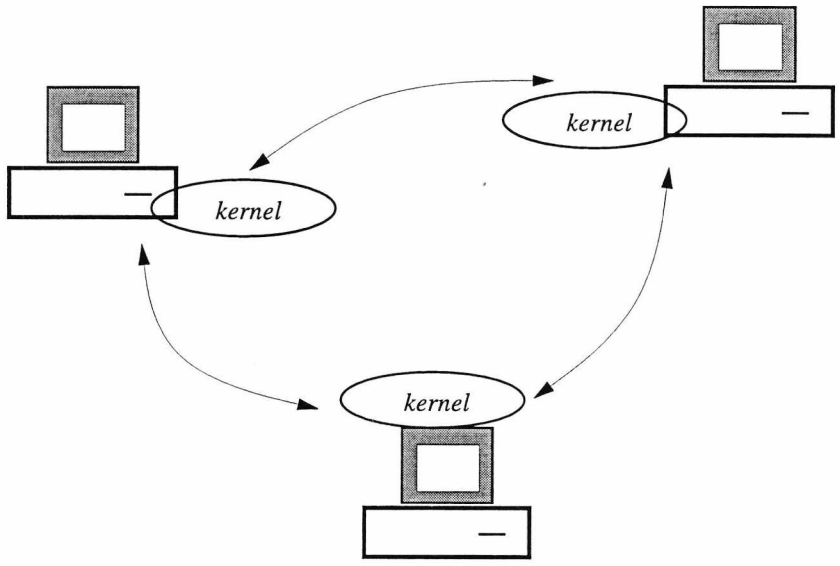

Figure 5.2: Distributed control

5.3 The kernel

The kernel is responsible for system integration by providing communication between components and external communication, name resolution and basic MHEG object decoding. There is a copy of the kernel in each *processing unit*, which means a host that is responsible for handling a set of objects. The kernel is basically an extension to the operating system providing additional services for interprocess communication (IPC), clock synchronisation and name resolution.

The system requires central services, such as a time server. It was decided that the first kernel started (the one that will handle the first object) would be the controlling one. Kernel processes started subsequently should use this kernel as a time server. The first server is regarded as essential, probably running in the workstation where most of the output is to be rendered, and if it fails, the whole presentation would be halted.

The kernel encapsulates the part of the system that needs to be changed if access to external naming services is added. Since it makes use of IPC mechanisms available from the underlying operating system to provide a transparent communication mechanism to the users processes, the kernel is the only part of the system to be updated when

a new form of IPC is provided. For example, under Windows for Workgroups, all process communication between hosts can be performed by using NetDDE (described in section 4.5.5); when the system is ported to future versions of windows using more efficient forms of communication, such as remote procedure calls, only the kernel modules need to be updated.

The kernel, running in a single machine (figure 5.3), is made up of three DLLs: the *Registry*, the *Link Factory* and the *MHEG Engine action processor*. The *MHEG Engine action processor* performs basic MHEG operations such as creating a run time object from a model object. There is also one process: the *Clock*, which provides a timer.

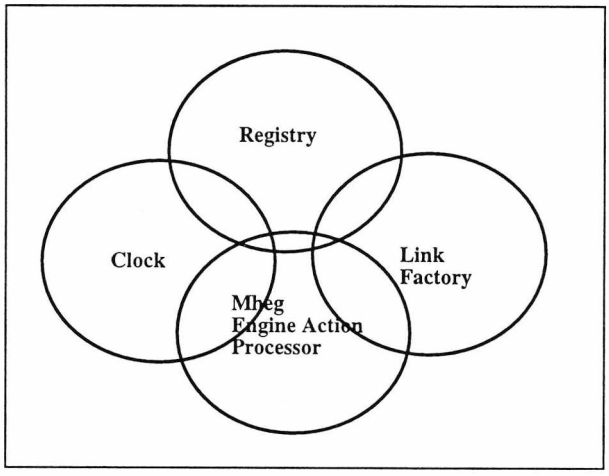

Figure 5.3: System kernel

The next sections will discuss these components in detail.

5.3.1 Link Factory

The *Link Factory* is made up of three main components (figure 5.4):

- *Decoder:* this is responsible for decoding all link objects. It reads each link in the exchange format and creates a link object in the internal format.
- *Storage:* this is a temporary storage for links which have been decoded, but for which the process holding the trigger condition has not yet been activated. A

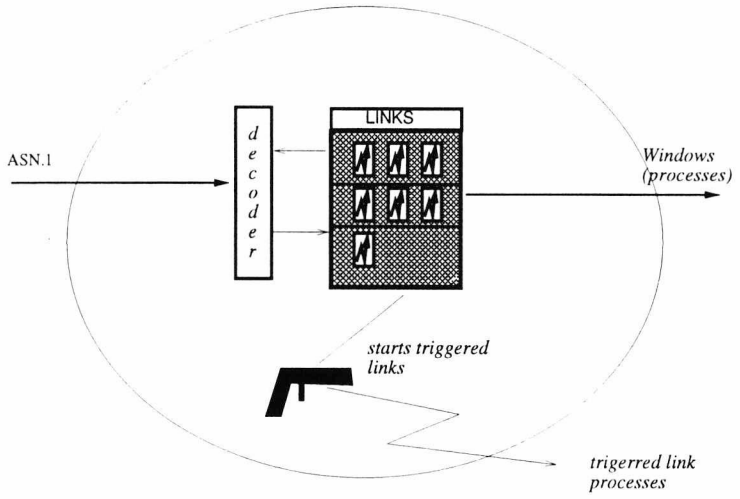

Figure 5.4: Link Factory Structure

reference to the link is kept by the registry for future retrieval by the appropriate process.

- *Triggered link starter:* this module starts an independent module to handle the link effect of a triggered link.

A link is stored in the link factory until it is explicitly destroyed, as it may be triggered several times. Once a link is triggered, a process for handling its effects is created. The dynamic behaviour of links is explained in detail in sections 5.7.2 and 5.7.3.

The *Link Factory* is implemented as a DLL shared by all processes on the same host. A DLL uses shared memory and the cost of activating its functions is the same as a regular function call, making its use very efficient in a non-distributed system.

5.3.2 The Registry

The registry provides a central point for message exchange. It was introduced as a result of experience with alternative designs. This section initially presents the first designs, the problems met in their implementations and then presents the final design.

First solution

In one of the first solutions built, the communication infrastructure was designed to be completely distributed with no central point for message exchange. All communication was performed directly between the processes[2] involved using DDE. Communication between parent and child was also performed using DDE. In this version of the prototype, the root object (a composite) would be the first process created. For each component within the composite, a process was created and two DDE conversations started between the two processes, as both parent and child could be either server or client with respect to the other. In this architecture, the only information each process had to know was who was their father and who were all their children.

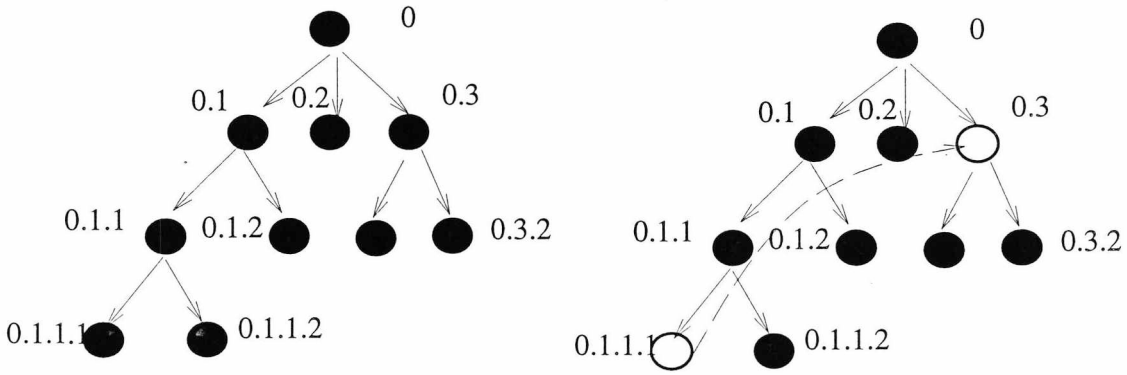

Figure 5.5: Distributed Message Passing

Figure 5.5 illustrates this situation. In the figure, suppose that the object represented by node `0.1.1.1` needs to exchange message with the object represented by node `0.3`. Initially, it would have to request its father to locate the desired object. The request would be propagated up to the root and down again to the other branches. Eventually object `0.3` is located and a DDE conversation is started directly between `0.1.1.1` and `0.3`. Actually, two conversations would be started as the destination object could also send unsolicited information. If the source node had children, there would also be a need to search for the destination node below its level.

[2]As explained in the previous chapter, a process is a window under MS-Windows 3.1

If the identification of an object was changed (by an author), the object had to communicate with all objects with which it had an open conversation and the author would have to decide either to close that conversation or to update the identification to the new value in all objects communicating with the object changed.

This approach had the advantage of allowing processes to communicate directly, not relying on a central point. However, in many cases, an object needs to be informed frequently about status changes in another process, for example to trigger a link.

Initially, it was thought that, although the cost of starting a conversation could be high, there should not be so many components in a running session as to make the location process prohibitive, and there was an apparent advantage of letting processes communicate directly with no extra delay added by an intermediate process.

However, the limitation imposed by DDE that each conversation must be handled by a process (a window), meant that even for a small number of objects, there would be a very large number of active windows leading to unacceptable system performance.

The final design

The solution proposed, which resulted in a much improved performance, is to maintain a central point where objects are registered (the *Registry*) and which is also used for exchanging messages between processes.

The *Registry*, like the *Link Factory*, is a DLL loaded by all windows. There is one central *Registry* that is loaded when the first object is activated.

For objects handled by different hosts, which cannot share memory, one registry for each host exists and each secondary registry informs the central one about its existence. All communication between hosts is kept within the boundaries of the registries and is transparent to the objects communicating.

This approach has the following advantages:

- Clients are kept simple as they only have to be know how to communicate with

the registry via its interface;

- Information can be cached at the registry;

In addition, the system can easily be extended to provide:

- *Access control:* access rights can be maintained in the local registry, if required;
- *Statistics:* user profile information and general statistical data can also be gathered locally, and on a per-session basis.

Processes also need to communicate to exchange status information, actions, objects references, etc. The Registry keeps a *post office* to provide for communication between objects and processes.

The components of the *Registry* are shown in figure 5.6; they are:

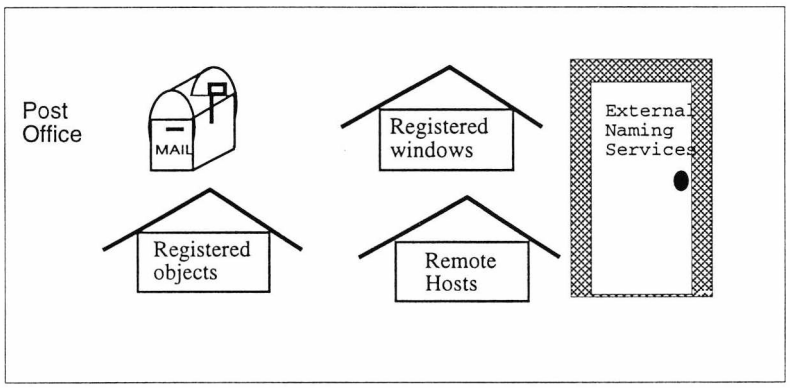

Figure 5.6: Components of Registry

- Post office: the place where messages exchanged between processes and objects within processes are processed. The post office may have to communicate with a remote office to have a message delivered;
- Registered windows: the handlers for all windows processed by the current host. This information is used by the clock process to broadcast timer information;

- Registered objects: the identification of all objects managed by the local host. The Registry keeps internal and external identifications, aliases (as defined by MHEG), and handlers for the windows managing these objects.

  The registry also keeps a cache for objects handled remotely but referenced locally. The cache is loaded during the process of locating an object: when the registry receives a request to locate the object, it first tries to find it locally in the objects it handles directly; if the search fails, it will then consult the other registries until the desired object is located. It will then "remember" the location of the object for future messages. If the desired object has moved since the last communication was established, the cache is invalidated and the process is restarted;

- Remote hosts: this information is kept when more than one host is being used in a session. It maintains the identification of all remote hosts, including the main host (the first one started) that is the time server. This information is used by the clock process for synchronisation and by the post office to deliver and retrieve remote messages.

- External naming services: in order to locate an object identified by an external name, the registry may make use of external naming services. This is the point where access to these services is provided.

Interface to the world

From the point of view of objects using it, the Registry is seen as a central point to resolve identifications (figure 5.7) and to send/receive messages. An object must register the identification of all its content objects and the process (Window) handling them.

The registry also has an interface to the outside world, as some identifications may make references to network or database objects or to other registries running on different

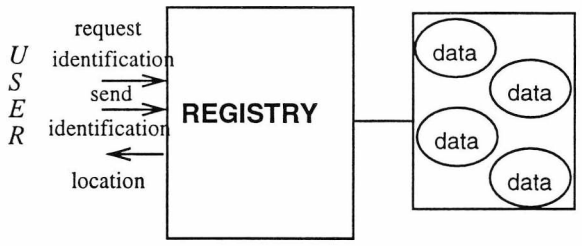

Figure 5.7: Registry as seen from a using object

machines. An object does not know that objects with which it is communicating may be remote. All external communication is handled by the local registry.

From outside MHEG, the registry is seen as a cache of identifications in use by the application with an external interface to other identification resolver processes, as shown in figure 5.8. With this approach, the registry can also be extended to provide name resolution capabilities to applications outside its scope, but which are interested in objects under its control.

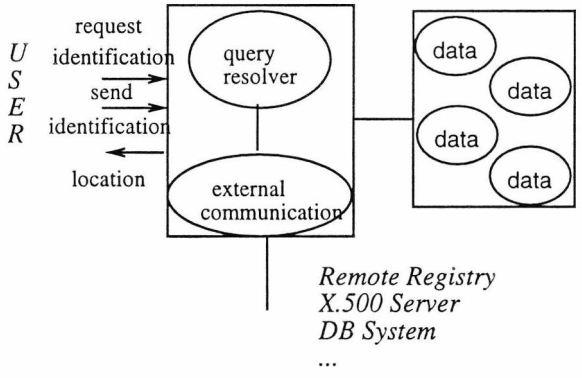

Figure 5.8: Registry as seen from outside MHEG

5.3.3 The Clock

Timers are scarce resources in the MS-Windows 3.1 environment. There is a limit of 32 active ones at any time. The maximum resolution provided is 55 ms. Setting a timer to a resolution is not a guarantee that an interrupt will occur at the precise time, as the operating system is not preemptive and a timer message is a regular Windows low priority message (i.e. it is posted to the window message queue).

A timer message is posted to the process queue like a regular window message with the guarantee that only one message will be queued at any time. This means that if more than one timer interrupt occurs before the process owning the timer gets the CPU, MS-Windows 3.1 will discard the redundant messages. The result is that if a process holds control longer than a timer tick, the distance between the ticks will be longer than the specified timer interval, and it could be shorter than that interval in some subsequent tick. Chapter 6 presents time measurements for several scheduling algorithms.

Because it is not possible to have a large number of timers running, it is not possible to propose a solution where every process uses a timer from the environment to provide its timing. To cope with this limitation, one "clock" process was implemented that uses one timer from the environment and this is the only timer provided by the environment to the system. The clock is used to schedule all processes on the same host. This clock uses information from the *Registry* to broadcast the derived messages to all registered processes.

The clock process is also responsible for synchronising with other clocks when more than one host is being used to present the objects. The first clock is loaded when the first object is started. When the first windows starts running, MS-Windows 3.1 loads the DLLs that are going to be used by this process, if they are not already loaded. The registry startup code starts the clock when it is first loaded.

One of the main goals of the scheduler is to provide a smooth presentation of objects. As we are using a non-preemptive operating system, the scheduler must avoid "clock jumps", ie., one process should not hold the CPU for a time long enough to cause the feeling that the presentation is progressing in jumps.

Synchronising clocks

For an interactive, real time system, it is important to keep track not only of the ordering of events but of the delays between them as well.

Although the system implements only one logical clock for each host, the existence

of dedicated cards with asynchronous control (like a video card) is similar to having processes running in more than one host but is simplified because memory can be shared and used for communication.

Tasks of the clock

Each timer message processed by the clock (i.e. every time the clock process holds the CPU), results in two activities:

- *Broadcast of Timer messages* to all processes registered in the registry. In this case, Clock uses MS-Windows 3.1 `PostMessage` to send the message to all windows.

  This is the place where the higher level system scheduling (or orchestration) happens. The different broadcasting policies can define higher level priorities amongst processes. Process orchestration is discussed in section 5.6.

- *Synchronise with other clocks.* The clock process uses its CPU share to synchronise with all other clocks. The first clock created is the time server and, as all processes on different hosts are also registered in the registry, this information is used to synchronise the various clocks.

  Several mechanisms for synchronising clocks exist [Tanenbaum, 1992a] which do not require a lot of communication or processing.

5.3.4 MHEG Engine Action Processor

As described in Chapter 3, MHEG objects have two distinct phases. Initially they are unknown to the system (or engine) processing them, and in the second phase they are available to the engine. The timing of an MHEG object's availability can be seen in figure 5.9:

- During phase **01** the object is not known to the engine, however, it may receive a request to `prepare` the object;

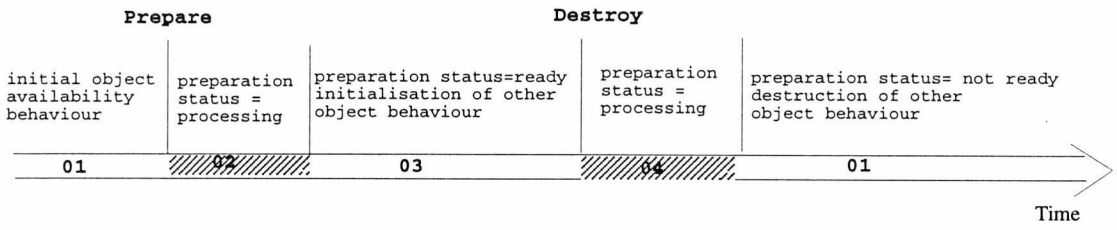

Figure 5.9: Timing diagram of model object availability

- The engine then starts preparing the object (**02**), which involves resolving the name of the requested object and it may involve retrieving data from a remote location, or from a slow video disk;
- When the object is prepared (**03**) it is available to the engine, which may then create run time instances from it;
- When the object is no longer required, it is destroyed (**04**) becoming unavailable to the engine again (**01**).

The MHEG Engine Action Processor is the module for starting the process that prepares an MHEG object. Once the object is identified, it will start a process to perform the necessary tasks to make the object available to the engine. The process that handles the preparation will then be responsible for creating runtime instances of that model object which may, in some cases, require it to make a copy of the contents data.

5.4 Processes

The processes that make use of the services provided by the kernel can be classified as:

1. *Model object processes:* these are the processes (figure 5.10) that handle *model* objects as defined in the MHEG class hierarchy, i.e. the actual MHEG objects that are exchanged, and *Container objects.* These processes will usually be transparent to the user and are responsible for preparing the object that will be

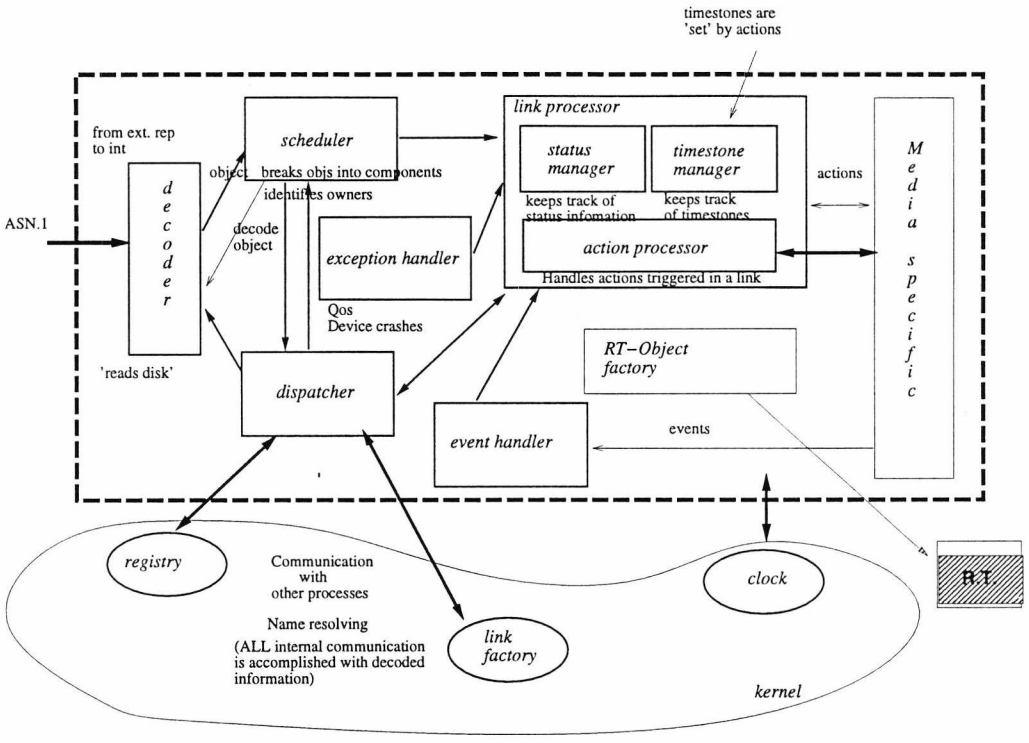

Figure 5.10: Structure of a Process Running a Model Object

presented. A *model object process* is started when the MHEG Engine Action Processor receive a request to prepare an object. Only one such process will exist at a given time for a model object.

The structure of a process in this category is given by figure 5.10; it contains:

- *The decoder:* the module responsible for transforming the object from the interchange format to the internal format (section 5.4.2);
- *The scheduler:* the module responsible for breaking up the decoded object into components and internally scheduling the process activities so that the process will not take too much CPU time in one cycle;
- *The exception handler:* deals with external exceptions such as devices crashes;
- *The dispatcher:* the module responsible for all communication between the process and the *kernel.*

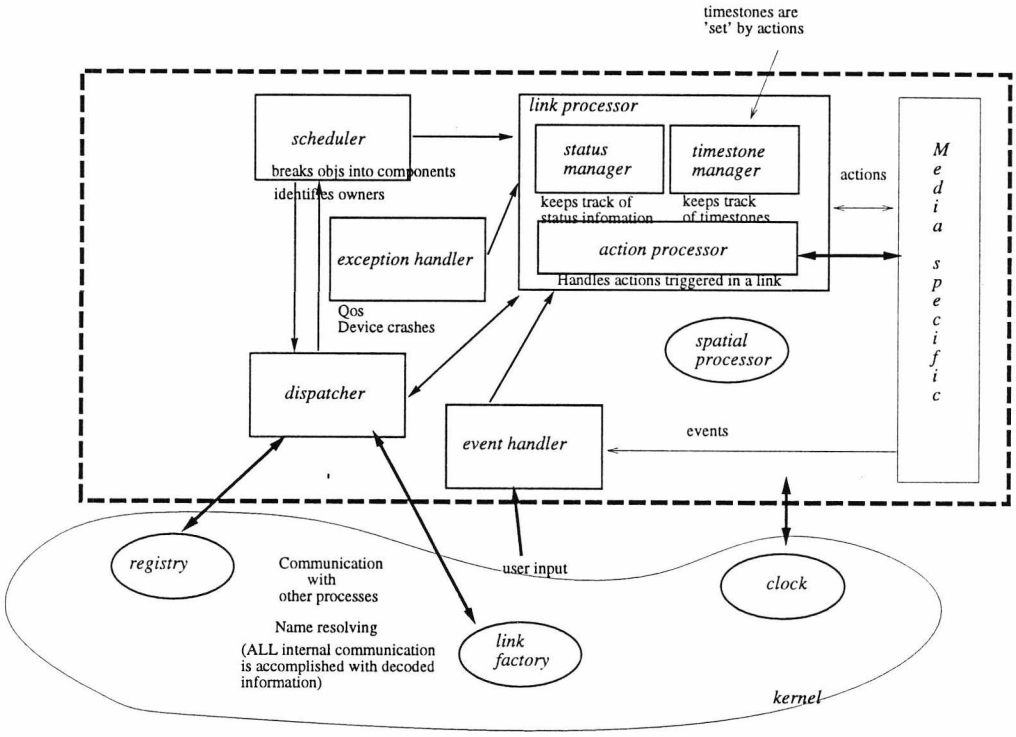

Figure 5.11: Structure of a Process Running a rt-object

- *The link processor:* responsible for handling all link related activities as described in section 5.4.3.
- *The event handler:* deals with external interruptions;
- *The rt-object factory:* creates run time instances of the object;
- *The media specific (processor):* the module that deals with the particulars of each medium (section 5.4.5).

2. *Run-time object processes:* these are the processes that *present* model objects to the user. Run time objects are created from model objects. For example, a model object can be a video and one or more run time instances of the video sequence may be used in a presentation but all instances will be created from the same video model.

   The structure of a run time process is slightly different from an MHEG object process and is shown in figure 5.11. A run time process does not create other run

time objects therefore it does not have a *Rt-object factory* module, but it must control its positioning in space which is accomplished by the *spatial processor.*

The basic differences between the two classes of processes is that a *run-time process* should be able to deal with user input and the *MHEG object process* is responsible for preparing objects, which may mean that it will decode exchanged objects transforming them into the internal representation structure that will be used by the *run time* processes. The *MHEG object process* is also capable of creating a run time instance from its contents which is a model object.

The categories above also make structural distinctions for:

1. *Multiple objects:* these are the objects that may include or make reference to more that one component. Examples of such objects are *composite and container objects* which usually include several other objects. Actions upon such objects usually broadcast their effects to all components. Run time composite objects do not usually have any perceptible effect, and exist simply to maintain the structure of the document.

2. *Simple objects:* these are objects with only one component, for example, a piece of graphics or a video sequence. Run time instances of such objects are usually perceivable.

5.4.1 Processing unit

In this section, we discuss processes which support run time objects. The overall structure for all MHEG objects is similar, with differences only in the possible actions and status.

The process unit within the system is a "monomedium composite". For example, a composite with content objects of two media (a text and a picture as in figure 5.12) will have at least three open windows. One window will be the composite object itself

(an invisible window) and will handle global information for the components; and the other two windows will be responsible for rendering the content information.

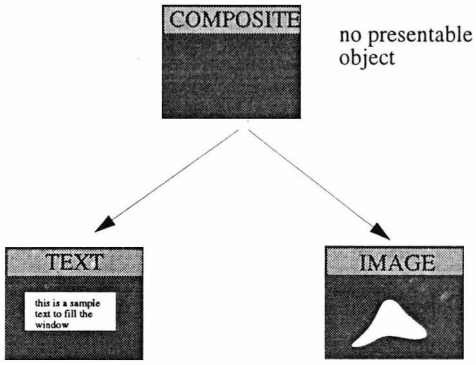

Figure 5.12: A composite with two media components

A window will be *visible* when it is rendering contents data or *invisible* when it is used for control only. Each visible window (a leaf in the tree structure) contains at least one presentable (a run time object with e "visible" effect) and one content object. A window is also responsible for scrolling, and for handling hypermedia *anchors*. The window may also handle several link objects.

The number of windows may also be dependent on specific hardware; for example, if the machine has hardware to play only one video sequence a time, there can be only one instance of the process (window) that displays video. On the other hand, there could be several instances of a "text" process as there are no hardware restrictions on the number of text objects being presented at a certain moment. All interaction among components is made via the *Registry*, as described in section 5.5.

A (reasonably) complex composite MHEG object can be seen in figure 5.13. The composite presented will render six *content objects* (the leaves in the tree). It is subdivided into five presentables (sub-)composites (figure 5.14). The hollow leaves indicate a reference to another composite. Solid leaves represent presentables with reference to a content object. Each sub-composite can be authored individually and assembled to build up the whole composite. Within each sub-composite the components are identified by sequential numbers starting from 1. The sequence of numbers in the box

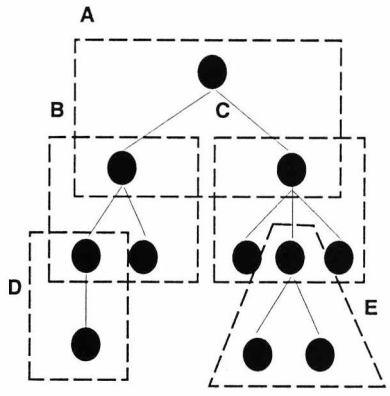

Figure 5.13: A complex composite

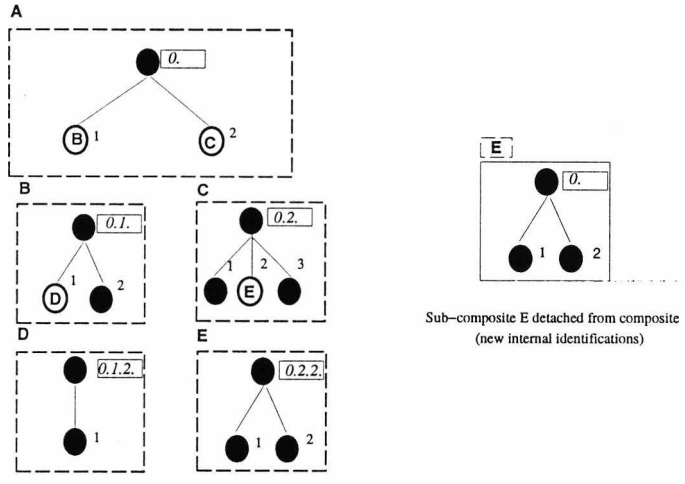

Figure 5.14: (Sub-)composites of figure 5.13

at the root of each sub-composite indicate its identification within the whole composite. For example the right leaf of sub-composite B will have identification *0.1.2* within the composite.

Alternatively, the sub-composites may have been exchanged in individual composites (no inclusion used), in which case the identification of the same object as above would be accomplished by using its external identifier and the component index (*2*).

The internal references in the figures described are only valid in the scope of the full composite of figure 5.13. If, say, sub-composite E is detached from the whole composite, its internal references would be changed as in figure 5.14 (right).

5.4.2 The decoder

The decoder is the component that transforms an object from the interchange format to the internal format. It may receive the encoded data from outside the process via the *dispatcher* or it may load it directly (eg. from a local disk).

Internally, the decoder communicates with the *scheduler* which breaks objects into components and retrieves their identifications. Objects that are not handled by the process are transferred to the kernel that will forward them to the correct destination.

Figure 5.15 shows an example of a composite object definition. The object has the external identification `comp1.cmp`. It has references to two link objects, one reference to a video object and one reference to a text object.

Figure 5.16 shows the process tree spawned when this composite is presented. At the root we have the composite process itself. This process will decode and create all the other processes, ie. the presentable processes (`msg.txt` and `esda1.vid`). The two links referenced by the composite (`vid1.lnk` and `vid2.lnk`) are decoded within the composite process but they will be processed (as described in section 5.3.1) by the process that contains the trigger condition.

5.4.3 The link processor

The link processor is one of the most important modules in the system. The link processor is made up of three basic components

1. *Status manager* that is responsible for keeping track of all status data for the objects handled by the process. Status information is used for triggering links. The status manager informs the link factory about any change in the internal status for which there is a dependency in a link condition.

2. *Timestone manager* that exists for continuous media. This module checks the media position and uses this information to change `TIMESTONE_STATUS` information.

```
\begin{COMPOSITE}
\Description
-Name: CO-001
-Owner: ESDA
-Date: 9401121137Z
-Comments:  4 contents
\externalid  comp1.cmp
\Composition status:0
\Components:4
\index: 1
\begin{CONTENT}
_Description: -Name: esda1.vid -Date: 9401121134Z
_MHEG ID:80, 114, 111, 116, 95, 51, 52, 49, ; 9
_MHEG classification: Motion JPEG video
_HOOK: Encoding: proprietary    EncodingDescription:
_Data:esda1.vid
_OriginalSize: X=19018 Y=34944 Z=0
_OriginalSpeed: true
\end{CONTENT}
\index: 2
\begin{CONTENT}
_MclId: 2
_Description: -Name: msg.txt -Date: 9401121134Z
_MHEG ID:80, 114, 111, 116, 95, 51, 52, 49, ; 10
_MHEG classification: ascii text
_HOOK: Encoding: ascii       EncodingDescription:
_Data:msg.txt
_OriginalSize: X=21845 Y=36816 Z=0
_OriginalSpeed: false
\end{CONTENT}
\index: 3
\begin{CONTENT}
_Description: -Name: vid1.lnk -Date: 9401121134Z
_MHEG ID:80, 116, 115, 116, 95, 51, 52, 49, ; 9
_MHEG classification: link
_HOOK: Encoding:
_      EncodingDescription:
_Data: vid1.lnk
_OriginalSpeed: false
\end{CONTENT}
\index: 4
\begin{CONTENT}
_Description:  -Name: vid2.lnk  -Date: 9401121134Z
_MHEG ID:80, 118, 118, 108, 181, 136, 52, 49, ; 9
_MHEG classification: link
_HOOK: Encoding:       EncodingDescription:
_Data: vid2.lnk
_OriginalSpeed: false
\end{CONTENT}
\end{COMPOSITE}
```

Figure 5.15: Example of a Composite Object with Four Components

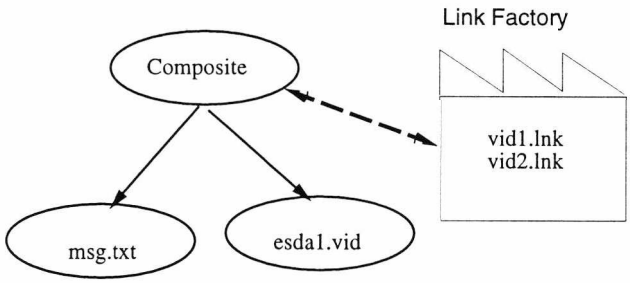

Figure 5.16: Process Tree for Figure 5.15

3. *Action processor* which is the module that schedules and processes an action that has been triggered by a link.

A description of the main components of an MHEG link is presented in section A.3. An example of a link is shown is figure 5.17. The link described in the figure would be triggered when object `demo1/bird.bmp#1` reaches or passes timestone 2 (lines 6 to 11) and object `demo1/creative.bmp#3` reaches timestone 1 (lines 14 to 19). The "effect" of the link is performed by one action object only (lines 27 to 32) targeted at object `demo1/bird.bmp#1` and the only elementary action specified is `@setposition 100 0`. If it was desired to have a set of actions happening in parallel, more than one "serial targeted actions" (`{sertargactions}`) group should be specified within the "parallel targeted actions" (`{partargactions}`).

5.4.4 The spatial processor

This is the module that keeps track of window and media positions and size. Spatial position are used within the process itself and to provide reference positioning for child processes in the MHEG generic space.

The *Spatial Processor*, in summary, is the module responsible for mapping measures from *Generic Temporal Units* and *Generic Spatial Units* to physical values.

```
01 \begin{link}
02   \mhegid demo3/d3_13.lnk
03     \begin{linkcond}
04      \rule AND
05        \begin{constraint}
06          @source      demo1/bird.bmp#1
07          @trigvalue   timestone_status
08          @beforevalue 0
09          @wasrel      *
10          @becomesrel ge
11          @aftervalue  2
12        \end{constraint}
13        \begin{constraint}
14          @source      demo1/creative.bmp#3
15          @trigvalue   timestone_status
16          @beforevalue 0
17          @wasrel      *
18          @becomesrel eq
19          @aftervalue  1
20        \end{constraint}
21     \end{linkcond}
22     \begin{partargactions}
23       \begin{sertargactions}
24         \targets demo1/bird.bmp#1
25         \performances 1
26         \transition   0
27         \begin{actionobj}
28           \mhegid ac_id_1
29           \begin{serialgroup}
30              @setposition 100 0
31           \end{serialgroup}
32         \end{actionobj}
33       \end{sertargactions}
34     \end{partargactions}
35 \end{link}
```

Figure 5.17: Example of a Link Object

5.4.5 The media specific processor

This is the part that should be individually written for each medium introduced. It handles all processing that is unique to the medium and in some cases it may be dependent on specific hardware. Status information specific to the media being handled is also updated here.

For example, in the case of video, there is a dedicated motion JPEG card. The media specific processor handles all communication between the card and the internal process. When the displaying window is resized or moved, in addition to updating the *Windows* window that provides the frame for displaying the images, the actual video images should be moved by informing the video card of the changes.

5.5 Exchanging messages between processes

Amongst the types of interprocess communication facilities provided by the MS-Windows 3.1 environment (as discussed in section 4.5), we use:

1. *Windows messages:* to broadcast timer messages within a processor. Windows messages are also used for all user input and interaction with the interface such as typing, moving and resizing windows, etc.

2. *DLLs:* are used for all inter process communication within one host; all its processes share memory within a DLL.

   All communication uses shared memory in the *Registry* DLL. A process, wishing to post a message to another process, leaves the message at the *post office* maintained by the the registry. Each process must check at the post office for incoming messages during the execution of its main loop.

   As we perform most interprocess communication via the DLLs, we have the flexibility to implement a scheduling algorithm that, within limits, provides a seamless presentation.

DLLs are also used for communicating with some dedicated hardware that is released with dynamic libraries for system programming (e.g. motion JPEG card).

3. *DDE:* is used in two cases:

   - Integrating existing applications in the environment. Existing DDE aware applications are integrated by the system as a *media specific* module in figure 5.11. In this case, the actions exchanged between the link processor and the *media specific* part is performed by a DDE conversation. An example is the integration of the module *Text-To-Speech* (a program that converts ASCII text to speech using a sound blaster card) [Labs, 1994] that provides for text being *spoken* rather than *displayed.*
   - Communication involving two hosts happens at kernel level and is object transparent. When a process requests the kernel to transfer a message, the registry is responsible for the delivery, and in the case where both ends are in the same host, shared memory in the DLL will be used. If the parts involved in the communication are being handled by different hosts, a net DDE link is established. Currently, the implementation handles only objects in the same host.

5.6 System orchestration

In order to tune performance to achieve defined bounds, we define two levels where resource usage, in particular CPU time, is shared between processes:

- *Process selection level:* in the common case, there will be several processes running on one host. The run time process must define which process is to have CPU control.

- *Activity selection level:* once the process is selected, we must define which of the pending activities is to be performed. It must be kept in mind that in a non preemptive operating system, the process must not take CPU control for a time long enough to be noticed by the user.

In the following sections, we discuss the two levels of the orchestration scheme.

5.6.1 High level orchestration: process selection

As it has been said, the *clock* broadcasts timer messages for all registered processes. In the higher level, priorities can be defined when selecting the next window to receive the timer message. Higher priority can be given to a process that is expected to answer a request for status information that may be used to trigger a link. On the other hand, a process that is presenting a video using a dedicated hardware may have a lower priority when no communication is pending to that window because the presentation will proceed with no interruption even if CPU time is not scheduled to the process (except, of course, if some event such as "end of media reached" happens).

Although several scheduling policies were tested (see Chapter 6), no single policy is recommended as the requirements for clock granularity, skew and acceptable action delays are application dependent and should be defined by the user.

5.6.2 Process level orchestration: The main loop

Processing within a window happens around a main loop that contains calls to routines performed at each time tick. This is the place where the process must schedule itself in order to optimise resources and provide for a smooth presentation. Under a non preemptive operating system, the process must be as *nice* as possible by holding the CPU for the shortest amount of time.

The activities performed by the main loop can be divided into two modules or groups:

1. Generic activities: those that all processes should implement, such as positioning and sizing windows, transferring objects, etc.; and

2. Media specific activities: those where processing depends on the media being rendered, such as obtaining the position within the media, etc. These activities may be complementary to the generic ones. For example, when a window is moved in a video process, it is necessary to update both the position of the window itself and the position where the video is being presented.

Generic activities

This module:

- *Retrieves incoming messages and objects* from the kernel. Usually these messages are queries about the status of objects whose results are used to trigger links within the window. The process also checks links whose trigger conditions are within the window but which were decoded by another process. The process may also receive an action to be performed internally which was triggered by an outside process. An example of a query is "what is your preparation status?", which can be `READY` or `NOTREADY`;

- *Transfers objects and messages to the kernel:* this involves actions complementaries to the previous ones. The process transfers to the kernel queries about the status of remote objects, or actions to be handled by other windows. An example of a message that a process has to transfer to the kernel is the answer to the query about preparation status given in the previous example.

- *Update link data:* link information is handled by the link processor and involves several activities:

  - Check conditions that may trigger a link, such as change of any internal status variable (e.g. timestone status, preparation status, etc.);

- For all triggered links being handled by the process, the links processor must schedule an action to be performed in the time tick. This involves selecting a link, and within the selected link scheduling the action to be performed according to the definition from MHEG.

  This step is required because the system is implemented under a non pre-emptive operating system, and all processes must avoid taking over the CPU for too long.

Media specific activities

This module:

- *Updates the current time* this applies to continuous media: the current position within the medium must be either estimated or determined. When the cost of determining the correct positioning within the medium is high, it is usually less prejudicial to estimate the position. This happens, for instance, with a JPEG motion card where a query such as *"what is your position?"* takes over the CPU for approximately half a second, disrupting the whole presentation process if it is performed frequently;
- *Update presentation status* such as timestones being achieved, end of media reached, etc.
- *Process actions* that are specific to the medium, such as setting speed, fading, setting volume, etc.

5.7 The Link Factory

The processing of links can be divided into three steps:

1. *Link decoding* that involves the decoding and transformation of the exchanged object into the internal representation;

2. *Link triggering* that involves the period when the link is waiting for events that will cause it to fire, and

3. *Link effect* which is the processing of actions triggered by the link.

5.7.1 Link decoding

The request for decoding a link happens when a composite or a container object containing the link is prepared. The link factory decodes and stores un-triggered links and it also maintains the triggering status of the link.

In the process of decoding a link, the link factory posts messages in the registry to all processes upon which the link condition tree depends. The processes will then be responsible for informing the link factory of changes in conditions. The link factory does not need to poll processes in order to evaluate a link and it can assume that the status information stored is valid up to the timing resolution that the run time system is maintaining.

5.7.2 Link Triggering

MHEG links are event triggered with a general structure as shown in figure 5.18.

During the phase prior to its triggering, a link will be waiting for the events that will satisfy the link condition. The link effect is not processed before the link is triggered.

The link must wait for status changes from all objects that are part of its link condition. One of the problems that must be taken into account is the cost of checking all conditions that are part of the link condition. Polling all objects to check for status change would be very expensive in terms of resources. The solution proposed uses the fact that the registry keeps a list of pending messages for every registered object. When a link is decoded, and therefore created, messages are posted in the registry for all objects referred to in the link condition. The messages are of the form `destination` (the internal identification of the link that expects a response) and `condition` that

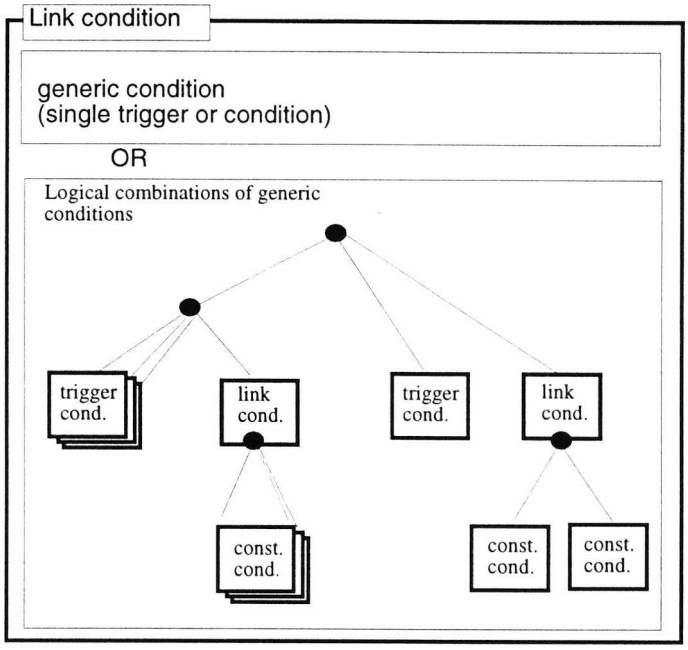

Figure 5.18: Link Condition Tree

defines the condition to be satisfied.

All processes check with the registry about events that should be reported during their processing loop, as described in section 5.6.2. When a *pending request* evaluates to true, it will cause an update in the condition tree. The process that generated the event will use part of its CPU share to update the link condition tree. If it happens to trigger the link, the link effect should be performed as described below.

A link must be fired as many times as its link condition evaluates to true, so the whole process is restarted. Therefore, a link, once created, will be maintained by the link factory until an explicit message for its destruction is issued.

Figure 5.19 makes the idea clearer. In the figure we see the *Link Factory*, the *Registry* and processes *video1* and *video2*. Inside the link factory we have link object *Link1* whose trigger condition is satisfied if *video1* reaches position 50 seconds or *video2* reaches frame 325. The Registry keeps communication queues for all objects registered. One of the queues kept maintains requests for status or attributes. Therefore, in the queue kept for *video1* there is a request to inform object *link1* when condition `position = 50` is true. For object *video2*, there is a request to inform object *link1*

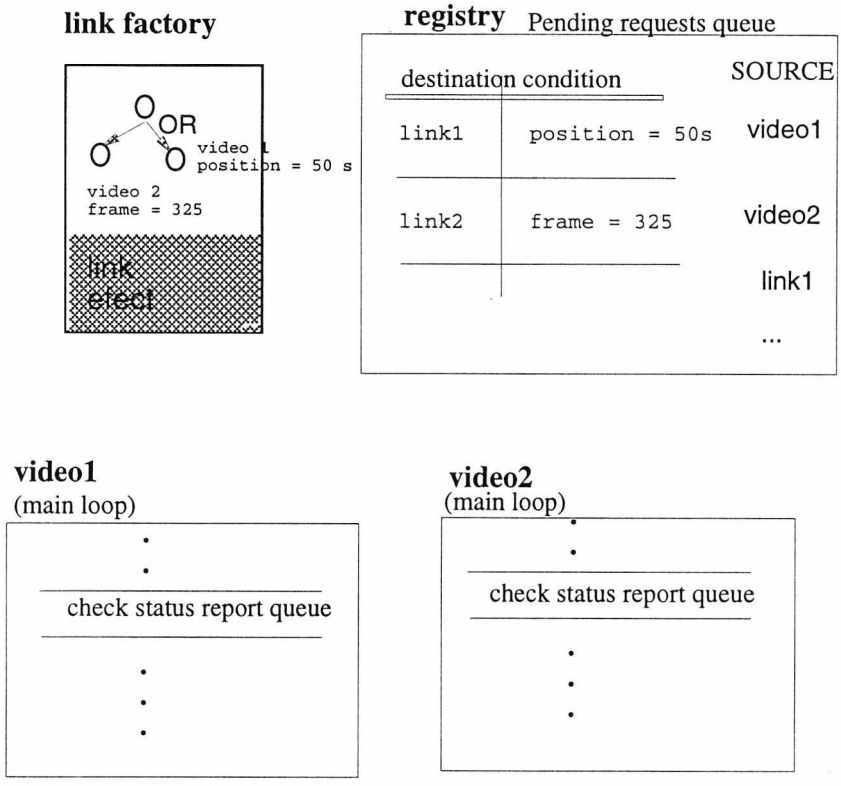

Figure 5.19: Link Processing Overview

when it passes frame 325. The video objects (*video1* and *video2*) check the registry for pending requests during their main loops. When a request is satisfied, the tree of conditions for the link in the Link Factory is immediately updated and the link may be triggered, in which case a new *Link Process* is started.

5.7.3 Link effect

After the link is triggered the link effect is processed. From the point of view of the processes handling the link effect, once triggering has occurred, the link condition is no longer important. The link can however be triggered again because its processing within the link factory remains unchanged, and processes will still test the link conditions.

The solution proposed to implement the processing of the link effect is to create an independent process that will perform the actions. A link process is not *visible* and it is a process whose location can be made completely transparent to the author or user. In

its main loop, a link process will deliver actions to the correct object using the registry. In this sense, as far as the link process is concerned, it works like a remote control to other objects that are affected by the link.

5.8 Timestamping messages

All communication between objects happens via the Registry. Messages are timestamped and the destination process can order all messages it receives by using its own real-time clock, provided by the Clock (section 5.3.3) module. The implementation uses the timer services provided by Windows multimedia extensions which provide a time resolution of one millisecond to timestamp messages. However, the timer resolution provided by the Clock process is still 55 ms.

5.9 Actions

Action objects have the structure shown in figure 5.20. Actions objects are part of the link effect and are executed when the link is triggered.

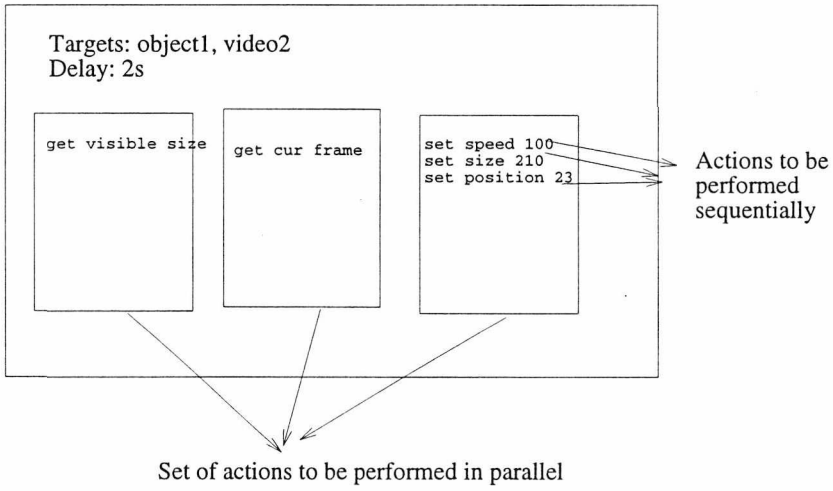

Figure 5.20: Action Object

The action object has a set of targets, defines a delay before its execution, and has

a list of actions that should happen in parallel. Within each list of parallel actions there is a list of actions that should happen sequentially.

The internal messaging system stores messages as sequences of characters and it is up to the destination process to interpret the action. Several strategies for optimising the communication of actions can be implemented but the best results were obtained when the link process transferred the entire set of actions targeted to an object as one chunk. The destination process is then in charge of using its own resources to synchronise the processing of actions.

5.10 Final remarks

This chapter has provided an overview of the proposed architecture and of the prototype implementation. Some performance measurements and a discussion on how they were used to tune the system are provided in the next chapter.

The system proposed is modular and can be extended with little difficulty, but it would benefit from using a preemptive operating system as discussed below.

5.10.1 Support for extensions

The system allows extensions to be made at several levels:

- new actions can be added because the base system does not interpret the actions;
- the interprocess communication infrastructure can be updated to more efficient mechanisms, such as RPC, since support for communication is centralised;
- existing applications that support some means of external communication, such as DDE, can be integrated by using a process to provide *remote control* for them;
- a reconfigurable system based on filters/preprocessing of objects such as in [Hill *et al.*, 1992] could be provided, since these features can be added by changing the

behaviour of the MHEG engine module in the kernel (section 5.3.4), with little effect on media dependent processes;

5.10.2 Considerations for a Preemptive OS

The implementation would greatly benefit from a preemptive operating system since much of the burden currently placed on the programmer would then be left to the underlying operating system:

- scheduling algorithms are simplified as there is no need to have *well behaved* applications. The operating system guarantees a "smooth" presentation by not allowing any process to use all the CPU time.
- the run time system depends on slow actions (painting a 1024x768 images with 256 colours can take almost 4 s). Here again, if the programmer is not disciplined, a simple window repaint may have a huge impact on overall performance.

Chapter 6

Performance measurements

In this chapter, we present and discuss some performance measurements and how these measurements were used to tune the system. Limitations imposed by the MS-Windows 3.1 environment are also discussed.

Initially, some performance graphs obtained for a small document are shown, and then figures are presented for various kinds of extreme load on the system.

Identifying processing bottlenecks is especially important in an application running under a non preemptive operating system as one badly behaved routine may cause the whole system to come to a halt.

6.1 Technique used

The measurements were obtained by using a DLL to store the timing data in memory and to transfer it to disk at the end of the run. The raw data was then processed off line. The client processes transferred strings marking the points at which the measurements have to be taken; each string is time stamped in the DLL for processing.

The DLL, when loaded, allocates a large enough amount of memory to store all data for that run. The first version of the DLL dynamically allocated memory for the strings but, as the time taken by Windows to allocate memory proved not to be constant,

it was decided to allocate all memory before the beginning of the presentation. In a typical run, the overhead introduced by the data collection was reduced by around 40 times by pre-allocating memory. The measured times dropped from over 5 ms for each measurement string to about 8 strings/ms.

All time stamps were obtained using the function `timeGetTime` in the multimedia extension DLL (`mmsystem.dll`) which provides a resolution of 1 ms.

6.2 A typical presentation

The first measurements were taken from a small complete multimedia presentation. A screen snapshot of this presentation is shown in figure 6.1. In summary, the presentation consisted of four bitmap images (`bird, creative, house and japan`) and an AVI video sequence `ca_world` from a CD-ROM. Figure 6.2 shows a time line with the main events after the run-time object to present `ca_world` is created. The images are displayed when links are triggered by timestones set in the video sequence. A timestone was also used to trigger links to destroy each run-time object. In figure 6.2, the last run time objects (`japan` and `ca_world`) are destroyed at 60 s.

6.2.1 Performance considerations

The measurements taken were centered around the execution of the run-time object main loop. As described in detail in section 5.6.2, the main activities performed during the execution of the main loop are:

- Retrieve incoming messages from the registry;
- Update internal status variables;
- Perform actions targeted at the window;
- Check whether any change in the current status will trigger a link, and use the link factory to trigger the link, if so.

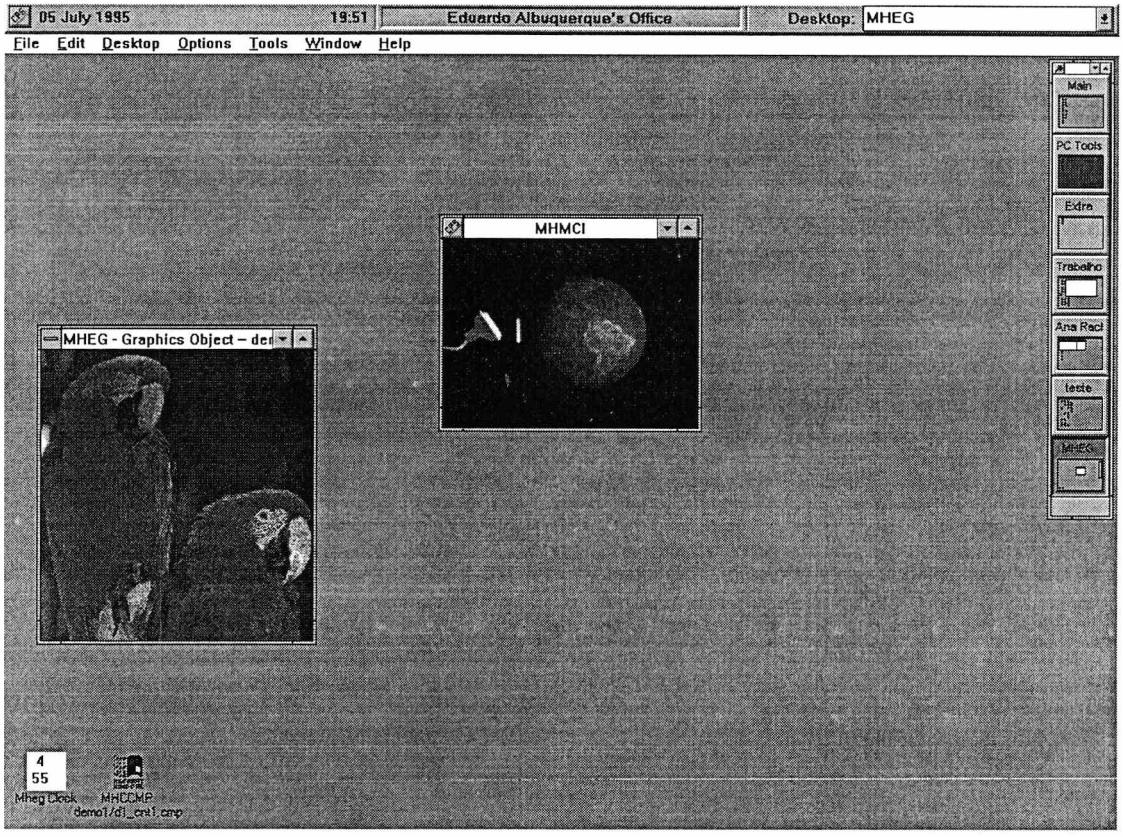

Figure 6.1: A Snapshot of a presentation

Repainting may be requested outside the direct control of the engine if, for example, a window is uncovered by user interaction. The time taken to repaint the window was also measured. During the repaint process, only the window performing the repaint will have CPU control, making this activity critical.

The measurements were taken on two Intel 486 based machines:

- *486-66:* clock at 66MHZ, with a PCI S3-864 (1 MB DRAM) graphics card;
- *486-33:* clock at 33MHZ, with a Western Digital (512 KB DRAM) graphics card (vesa local bus). This machine is also equipped with a Video Logic motion JPEG card.

Both machines are also equipped with a Panasonic 563 double speed CD-ROM drive and a Creative Lab Sound Blaster 16 Value audio card.

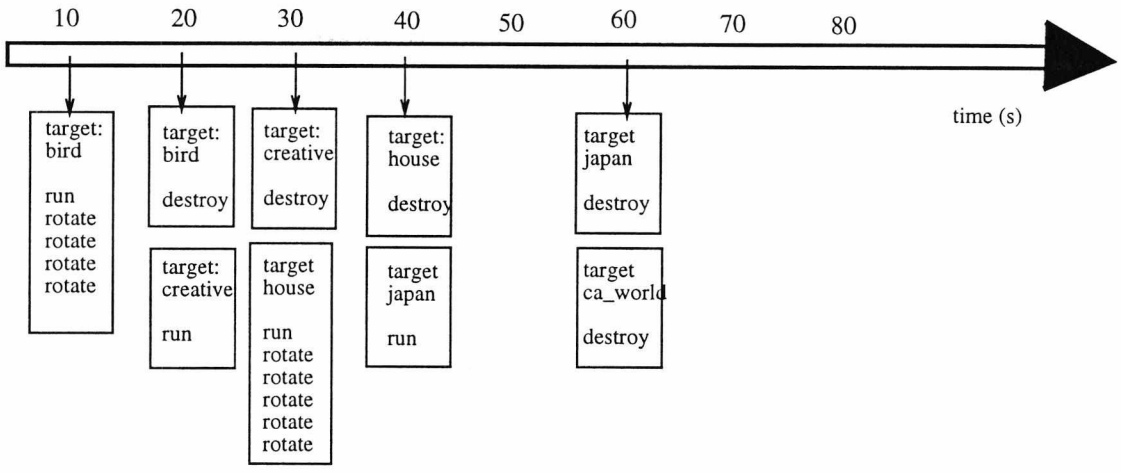

Figure 6.2: Timestones set in `ca_world` run-time object

In order to determine the frequency at which processes should release CPU control, the measurements were organised as follows:

- *Policy 1:* in this case, the process performs a single activity in the main loop, then releases control, providing a *nice* behaviour by holding the CPU for a very small time slice. As the window that releases the control can receive a new clock message before finishing the previous main loop, it is necessary to make the main loop code protected, i.e. it will be restarted only after the previous cycle has finished.

- *Policy 2:* each process performs all activities in the main loop before releasing control. Only one action in each category is performed: e.g. it will retrieve at most one message from registry; if there is an action object to be executed, only one elementary action will be performed. However, the process will check all link conditions that depend on status information updated since the previous time it had control.

The items in the tables of measurements are as follows:

- *Number of times main loop was executed:* the number of times the main loop was fully executed: in the case of *policy 1*, this means the number of times the main

loop is completely executed regardless of the number of times it was interrupted to yield control to other processes.

- *Average time slice:* average CPU time used in each slice. To provide a smooth presentation, this value should be as short as possible. The higher this value the worse the presentation will be as only one process can take control of the CPU at a time. The optimum value varies depending on the kinds of media being presented (competing for resources).

- *Longest time slice:* the longest time for which the process took over the CPU. For all graphics object the longest slice happened the first time the window had to be painted. This value should also be minimised.

- *Standard deviation (time slice):* the standard deviation including the longest time slice.

- *Average time slice (ignoring the longest):* this measure gives the average time slice the process took ignoring the first time the window was painted (the longest one).

- *Standard deviation (ignoring the longest)* is also the value ignoring the slice in which the first paint happened.

- *Longest time between slices:* the longest gap between slices. The longer the distance between the slices, the later internal status will be updated, causing synchronisation problems.

- *Number of times the window was re-painted:* the window is painted when it is started; in the case of images that were rotated, each rotation requires a re-paint.

- *Average time used painting:* the average time the process required to perform a full repaint of the window.

Tables 6.1 and 6.2 represents the measurements obtained using policy 1. It shows that the average time slice used by the process when there are no activities to be performed is very low, typically less than 1 ms per cycle (e.g. the case of the `creative` object which is painted only once and is not the target of any action). Most of the time used by this process was to repaint the window.

Tables 6.3 and 6.4 represent the measurements obtained using policy 2. Here again, the time used by a process when there is no activity is less than 1 ms; and most of the time was used repainting the window.

Comparing tables 6.3 with 6.1 and 6.4 with 6.2, it is clear that interrupting the processes frequently brings a penalty to the overall performance. Each process on average takes around 10% longer to perform a full loop. In the whole presentation, the total number of times the main loop was fully executed is also reduced, which is not desirable. On the other hand, the tables above do not give a full picture of what happens since although yielding control more frequently means the time to perform the whole set of actions each time slice is longer, the processes are interrupted more frequently. It can be seen that raising the frequency with which the processes release control while performing the activities within the main loop does not lead to an improvement in the presentations "smoothness" as the longest time measured between the time slices is not reduced significantly because it is primarily dependent on the execution of heavy (or slow) actions. Therefore, all processes should try to break up heavy processing activities in order not to affect other processes.

The measurements obtained with a *486-33* with a slow video card (tables 6.2 and 6.4) also shows that the response times obtained were not acceptable for a real application, as the figures were above the desirable maximum delay which is in the order of 0.2 s [Shneiderman, 1984].

Item Measured	Object				
	ca_world	bird	creative	house	japan
Times main loop had control	490	250	337	382	574
Times main loop was executed	188	84	119	135	192
Time slice:					
Average	5.96	1.88	0.48	2.76	0.34
Longest	996	190	144	147	151
Standard deviation	43.62	14.51	7.84	14.38	6.30
Ignoring longest slice:					
Average time slice	0.42	0.08	0.05	2.38	0.08
Standard deviation	0.60	0.27	0.22	12.33	6.30
Time between slices:					
Average	127.62	94.53	96.58	112.55	108.91
Standard deviation	159.64	127.26	119.75	144.80	135.20
Longest	995	771	698	799	837
Painting:					
Number of Times	2	6	2	15	2
Average	1	149.45	128.90	137.93	124.98

Table 6.1: Measurements with processes yielding control within the main loop (times in ms)— policy 1 *nice behaviour/ (486-66)*

Item Measured	Object				
	ca_world	bird	creative	house	japan
Times main loop had control	223	112	142	166	334
Times main loop was executed	114	39	49	56	113
Time slice:					
Average	5.83	8.62	3.20	7.20	1.51
Longest	1136	434	418	451	453
Standard deviation	76.05	47.32	35.10	40.49	24.81
Ignoring longest slice:					
Average time slice	0.74	4.78	0.26	4.51	0.15
Standard deviation	2.08	24.38	1.70	20.94	1.23
Time between slices:					
Average	410.93	237.09	239.20	282.02	270.23
Standard deviation	362.55	300.54	286.82	343.02	287.45
Longest	2272	1726	1728	1823	1933
Painting:					
Number of Times	2	6	2	8	2
Average	4	434.00	362.52	412.23	409.33

Table 6.2: Measurements with processes yielding control within the main loop (times in ms)— policy 1 *nice behaviour/ (486-33)*

Item Measured	Object				
	ca_world	bird	creative	house	japan
Times main loop was executed	195	87	121	137	194
Time slice:					
Average	5.87	5.31	1.26	7.21	0.89
Longest	941	171	137	149	143
Standard deviation	67.31	23.51	12.44	22.19	10.26
Ignoring longest slice:					
Average time slice	1.05	3.30	0.13	6.16	0.15
Standard deviation	216.52	14.21	0.34	18.54	0.36
Time between slices:					
Average	320.53	262.89	269.43	312.63	321.43
Standard deviation	216.52	204.49	189.44	224.11	206.63
Longest	1135	935	935	950	983
Painting:					
Number of Times	2	6	2	15	2
Average	1	142.33	129.00	134.53	134.00

Table 6.3: Measurements with processes performing all activities in the main loop before yielding control (times in ms) — policy 2 (*486-66*)

Item Measured	Object				
	ca_world	bird	creative	house	japan
Times main loop was executed	114	39	49	56	113
Time slice:					
Average	11.90	23.28	8.41	19.63	4.41
Longest	1167	434	396	459	472
Standard deviation	109.15	76.61	56.53	68.12	44.38
Ignoring longest slice:					
Average time slice	1.68	12.47	0.33	11.63	0.23
Standard deviation	1.19	36.23	0.47	32.64	0.42
Time between slices:					
Average	786.35	663.82	707.73	827.78	789.05
Standard deviation	470.15	518.49	472.45	557.19	405.28
Longest	2679	1995	2045	2469	2726
Painting:					
Number of Times	2	6	2	8	2
Average	3	414.17	359.50	414.63	412.50

Table 6.4: Measurements with processes performing all activities in the main loop before yielding control (times in ms) — policy 2 (*486-33*)

Timestone Reached	Policy 1 (ms)	Policy 2 (ms)
1	168	98
2	132	65
3	198	198
4	198	269
5	237	163

Table 6.5: Average errors setting timestones

Table 6.5 shows the average error in the actual time of occurrence of the timestones set for the video process. The longest error found (269 ms) happened when a window was painted (taking 145 ms) between two time slices used by the `japan` process. One way of compensating for such errors is to establish an error margin for comparing times. This value can be dynamically adjusted depending on the system load and in the above example, a value of 200 ms is enough to guarantee that most timestones will be triggered in time.

One conclusion that can be drawn from the above tables is that although processes must be well behaved by not holding CPU control for a long time, they should also avoid doing too few activities in one cycle as the overhead imposed by the operating system for task switching will then affect the overall performance. The process must, however, avoid performing more than one slow activity (e.g. repainting the window) in the same cycle as the longest time between two CPU slices is more dependent on the execution of *slow* actions (eg. painting) than on the frequency with which control is released between *fast actions*. In the case of pictures, several alternatives for reducing the load can be used [Bulterman, 1993] such as updating parts of the image at a time, using reduced sized images, or even displaying a text description of the image.

Table 6.6 shows the average time the window (process) that will process the link effect takes to retrieve a triggered link from the link factory. The times measured range from 135 ms to 529 ms. A large proportion of this time is used creating the window that will handle the triggered link: the values measured range from 90 ms to 430 ms. This

Timestone Reached	Measure 1 (ms)	Measure 2 (ms)
1	390	300
2	135	195
3	267	465
4	529	479
5	320	323

Table 6.6: Delay to retrieve a link triggered (creating *triggered link* process)

Timestone Reached	Measure 1 (ms)	Measure 2 (ms)
1	151	98
2	83	28
3	57	157
4	63	39
5	8	10

Table 6.7: Delay to retrieve a link triggered (triggered link process not created)

values shows that the process of creating a window in Windows 3.1 is too heavy for handling events that require a fast response time. In addition, the creation of an extra process uses up the scarce resources, as discussed in section 6.5.

Table 6.7 reflects the times obtained when the processing of the link effect is within the kernel and therefore avoids the creation of a process to handle it. The changes were implemented in the *link factory*: when a link is triggered, instead of starting the process to handle the effect, the link factory itself posts the effect to the destination objects. The changes made, as a prototype, do not include all the functionality maintained by the independent process, such as the ability to interrupt the processing of the link effect. However, the times measured were around 200 ms faster than in the previous case. The improvement obtained is due only to avoiding the overhead imposed by Windows 3.1 to start a process up. The disadvantage of the approach is that the processing of all triggered links is kept within the link factory preventing distribution.

6.3 The timer

The limitation imposed by MS-Windows 3.1 on the use of timers was discussed in Chapter 5. Although the timer provides a maximum resolution of approximately 55 ms (54.925 ms), in practice it was observed that the behaviour of the clock is not reliable. Figures 6.3 and 6.4 show the measurements obtained for a process running a timer only. In the first case, the shell (similar to a window manager under X) used was the *Central Point Software (CPS)* shell and in the second the standard *Program manager.* Although the CPS shell interferes with the overall behaviour of the clock, in both cases, one clock tick is lost regularly.

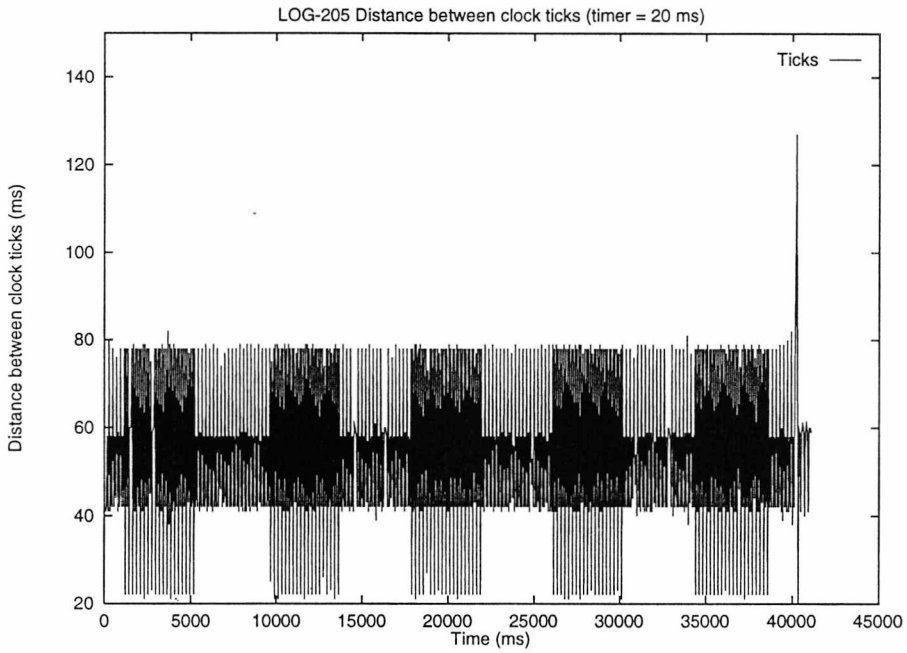

Figure 6.3: Behaviour of Windows timer using CPS shell

The behaviour observed shows that although it would be expected that a process would be activated at every 55 ms (provided all processes are *well behaved*), in practice that time ranges from around 20 ms to 80 ms. Values less than 55 ms are only obtained if the previous tick was late, because Windows guarantees that not more than one timer message is queued at any moment.

The first design to deal with links let the triggered link process schedule all action

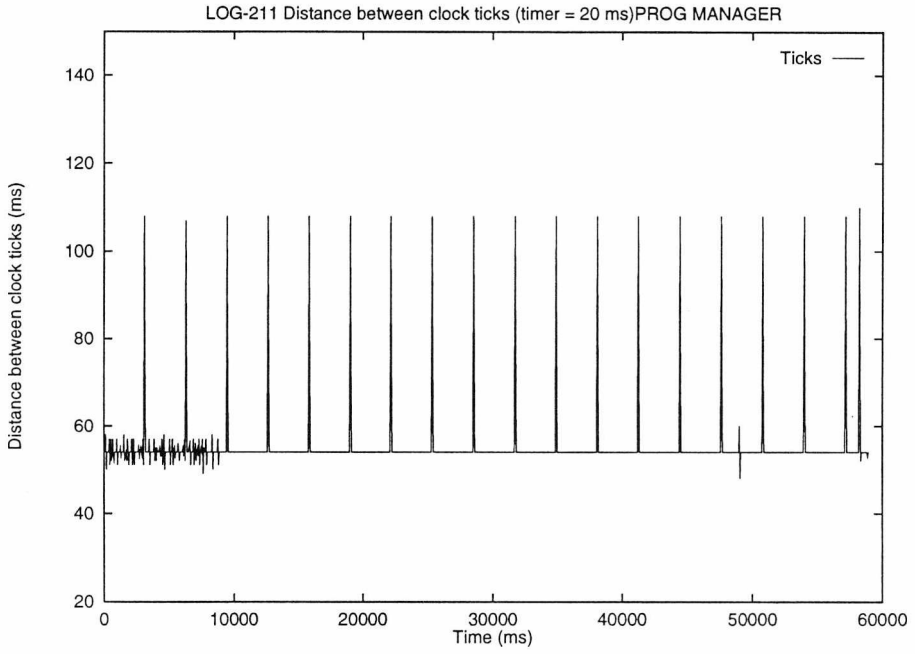

Figure 6.4: Behaviour of Windows timer using Program Manager shell

processing. The client processes then needed only to process elementary actions which have a much simpler structure. However, the limitation imposed by the timer resolution makes this approach infeasible as actions that should happen in sequence or in parallel would have at least a delay of 55 ms. As implemented, the destination process of an action object is responsible for its own internal scheduling to process the elementary actions. Although the process becomes more complex, the response times are greatly improved. Once a link is triggered, the process handling it is basically responsible for informing destination processes that the link effect has been canceled. It can, therefore, have a lower scheduling priority in the system kernel. Figure 6.5 illustrates the sequence in which a link is triggered:

- Once the link is decoded, it is stored and handled by the link factory;
- processes that contain conditions that may trigger the link update the information in the link factory;
- When a link is triggered, a process is created to handle its effect. It maintains only the *effects* part of the link.

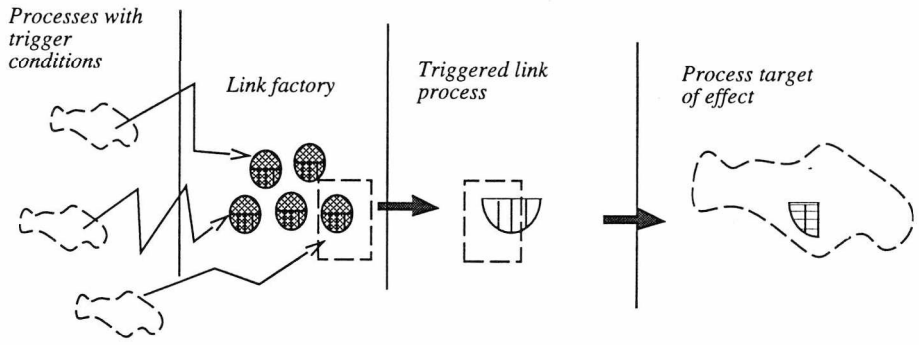

Figure 6.5: Processes involved in dealing with a link

- The triggered link process transfers the action objects to the processes that are the target of the effect. As described in section A.2, the action object may hold several elementary actions that should be scheduled for processing within the destination process.

6.4 Effect of continuous media

The presentation of continuous media such as video or audio causes a considerable load on the system. This load is high either when a sequence is played using Windows MCI extensions or by using the dedicated motion JPEG card.

In order to evaluate the overload imposed by a continuous medium, we ran an altered version of the previous document with an added interaction object in the form of a menu that could pause and resume the video sequence.

Figure 6.6 shows the behaviour of the AVI video sequence. The video was stopped at 24,680 ms (resumed at 31,986 ms), 39,200 ms (resumed at 46,170 ms) and at 58,167 ms (resumed at 70,795 ms). The graph shows clearly that when the video was stopped (marked with vertical lines) the distance between slices and the time required to process windows messages was greatly reduced, approaching 55 ms, i.e. once per clock tick.

If we take a closer look at the behaviour of object `bird` (figure 6.7) which was being continuously rotated when the video was stopped for the first time (at 24 680 ms), it can be seen that not only the distance between the slices was reduced but also the time

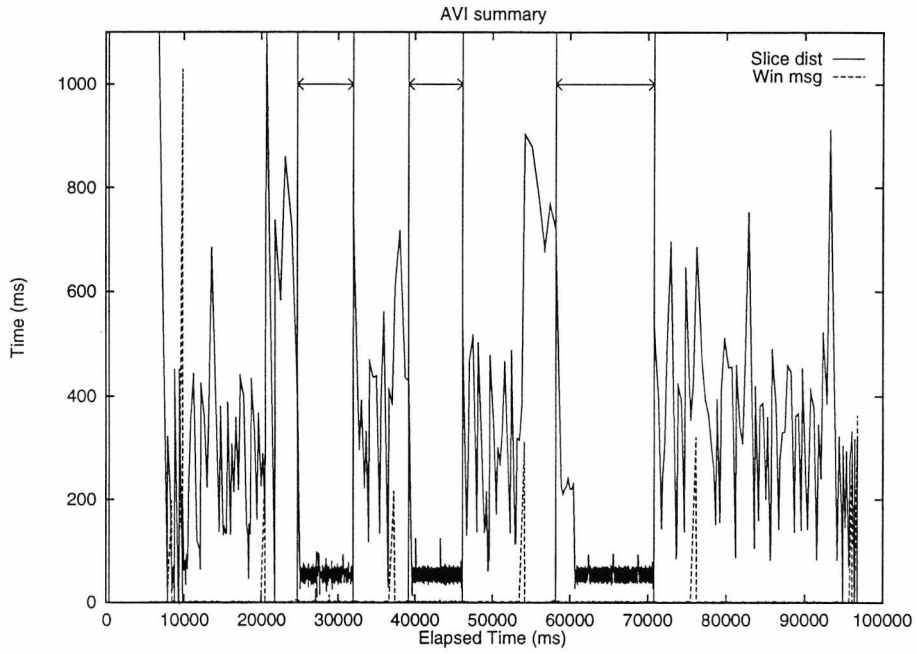

Figure 6.6: Effect of continuous media on a presentation. The arrows indicate regions where the video was paused.

required to process windows messages. Figure 6.8 shows in more detail the system behaviour in the interval between 20 s and 34 s.

The graphs show that just playing one continuous medium adds a heavy overhead to the overall system response, with the measured *distance between slices* suffering at least a 5 fold increase.

6.4.1 Performance of non continuous media only

If the video sequence in the previous example is removed, and more than one graphics object allowed to be processed at the same time the distance between slices as shown in figure 6.9 is obtained. The points with a large distance between slices are in regions where the object is being requested to rotate the image it is displaying, which takes around 140 ms for each image.

From the graph, it can be seen that when there is no painting activity, the distance between the slices is reduced to the clock resolution of 55 ms. If we analyse the region between 10,000 ms and 15,000 ms, we notice that the objects `bird` and `creative` are

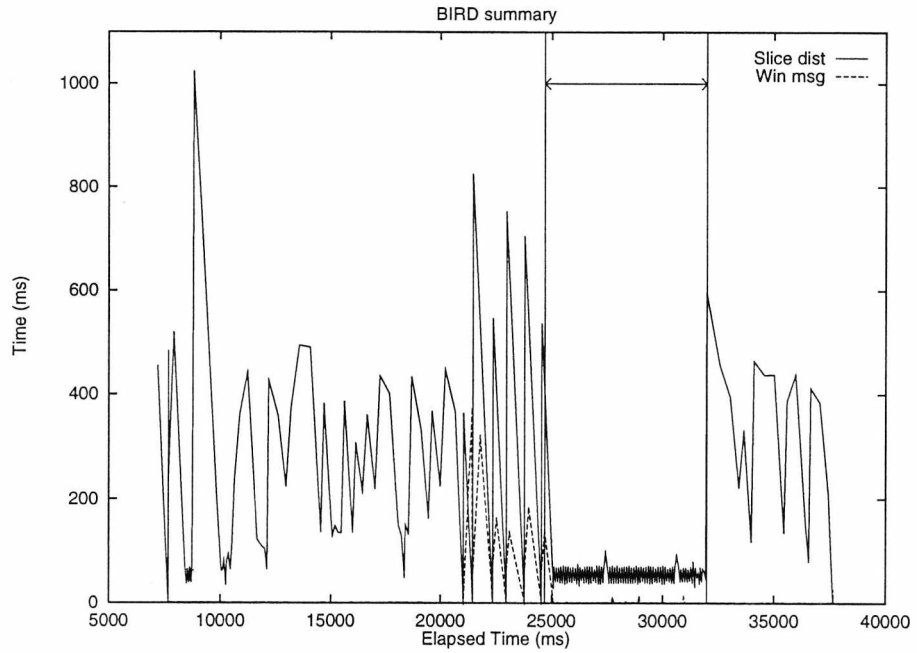

Figure 6.7: Effect of continuous media on a non continuous object. The arrow indicates the region where the video was paused; in the regions where the `win msg` line is not visible, it is near zero.

responsible for most of the activity going on. They are basically repainting the window continuously. Figure 6.10 shows in detail the activity of `bird` between 10,000 ms and 15,000 ms, when it was very active and was, therefore, one of the main causes of the the longer "distance between slices".

The graphs show that non continuous media do not add a background overhead to the system performance as happened with the continuous media. The *distance between slices* is influenced by the time taken by specific activities such as repainting the window.

6.5 Limits on the number of processes

Under MS-Windows 3.1, the number of processes running depends not only on the total amount of available memory, but also on the availability of *system resources* such as "graphics pens", icons, menus, etc. Windows provides only 64 Kbytes of resource memory for all running applications. There is also a limit on the availability of memory under the first 640Kb, as all processes use part of that memory. The final result is

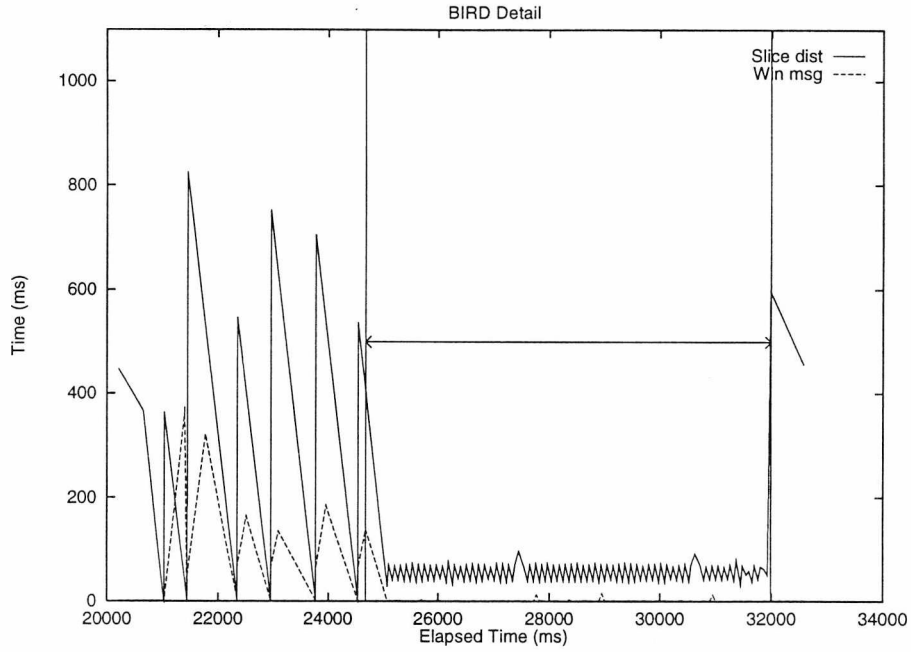

Figure 6.8: Detail of figure 6.7

that in general the environment runs out resource or "memory" even when megabytes of extended memory are still available. This limitation also imposes restrictions on creating a process to handle a triggered link, as many links may be active at the same time. In order to tackle this limitation, there are two versions of the link process: in the first case all link activity is handled by the link factory, even when a link is triggered; and in the second version a process is created to deal with each triggered link.

This problem should not exist in future versions of the operating system which should provide more resources for each process.

On average, it was found that the system runs out of conventional memory when around 45 to 50 tasks are active. To explore the limits on the number of active tasks, a very simple presentation was designed. The presentation contained one model object, a 250x250 bitmap object, and 36 run time instances of it were created, which was enough to cause Windows to swap out memory.

The graph in figure 6.11 shows the behaviour found: The beginning of the graph shows that there was a slight increase in the average distance between the times when the process would have the CPU (`Slice dist`) as the number of instances was

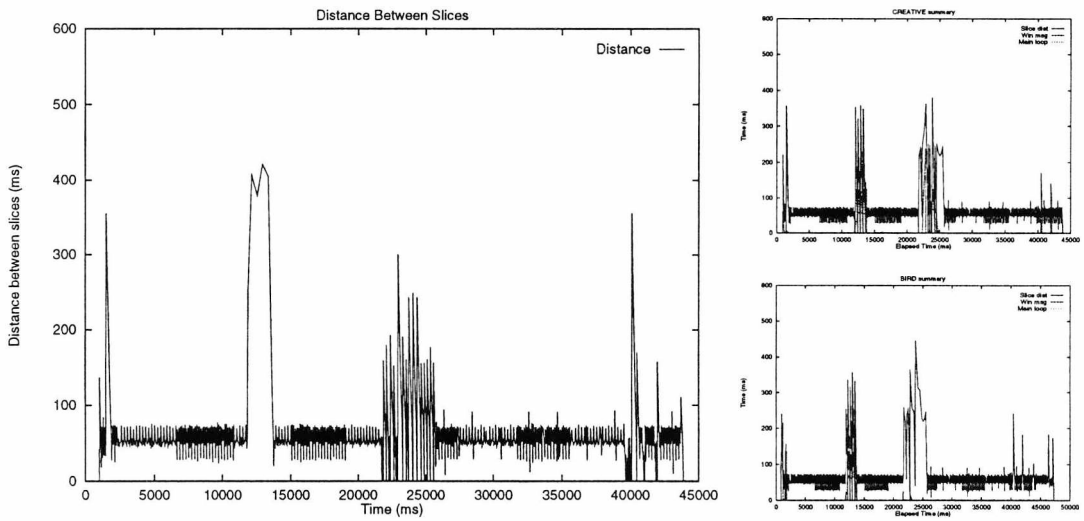

Figure 6.9: Behaviour of a presentation with graphics only

increased. During this period, each instance created represented the creation of a process (a Window), which had to access the disk to retrieve the bitmap. At around 7000 milliseconds, we notice a peak in the graph where for the first time Windows swaps out some memory; then there is a short period with a good response time. Finally, and when the memory usage reaches the limit of the RAM there is a major delay, with a lot of disk activity, when Windows reorganises the memory. The main swap happens just after the creation of the last run time instance. The same behaviour was found when varying the total number of instances which indicates that windows tends to reorganise memory when it becomes relatively unloaded.

Another measurement taken was of the time used by the processes to perform the main loop (see figure 6.12). The time is usually less than 1 ms which proves that the overhead for checking the queues in the registry is very low. The initial time taken to start up the process is not represented in this graphic, and would represent a large peak at the beginning of the graph.

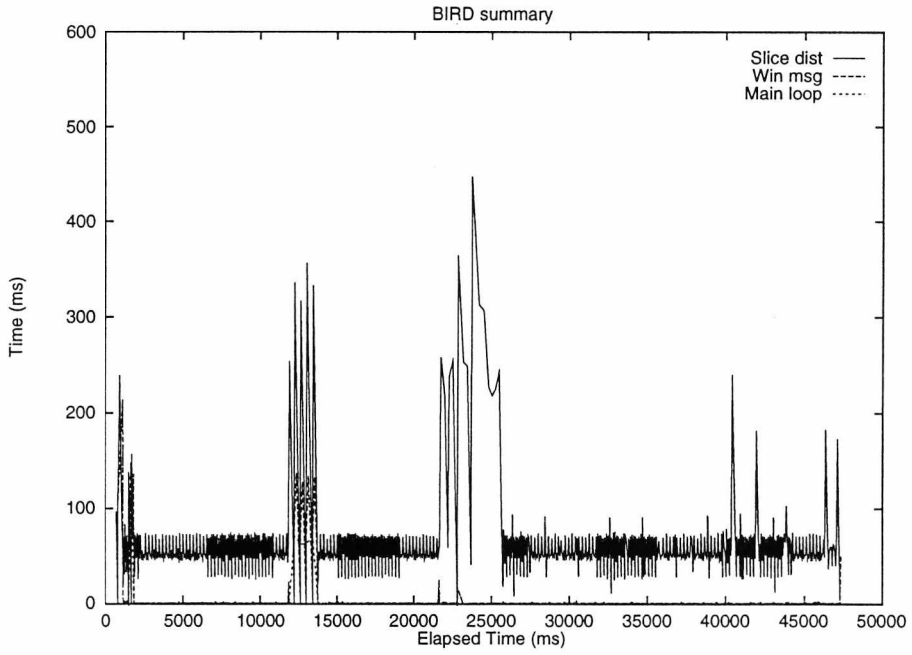

Figure 6.10: Effect of continuous media on a non continuous object presentation

6.6 A highly interactive presentation

In order to assess how the system behaves with a document with a large number of links, a document was defined with four run time objects and timestones set at 1 second intervals for 20 seconds in each of these objects. Each timestone will trigger a link whose effect will be a void action targeted at one of the run time objects. The reason for using a void action is to prevent the time required to process the action from interfering with the measurements related to link triggering.

Figure 6.13 shows the behaviour when 80 links are created and each process handles conditions on 40 of them. In this situation, when any status variable is updated, the process has to check 40 links that might be triggered by the change. Figure 6.14 shows the average error in reaching a timestone. The system allows (in this experiment) a timestone to be "reached" within 10 ms its due time. A positive value (i.e. a positive "error") indicates that the timestone was reached before the time it was due and a negative value (i.e. a negative "error") indicates a delay. If the curves with the error at reaching the timestone and the distance between the clock ticks are plotted (figure 6.15),

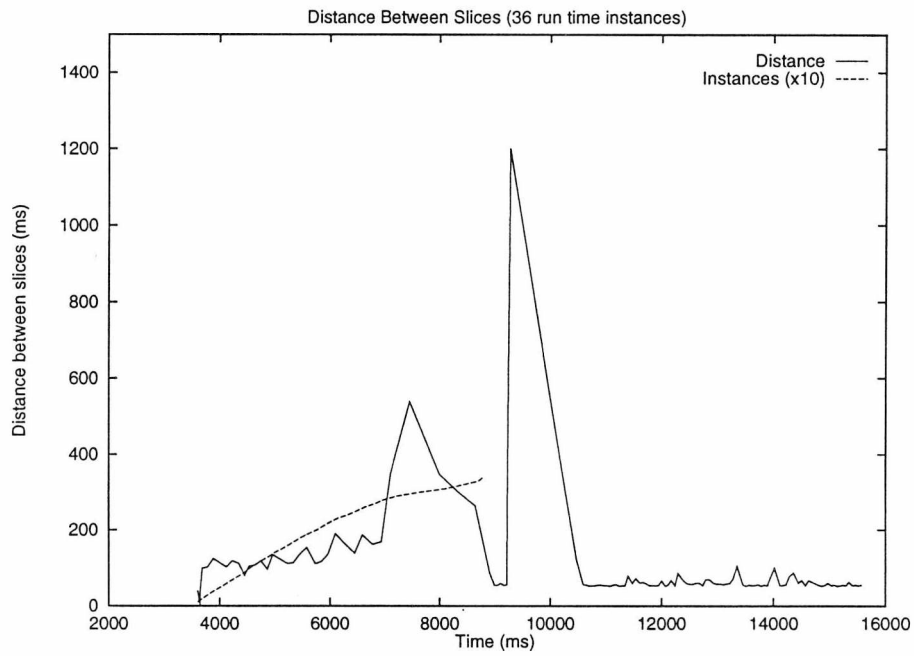

Figure 6.11: Distance between CPU slices

it can be seen that the peaks where the Windows timer is late cause an extra delay at reaching the timestones.

Figures 6.16 and 6.17 shows similar graphs when the number of links is increased to two hundred and each process handles one hundred link conditions.

The graphs shows that the behaviour of the timer is not very precise and at intervals of about 3 seconds, there is a delay caused by its re-synchronisation. In order to verify the effect of the clock resolution on the overall performance, the same document was run using an altered version of the clock which will post timer messages as fast as it can. The effect of is that the CPU load will be 100% during all presentation. Figures 6.18 and 6.19 shows the graphs when two hundred links are created. It can be noticed that the distance between slices is greatly reduced and the timestones are reached with much more precision than in the previous cases. This is even more noticeable when there are no delays in the timer for re-synchronisation.

The final conclusion from the graphs is that although the resolution of 55 ms provided by MS-Windows 3.1 is good enough for a large range of applications, its unreliability make it unsuitable for many uses.

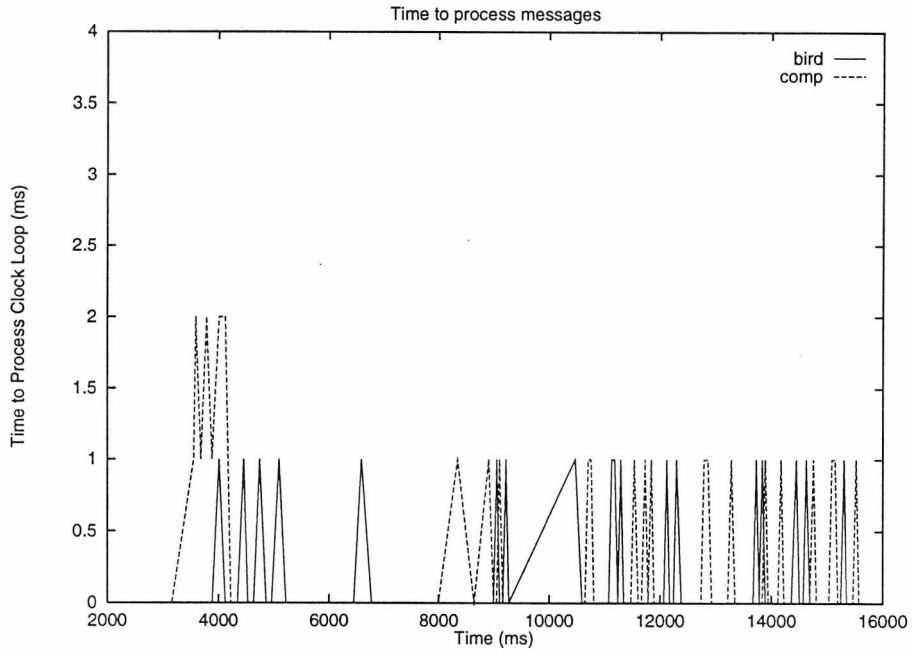

Figure 6.12: CPU time used by process (`bird` and `comp` are active processes)

6.6.1 Complex link conditions

As the link handling is implemented, the actual number of conditions in the link condition has very little impact on the time it takes to trigger the link. As described in section 5.7.2, the conditions in the link are in a tree and the processes that hold them are responsible for keeping that information updated in the link factory. Therefore, when one of the conditions is changed, it is very fast for the link factory to check whether the trigger condition is satisfied as it can assume that all information has already been updated. The time required to update each condition grows linearly with the number of conditions.

6.7 Final remarks

In summary, the conclusions that can be drawn from the analysis presented in this chapter are:

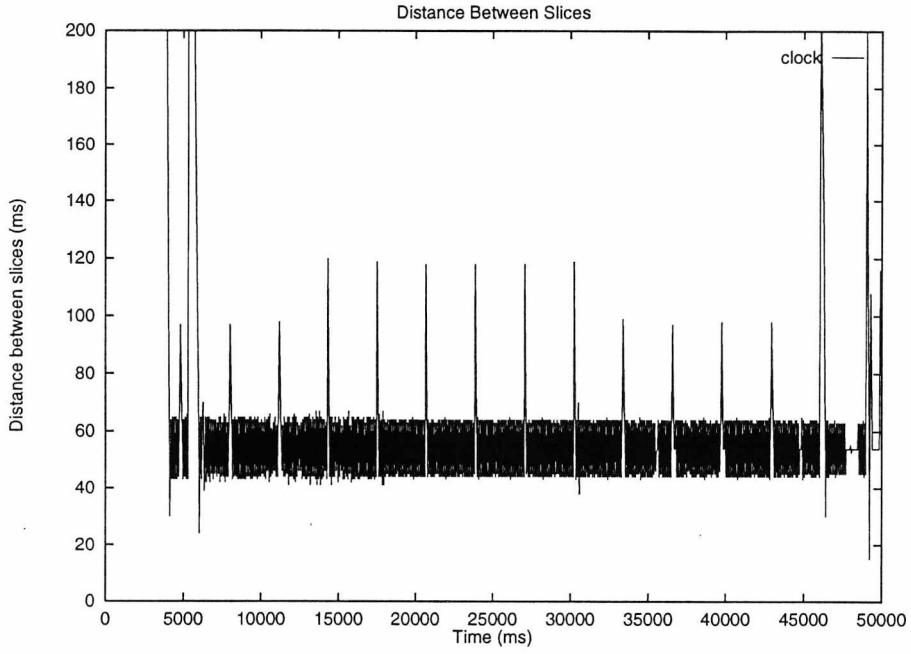

Figure 6.13: Distance between CPU slices (80 links triggered)

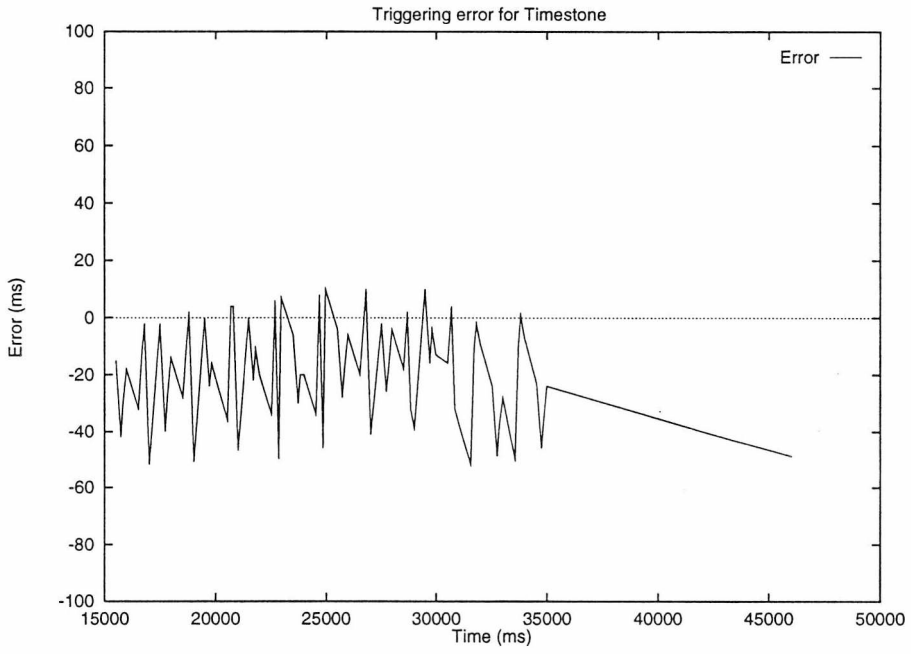

Figure 6.14: Error reaching timestones

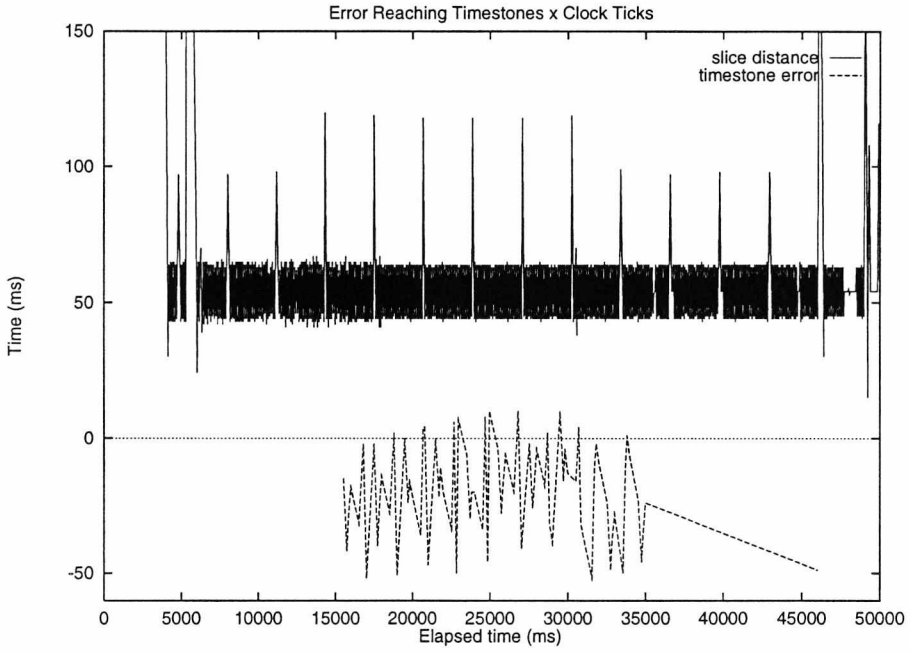

Figure 6.15: Distance between slices vs Error reaching timestones

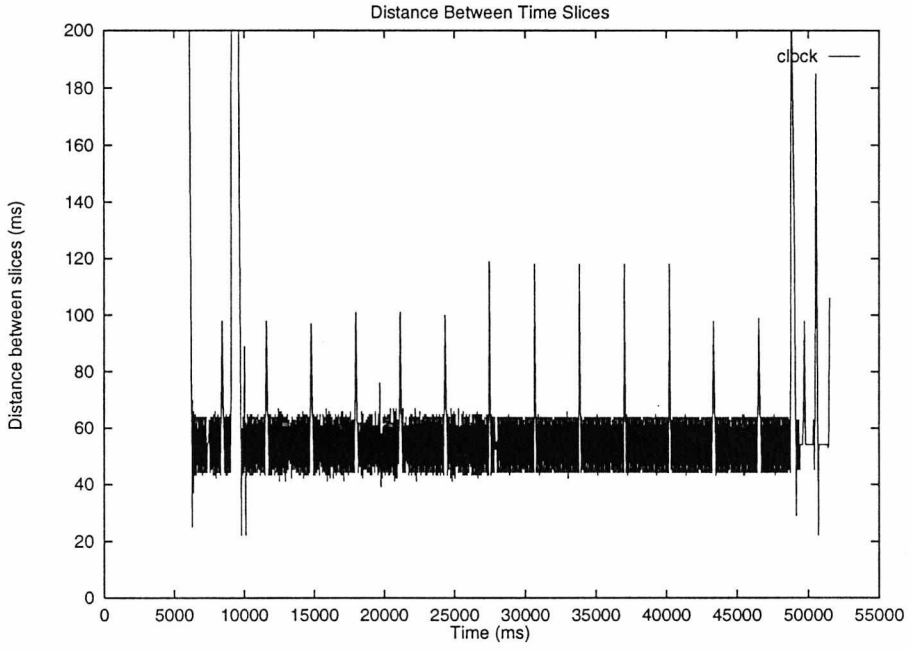

Figure 6.16: Distance between CPU slices (200 links)

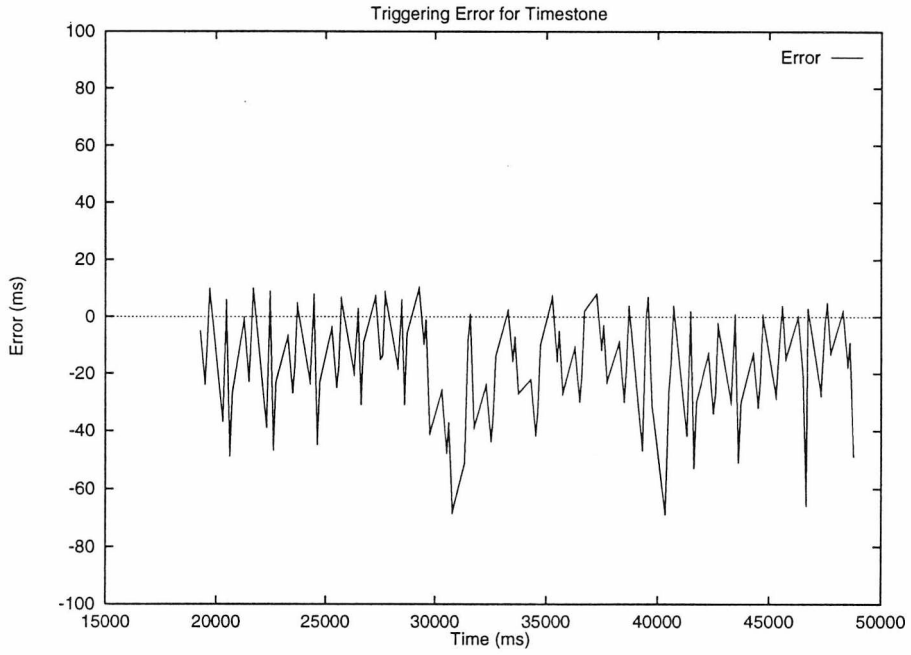

Figure 6.17: Error reaching timestones

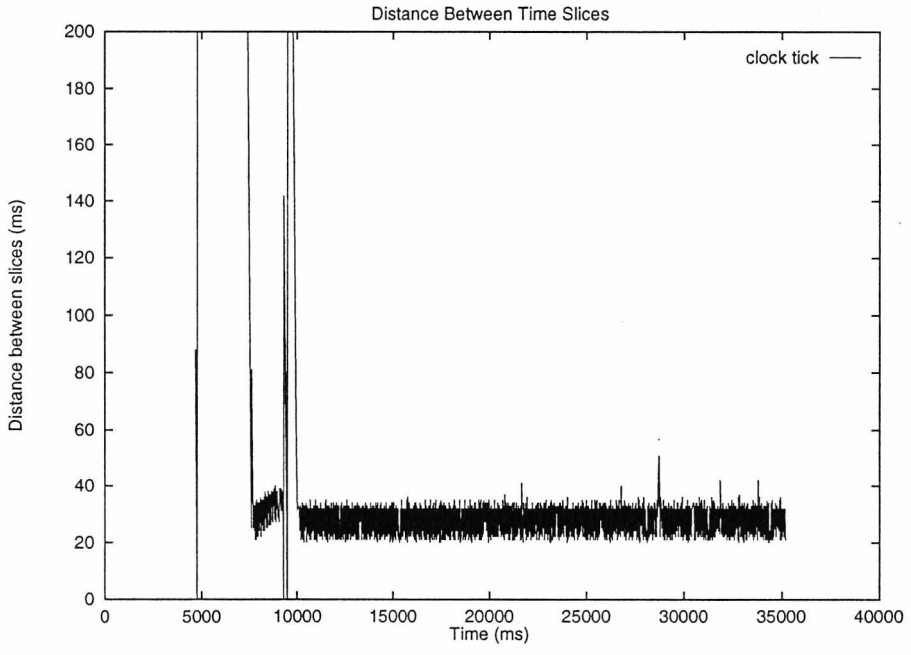

Figure 6.18: Distance between CPU slices using rapid ticks (200 links) — the initial peaks are due to the initial processes creation

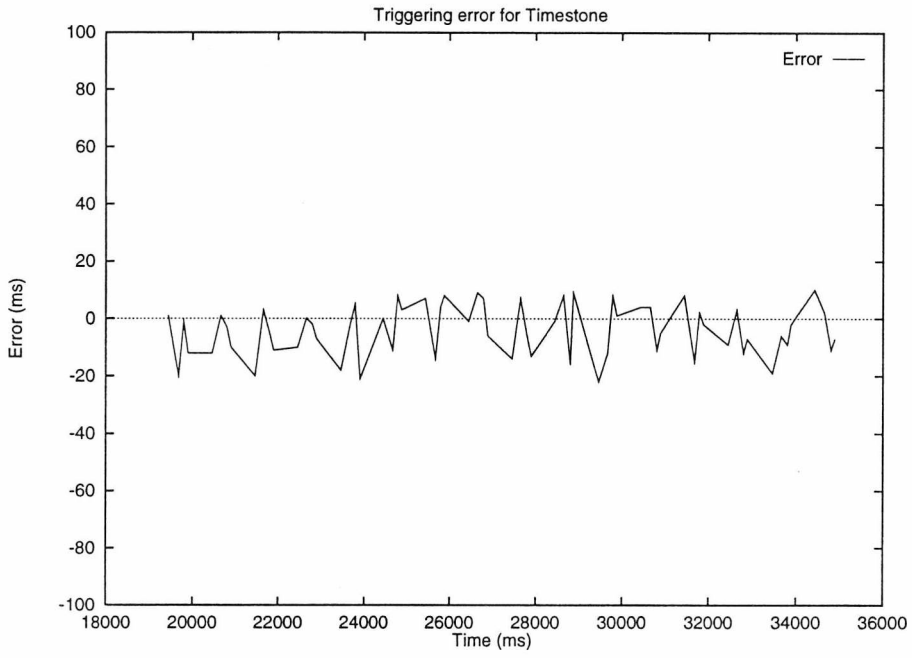

Figure 6.19: Error reaching timestones (using rapid ticks)

Item Measured	Machine	
	486-66	486-33
Paint a 1024x768 pixels bitmap	1720	4320
Paint a 320x180 pixels bitmap	150	355
Mount and start an AVI file	1700	3100
Mount a motion JPEG file	-.-	1720
Get position of a motion JPEG file	-.-	960
Get position of an AVI file	<1	<1

Table 6.8: Some slow actions

- It is not worth making the points where each process releases control to the operating system too frequent as the overhead incurred does not compensate for the improvement in the granularity of CPU usage by the processes.

The *smoothness* of a presentation is much more dependent on the execution of "slow" actions than on the time used to update internal status and interact with the registry. The processes, therefore, should avoid performing more than one slow action within the same clock cycle. Table 6.8 shows the times measured for some demanding operations, which should be broken up when possible in order

to reduce their influence on the overall performance.

- Since the main factor influencing how the whole presentation performs is the time it takes to process *slow actions*, it is more important to define the activities within the main loop in each process carefully than to define overall process priorities; interaction with the system kernel is very fast, not imposing a heavy overhead.

- The creation of processes also imposes extra delays to the system. Table 6.9 shows some figures for starting up processes. The huge difference between the minimum and maximum times are due to the fact that the longest times are usually obtained when the first instance of the process is started that requires some dynamic libraries to be loaded.

Item Measured	Minimum	Maximum	Average
Startup graphics window	95	2136	260
Startup triggered link	90	430	187
Startup AVI window	139	2420	265

Table 6.9: Times required to startup processes (times in ms) — *(486/66)*

- The total number of processes is limited not only by the total available memory but also by "resources" and the amount of memory available in the first 640 KB. The design should not therefore require large number of processes.

- The average clock resolution influenced how links are processed: it is worth transferring a large group of actions to the target process rather than transferring one action at a time;

- Programmers must be aware that the timer services provided by MS-Windows 3.1 are not reliable;

- The maximum CPU time a process can use can be configured dynamically depending on the system load although there is not much support provided to interrupt "slow actions" that have to be atomic, such as interacting with the JPEG card.

When there are several pending *fast activities*, such as retrieving triggered links or link conditions from the link factory or executing actions, the process should perform as many activities as possible within the CPU time limit.

- The use of a non preemptive operating system forces a programming style that requires a very disciplined way of handling CPU usage by every process. This requires the programmer to be much more careful with resource usage than is necessary when developing programs for a preemptive operating system.

Chapter 7

A Critical Analysis of MHEG

This Chapter presents an overview of the evolution of the MHEG work, its current status and limitations imposed by the model.

7.1 The evolution of MHEG

The MHEG standard has suffered several basic structural changes since the start of the work of SG29 WG 12, and it has been a difficult task to to keep up with the frequent changes. Figures 7.1 and 7.2 show the class hierarchies in 1991, 1992 and the present edition.

The figures show how the hierarchy has become simpler as the standard evolved. The hierarchy proposed in 1991 (figure 7.1) presented too many details, almost to the level of a graphics widget set, defining forms of input and output. The structure was also very limiting in the sense that media types were defined in the hierarchy (text, graphics, picture, audio and audiovisual), therefore requiring the basic structure to be changed when a new medium was added.

The hierarchy in 1992 (figure 7.2 left) was cleaner but the types of supported media were still pre-defined providing poor support for extensions.

The current hierarchy (figure 7.2 right) is much more generic, providing the basic

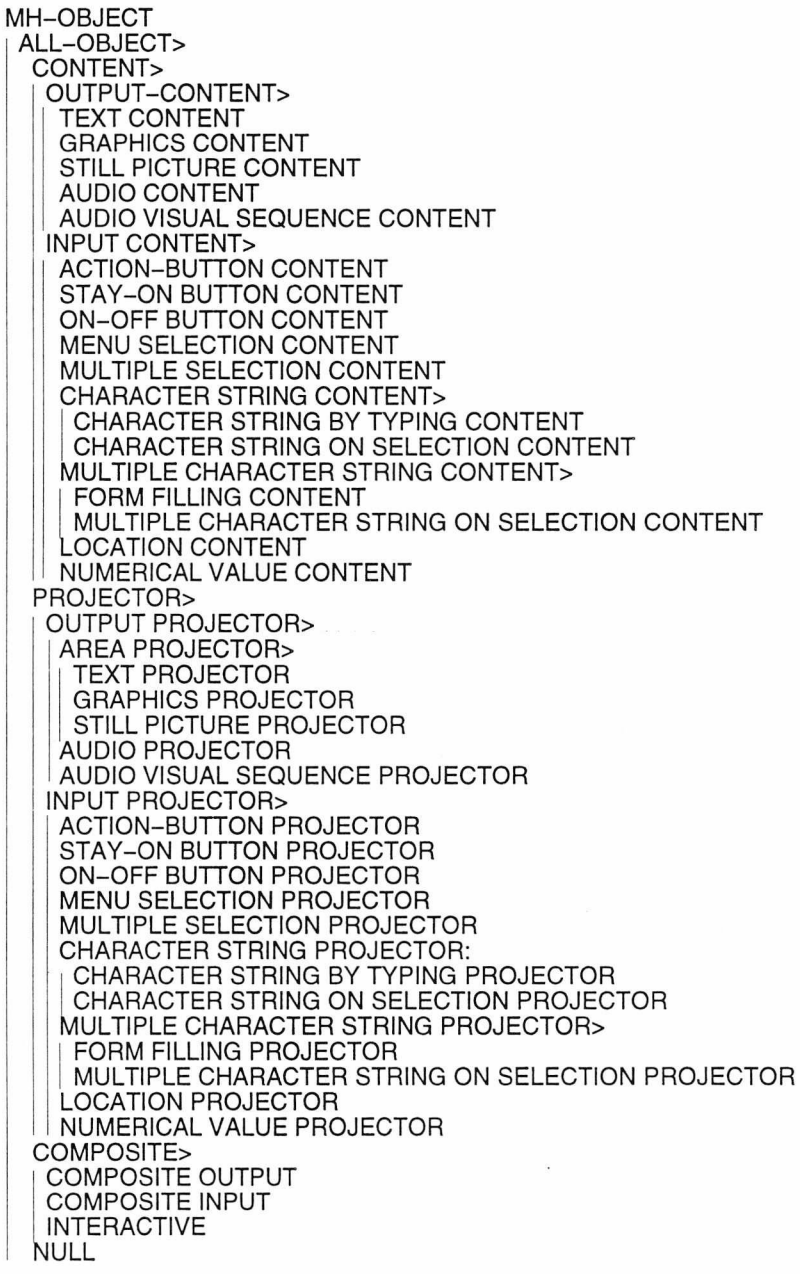

Figure 7.1: MHEG Class Hierarchy in 1991 as in [MHEG, 1991]

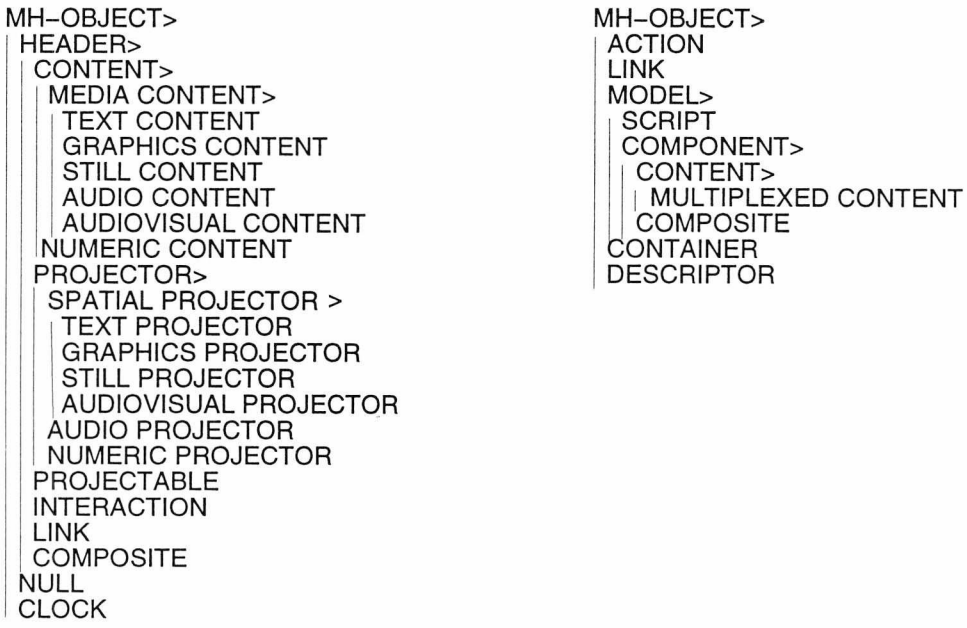

Figure 7.2: MHEG Class Hierarchy in 92 (*l*) and today (*r*)

framework for extension: media types are not defined in the hierarchy; they are included in subclasses of the *Model* class, therefore any new medium will be just a new instance of a *Content* class. The model is now abstract enough to cope with new additions of media on the exchange level.

The important point to be perceived from the evolution is that concepts gradually became clearer: early hierarchies mixed up *exchange* with *presentation* resulting in a less powerful format, with an undesired focus on implementation.

Another very important evolution in the standard was the definition of run-time objects. Early versions of MHEG did not refer explicitly to run-time objects, although their existence had to be inferred for the implementation of any run-time system. The initial absence of run-time objects can be understood as it was initially thought that the standard was dealing with information *exchange* rather than *presentation* and it could be expected that the use of one content object (already exchanged) would be implicit.

The absence of explicit run time objects, however, led to a confusion in naming where content reuse was attempted. For example, if a picture was to be used more than once during a presentation, there was no way of distinguishing one instance from the

other as all references were made to what is (in the current hierarchy) a model object. It was not clear how an application would make a distinction between instances; this would require, in the worst case, an undesired multiple exchange of the model object (*content*) so that the application could make the distinction in terms of the model. The inclusion of run-time objects was enough to resolve the naming inconsistency.

7.1.1 Abstraction level of MHEG

As the class hierarchy became more abstract during the evolution of MHEG, modeling "simple" applications became more complex. This is a natural evolution in defining a standard. As a trade off between generality and expressive power, an interchange format has to be able to represent different formats so it has to be as generic as possible but this may lead to it not being simple to use (or even usable).

The decision on what should be included in an international standard is a very difficult one. If the standard is too generic, it is almost impossible to define a minimum level of compliance. On the other hand, if it tries to encompass the requirements of only a certain group of applications, it may prove difficult to accommodate future extensions.

This led to discussions, even among members of MHEG, on how to define a simple application, such as a hypertext. As a standard aimed at being *general*, MHEG does not address specific types of applications (e.g. hypertext) and may therefore be too "heavy" for some uses.

This is the specific case with *hypertext*. In hypertext, pieces of text can be used as anchors for links. In order to have an adequate support for this kind of abstraction, MHEG should support directly at least the concept of *anchor* or some similar mechanism. However, when thought of as *hypermedia* in general, rather than *hypertext*, the support provided is generic enough as any run time object can be made selectable and therefore can be used as an anchor. In the case of a text only document in which it is desired to have every word as an anchor, it is necessary to define each word as an individual object, which may prove that the general approach does not have adequate

performance at such a high level of granularity.

The decision on what should be the scope of MHEG and consequently the definition of its abstraction level has been one of the major tasks of the group and the difficulties are reflected in the definition of the class hierarchy: there is a clear need to be very general but on the other hand the necessity to cater for specific applications can not be underestimated. However, there is at least one project —the Berkom Globally Accessible ServiceS (GLASS) [Fokus, 1995]— that proves that the structure proposed by MHEG Part 1 is powerful enough to model hypertext/hypermedia.

A natural evolution of MHEG is the same as the one suffered by Dexter (discussed in section 2.7). The standard initially proposed is generic enough to accommodate all uses but it is not efficient for specific uses. To cater for some specialised applications, specialised subsets of it will be defined. The MHEG itself has noticed this requirement by proposing *MHEG Part 5*.

MHEG 5

13522-5 MHEG Subset for Base Level Implementation (MHEG 5 for short), was added to the MHEG work in November 1994 with the objective of specifying "requirements for a basic set of MHEG objects" [MHEG, 1995b] to be applied to the domain of "simple multimedia applications" (e.g. video on demand, navigation and browsing applications).

MHEG-5 defines a subset of MHEG-1 (discussed in Chapter 3) by specialising some of its classes. The class hierarchy of MHEG 5 is shown in figure 7.3 and the relationship with MHEG 1 is shown in figure 7.4

An MHEG 5 presentation consists of a set of *scenes* which are made up of *presentables* which represent the actual perceivable information (i.e. a run time instance of a model object in MHEG 1); *lists* used to group up elements; *fonts* used within textual presentables; *links* responsible for defining the reaction on events; *script* used to define more complex behaviour than a link (link is just an abstract class, as "scripting" is

outside the scope of MHEG 5); *variables* which can hold the state of a scene ingredient, and are used to exchange data with an outside entity.

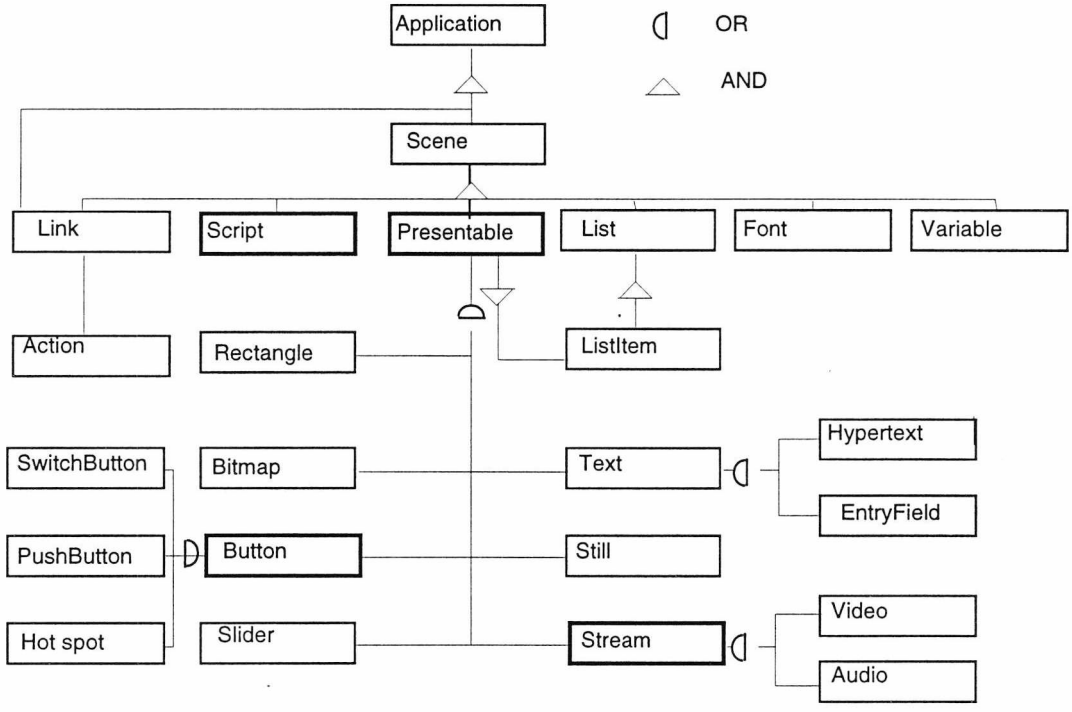

Figure 7.3: MHEG 5 classes (from [MHEG, 1995b])

The class specialisation presented by MHEG 5 in a sense represents a return to previous stages of MHEG providing less expressive power, but providing a more efficient representation for a large (maybe the largest) group of *hypermedia* applications.

7.2 Defining the look and feel of a presentation

Defining the "look and feel" of a presentation is outside the scope of MHEG, although very often desirable. More abstract classes as in MHEG 1 make it more difficult if not impossible for authors to define how objects are presented.

A specialised hierarchy such as the one defined in MHEG 5 allows closer control over how to present elements even if the final *format* of the objects is not defined (e.g. the author can define a "push button" but not what it will actually look like).

If it is desirable to actually define how objects will be rendered, MHEG is not

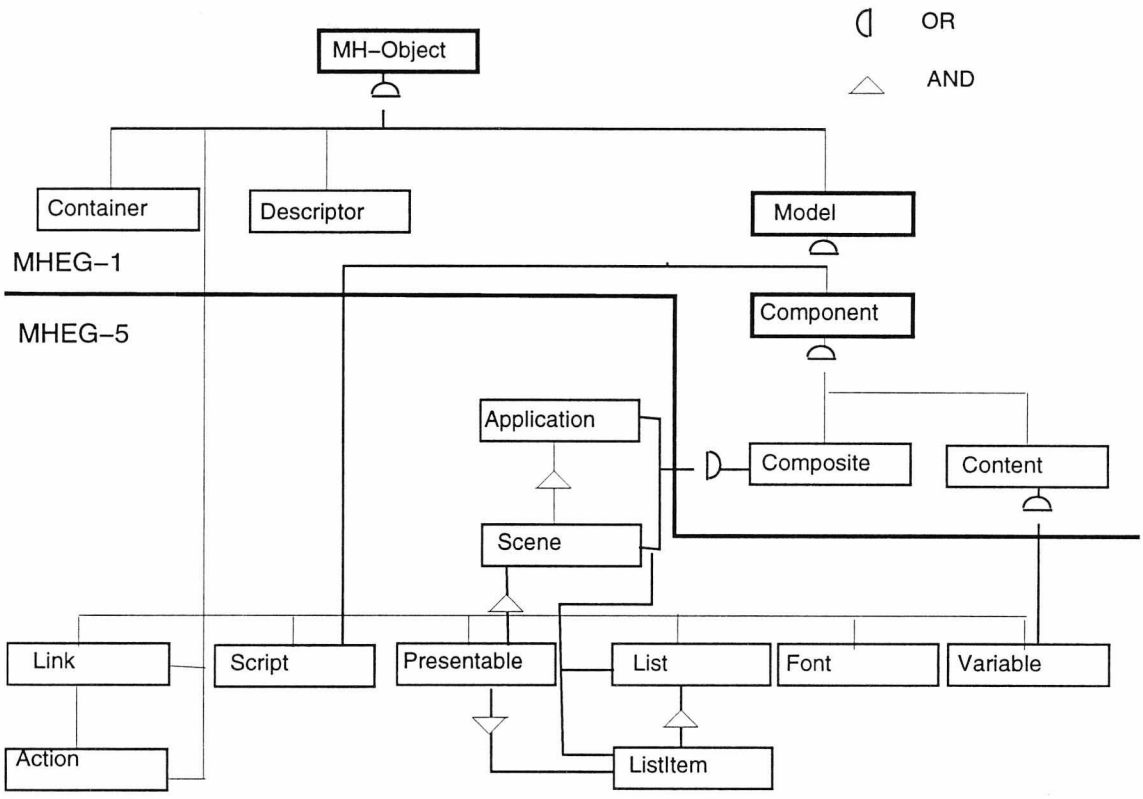

Figure 7.4: Relationship between MHEG 1 and MHEG 5 classes (from [MHEG, 1995b])

suitable, although it may be used in conjunction with other standards with a focus on graphics such as PREMO. However, it is important to note that not defining the "look and feel" is not a handicap for many hypermedia applications. For a large number of applications it is not necessary to define exactly how information will be presented at the source level. This is the approach taken, for example, by HTML (section 2.12.2) which defines the logical structures but allows each browser to present that structure in a different way. It should also be noted that one of the main objectives of MHEG is efficiency, and the higher level of abstraction provided by not transmitting the "look and feel" improves efficiency.

7.2.1 Final form representation of objects

MHEG defines objects in *final form*, which means that they are no longer revisable. Therefore, MHEG allows a content to be replaced but not edited. This behaviour,

although suitable for reading and browsing documents, is not adequate for highly interactive authoring systems. From this point of view, it is expected that MHEG will be used to *run* documents created by authoring tools based on other standards such as PREMO and HyTime.

Here again efficiency is improved by having final form representation of objects. However, another implication of "final form" is that links are structures fully resolved in MHEG which, although it enhances performance, does not provide support for links computed on-the-fly which are regarded as "complex" and therefore should be specified by scripting languages outside the scope of MHEG. The author is of the opinion that the existence of computable links is required by a large number of applications and that MHEG should allow the definition of *processing links*, not leaving to external scripting languages the whole task of defining relationships more complex than an end-to-end link.

Links that require further processing before the execution of their effect can impose severe overheads to the overall system performance, therefore some strict constraints must be imposed. One reasonable constraint that can be imposed on processing links is that the scope of their effect is limited to objects whose "preparation status" is "READY". This will limit the eventual search process that will be started by the link effect. The author should always be aware of the performance degradation risk that such links can impose. This limitation is reasonable, and seems to impose little extra effort on the author. An example where such type of link would be useful is when, for example, it is desired to "reduce the volume of all audio objects by 20%". In this case, the link process must locate all audio objects to process the link effect. The definition of more complex interactions should still be left to external scripting languages, not only to optimise performance but also because the definition (authoring) of such interactions will require a higher level of abstraction than that provided by MHEG.

In the proposed architecture, the implementation of processing links that comply with the above restrictions can be easily implemented and can be efficient because:

- The scope of the processing is limited to objects already "known" to the engine (the 'preparation status" is "READY") avoiding the risk of a search of "all objects in the world".

- The *registry* and the *link factory* already keep several status information variables about objects which can, in some cases, be the only information required to compute the target of the link effect.

The implementation of processing links would require a change to the process that deals with *triggered links* as shown in figure 7.5. The changes add a module to compute the destination (targets) of the link before the link effect is processed. The module has to interact with the link factory and the registry to identify the targets.

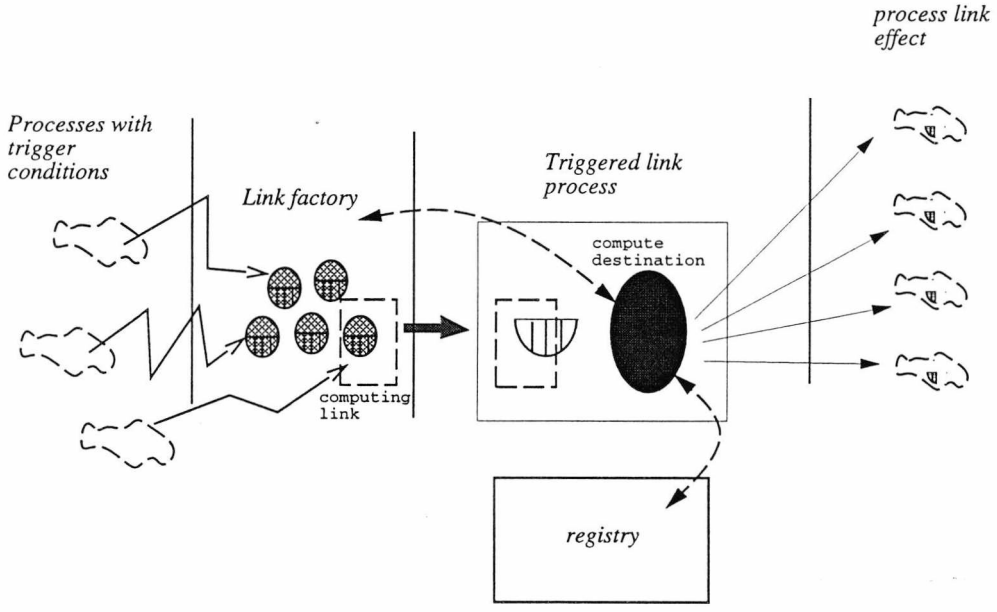

Figure 7.5: Processing sequence for a *processing link* (adapted from figure 6.5)

7.2.2 Relationship to HyTime and PREMO

The limitations on the definition of the "look and feel" and also on changing the contents of objects makes MHEG suitable for use either as the output of authoring tool based on PREMO and HyTime or for providing objects to be rendered by PREMO or

HyTime engines. Figure 7.6 shows a possible relationship between MHEG, HyTime and PREMO, adapted from [ISO, 1994a].

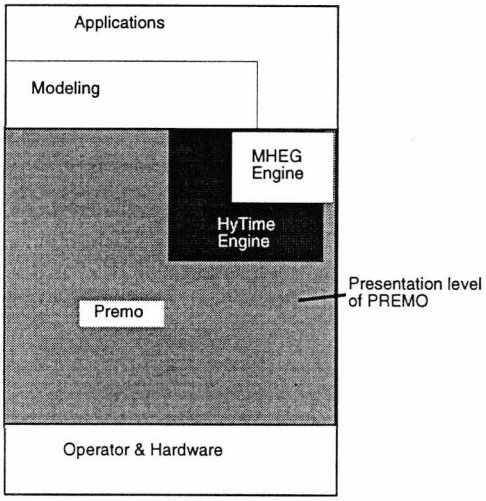

Figure 7.6: MHEG, PREMO and HyTime relationship (adapted from [ISO, 1994a])

The view of the author is that the three standards can be used together, depending on use:

- PREMO is very much implementation oriented, and is capable of allowing users to define a look and feel for a presentation. However, PREMO does not provide the general abstraction provided for MHEG for multimedia exchange, making it less generic than MHEG.

- MHEG aims at providing efficient and generic forms for interchanging documents in final form. The mechanisms for exchange are powerful but if it is desired to define rigidly what a document should look like, MHEG is not adequate.

  For documents aimed at browsing, the mechanisms provided by MHEG are not only powerful but also powerful enough to allow local customisation: the reader can specify how structural components are presented. This is the approach used by the WWW with success.

- HyTime, in part because it is more mature than the two previous standards, provides the most powerful abstractions for generic multimedia authoring. The

generic mechanisms provided for manipulating time and space (by the use of *batons* and *wands*) is more powerful than the mechanisms provided by the other standards. However, as it is at a high level of abstraction, it is not efficient in terms of presentation.

The author expects that tools for translating documents created using HyTime to MHEG will be created. Once the high level behaviour of a document is defined, it is not too difficult to translate that to the more efficient, but less expressive MHEG form, when, for example, a generic specification to define timing must be translated into corresponding MHEG set of actions.

7.3 MHEG engine

The standard does not specify how the *engine* should be implemented but assumes that it is capable of performing actions on objects. This is discussed in the next section.

7.3.1 Object orientation in MHEG

Limitations

MHEG defines classes of objects to denote the structure of interchangeable objects. It does not define *methods* for the classes and therefore it uses a more restricted form of object orientation than in a programming language [Cardelli and Wegner, 1985]. The standard regards MHEG objects as "passive information entities which have to be interpreted by an appropriate presentation environment (i.e. the MHEG engine) to realise behaviour" [MHEG, 1994b, section 6.2.1] The engine should be able to apply actions to objects, and the behaviour of such actions is polymorphic: a 'run' action targeted at a video run-time object has a different semantics from a 'run' action targeted at a audio run-time object.

An action, as defined by the standard, is targeted at an object. If it is desired to

retrieve the speed at which the run-time video `my_video#1` is being played, an action such as `get speed` would be targeted at the object `my_video#1`. The target object must be known to the engine in order to perform the action.

In the same way, MHEG defines that, in order to prepare a model object, an action `prepare` should be targeted to the object. Although this seems to be providing a uniform framework for delivering messages, in this case the target object is not known to the engine and therefore the way the action is handled depends not only on identifying the target but also on the type of the message. Similarly, a "new" action is targeted at a run-time object which is not yet available. A new run-time object is created from a model object that must be in phase `03` or `04` in the timing diagram shown in figure 7.7. If the model object is not ready, there is an implicit prepare action targeted at it.

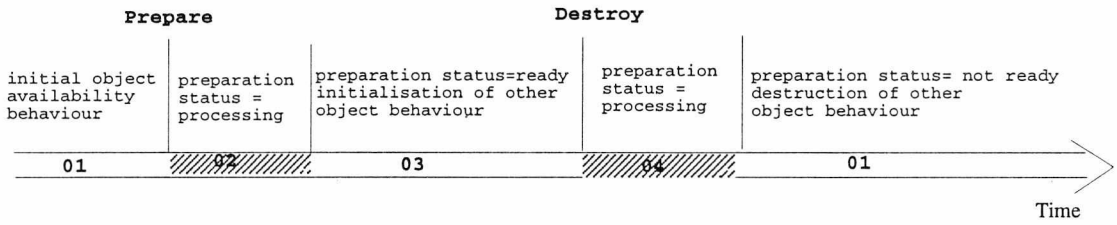

Figure 7.7: Timing diagram of model object availability (in period `01` the object is not known to the engine)

If the engine is thought of as being monolithic, this causes no problem, because it will have to identify the action and the target in order to perform the processing. However if, as in our proposed architecture, the engine is a set of independent cooperative processes, with a central registration and message routing point, this solution is inadequate; the decision as to which process will handle an action depends not only on the target of that action but also on the action itself, as shown in the examples below:

- *an action is targeted to a model object in phase 03 or 04 (figure 7.7):* the engine will receive the request to process the action on the object identified. In this case it will have to identify the process that handles that object and post the request to it. The core of the engine (or the kernel in our case) does not have to know how

to process that message, which is a desired feature if we want the system to be extensible.

- *an action is targeted to a model object in phase 01:* in this case, the object is not yet available to the engine; therefore it must interpret the action to identify the processing that should happen. If the action is `prepare`, the engine will process it. Any other action is likely to be an error.

  This case is different from the previous one in the sense that only the process handling the *engine* itself can process an action targeted to an unknown object, especially if this action aims at making the object available.

  In our implementation, a `prepare` action is processed by the kernel, and all it does is to start the process that knows how to prepare that type of object. When a new medium is introduced, the kernel is simply reconfigured to be aware of the medium and the process that should be invoked to prepare an object of that type.

- *a* `new` *is targeted to a non existing run-time object:* the standard defines that the run-time object is created using the object defined by the model class. If the model object is not ready it will be prepared. The author is of the view that an action to create a run-time object should be targeted at the model object with the run-time index as a parameter. The engine (or the process handling the model object) has all the details to perform the creation of the object.

  In our implementation, a `new` action is processed by the process that handles the model object. The process of creating a run-time object varies according to the type of the object (it is not covered by the standard) and in some cases requires the contents of the model object to be copied to the run-time object.

Suggested improvement

Some of these difficulties arise from trying to incorporate only some of the concepts of object oriented languages in MHEG. In a pure object oriented language (such as

Smalltalk [Goldberg and Robson, 1986]) all messages are directed to an object. Unlike in an MHEG engine, in Smalltalk all objects are available to the environment (which has a similar role to the MHEG engine). Therefore the run-time support system always knows who should be the target of a given action. The object may not be capable of handling a certain message and it then delegates the processing to its superclass. When a new instance of an object is created, a message is sent to the class of that object and not to the instance to be created itself. The environment stores all the information required to create the instance object in objects belonging to a meta class. When a new class of objects is created, a meta class describing those objects is defined providing the information required to create instance objects.

The object is created by sending a message `new` to the class of the desired object, which is at a higher level of abstraction than an object instance, and is fully resolved when the message is sent. In most cases, that class will ask its super class to process the `new` message and will send an initialisation message to the created instance. Eventually, the class `Object` (super class of all classes) will create the object using the information of its meta class in a uniform way for all sub-classes of object.

As in the MHEG world we are not talking about pure object oriented languages the uniformity of handling object creation is not possible or even desired. The current approach assumes the existence of an *engine* and that this engine is capable of applying actions to objects. If the existence of the *engine* is made explicit and if it must be able to perform some actions, the authorship and implementation of engines and objects is simpler to understand, particularly if a distributed approach is desired.

The changes implied can be demonstrated by the following examples:

- The `prepare` action should not be targeted at the object to be prepared, but at the *engine*, which can be seen as a higher level object. The engine should either know how to process the prepare action, or should know which process will handle that type of object.

- The `new` action should be targeted at the model object with the run-time index as parameter. The process handling the model object should know how to create the run-time object. This approach would also prevent the implicit preparation of a model object when a `new` action is targeted at an object that is not ready.

The above changes can lead to more efficient implementations (which is one of main goals of MHEG) as it does not assume that the engine will be the only point for handling actions, and that it will only have to sort messages depending on their destination: those aimed at available objects; and those aimed at non available ones. As it is now, the engine has not only to be aware of the availability of the destination object but also on the type of message being transmitted. This approach requires a change to the system structure when new media and actions are added, which is not desirable. The process of adding or making an object available to the system will always require some special treatment and should be explicitly addressed to the *engine*.

7.4 Final remarks

The strong and weak points of MHEG can be summarised as follows:

- *Good for integrating 3rd party applications:* MHEG does not deal with the internal representation of contents data (the *within component* layer in Dexter's model (section 2.7) which makes it suitable for extensions and for accommodating new types of objects.

- *It is too generic:* this is the case with MHEG part 1, which is abstract enough to accommodate the various applications of hypermedia. It is expected that subsets of the standard will be defined to deal with specific uses, as is already the case with MHEG 5.

- *Relationship with HyTime and PREMO:* the view of the author is that MHEG will be used together with PREMO and HyTime: HyTime is more abstract,

and therefore more powerful in terms of defining the high level behaviour of hypermedia but is not efficient enough to cope with timing and QoS requirements; PREMO is directed to the final presentation of multimedia and lacks the generic mechanisms for document interchange.

- *It overlaps with PREMO:* the boundaries between the two standards are not yet fully resolved and require more research to define their scope.

- *It does not fully define the role of the engine:* MHEG assumes the existence of an engine that it is capable of applying actions to objects but not that objects are capable of performing actions on themselves in the sense used by object oriented languages.

  The view of the author is that although the standard should not define *how* the engine should be implemented, it should define a set of actions that the engine should understand. In particular, all actions targeted at objects not available to the engine should be addressed to the engine itself and not to the object, as currently happens.

  This change would provide more efficient distributed implementations of the MHEG engine and would not penalise a monolithic implementation.

- *Final form representation of objects:* this leads to efficiency in the presentation of objects as no further processing is required. However, MHEG is not suitable for interactive uses and contents cannot be updated, only replaced.

- *End-to-end links:* In MHEG all links must be resolved, with no further processing required. This is related to the previous point.

  The view of the author is that this is not always desirable and MHEG should provide two types of links: a *resolved link* as defined now; and a *processing link* that requires some pre-processing. To avoid problems with security, an engine may decide not to allow *processing links*.

Chapter 8

Conclusion

This last Chapter presents concluding remarks about the work performed. It starts with a summary of the thesis, and a discussion of the proposed model and its implementation. The final part contains suggestions for enhancements to the work presented and suggestions of future research.

8.1 General comments

As was said previously, one of the main impediments to the development of *multimedia* and *hypermedia* applications has been the lack of standards. The investment required to develop such applications is too high not to have portable systems and documents.

The standardisation process started with the development of standards for *mono-medium* data such as JPEG for images and MPEG for video. However, standards for the exchange of complete applications or documents are still under development.

Some of the requirements that such standard should take into account include:

- *Extensibility:* it is not possible to predict the future availability of new media or technology;
- *Openness:* no standard should depend on technology controlled by one specific

vendor. A standard should take advantage of networks and distribution and provide means to interact with external services such as name servers;

- *Separation of data and structure:* contents data must exist independently of its structure, so that several "views" of that data may be created depending on user's need;

- *Application independence:* a standard, to be generic, should not focus on one specific type of application.

This thesis analysed the suitability of one of the emerging standards for multimedia exchanges (MHEG) and proposes an architecture for its implementation. One of the main problems met during the development of this work was the constant changes to the MHEG proposal. The prototype developed is, therefore, not up to date with the latest version of MHEG.

8.2 MHEG

MHEG is the first attempt to develop an international standard to cater for the generic requirements of multimedia interchange documents. As a standard which is not targeted at a particular type of application, it is very generic and it is expected that several sub sets of the standards will appear taking into account specific applications.

MHEG defines objects in final form, which make it efficient for dealing with documents that do not require changes but unsuitable for interactive authoring applications. As documents do not require further processing to be presented, it is possible to implement very efficient systems based on the standard.

The standard does not define the look and feel of a presentation. Here again this leads to efficient exchange, as less information has to be transmitted; it allows the look and feel to be defined at the end point, but provides poor support for, say, an artistic document where the look and feel is as important as the data itself. For this type of

application, the point of view of the author is that MHEG will be used together with other standards such as PREMO.

One fact that may jeopardize the adoption of MHEG is the time it is taking to make it an International Standard. The work is at least one and a half years behind schedule, and it is not expected that it will be concluded before 1996. It is also important to note that during this period of time, it has suffered a lot of change in its structure, some of it basic.

8.3 A proposed architecture for MHEG objects

The architecture proposed is based on a *kernel* that provides an extension to the operating system and processes that interact with this kernel via interfaces.

The kernel is responsible for registering objects, name resolution, interprocess communication and clock synchronisations. The manipulation of contents data is performed by client processes specialised for one specific medium. This separation provides extensibility at two different levels, as discussed below.

8.3.1 Extensibility

Extension is desirable at several levels:

- *Communication level:* the communication infrastructure may be upgraded to new technologies, and it is desirable that this upgrade is insulated from the client processes. In the architecture proposed, all interprocess communication is performed via the kernel and a change in the infrastructure has to be dealt with at that level alone. As long as the kernel interface to clients is the same, from the point of view of a client process it is transparent whether the kernel is using CORBA, direct RPC or DDE or if it is now able to access a new external naming service.

- *Media processing level:* adding new media, devices or actions to be performed on them is expected to be a frequent operation.

  In this case, it is desirable that only the processes that will be affected by the changes will be updated; the kernel should not be responsible for dealing with medium specific information. Here again, the architecture proposed provides an adequate framework as only the specialised client — either dealing with a *model* or a *run time* object (section 5.4) — "knows" and is responsible for dealing with media specific requests. The kernel is unaware of how actions targeted at a medium will affect media specific data, because it does not interpret this type of information.

  At this level it is also possible to add extensions that are specific to applications (e.g. some specific action to be understood by a process).

- *Pre-processing capabilities:* although MHEG defines documents in final form (discussed below), it may be desirable (for specific applications) to add mechanisms to perform some sort of processing on data (e.g. to compute dynamic links in a hypertext).

  Here again the architecture proposed can cope with extensions in this directions as this type of extension can be added to the process responsible for "preparing" the object type to be pre-processed. This process then generates the appropriate MHEG structures with no change being required at the kernel level.

- *Adding new types of media specific processes:* extensions to the types of processes capable of dealing with media will be made by programmers. It is required that a minimum of programming effort is demanded. The architecture proposed for processes provides for code reuse as only media specific code has to be written (which may be the code to control a device).

  The architecture also provides:

- *Default behaviour:* although processes can be extended as discussed above, the infrastructure provides default behaviour for several actions that are inherited by all processes but that may be overridden by the programmer of a client process.

- *Reuse of existing tools:* it is possible to integrate any tool that provides an external controlling interface by implementing a process to deal with that tool as if it were a new medium. An example of such integration was described for the inclusion of the *Text-To-Speech* module. In particular, any application which is DDE aware can be integrated by using a process to "remotely control" it.

8.3.2 System performance

The measurements obtained show that the architecture proposed is very efficient. The overhead imposed by the system kernel for dispatching messages is very low. Typically, using an entry level PC (*486-66* IBM clone), an interaction with the kernel takes less than one millisecond.

Performance bottlenecks are created by slow actions such as repainting large parts of the screen or starting up a process such as a video clip. However, under normal usage it is acceptable to have longer delays for an operation that causes a major change to what the user is perceiving.

Also because location of output devices is important in most hypermedia presentations, the system relies on a central point always being available. The machine that starts up a presentation is the one that will host the *central kernel*, including the time server, and is also responsible for registering other servers. In most cases location transparency is not desired, because users need to be able to define the physical location of output, and so the reliance on a central point makes the overall system control simpler, improving performance.

8.4 Enhancements and further work

The work started in this thesis can be extended in several direction:

- *Update to final version of MHEG:* it is likely that MHEG will become an International Standard next year. Updating the prototype to the final version is the natural extension to this work.

- *Operating system:* the prototype was developed under Windows 3.1, which presents a series of limitations as discussed in Chapters 5 and 6. The recent release of a new version of Windows (Windows 95) which provides preemptive multitasking, offers a more suitable environment for developing multimedia applications for desktop computers.

  The use of a preemptive operating system will require less in terms of programming discipline for developing applications, as discussed in Chapter 5.

- *Integration into a distributed system environment:* as we previously discussed, Windows provides poor inter-process communication features. A desirable extension to this work is to analyse how the structure behaves under a distributed environment, such as the one provided by OMG's CORBA, or Microsoft and DEC's COM.

- *Authoring interface:* the documents used for testing were manually created. It is clear from that experience that MHEG needs a computer supported tool for authorship.

  One interesting research direction is to define an authoring tool based on the emerging PREMO standard to generate MHEG documents. This work would also be useful to help define the boundaries between the two standards that still seem to have large parts which overlap. As PREMO is *Object Oriented*, providing operation signatures for objects, it seems a natural standard to be used by an engine

for MHEG, which is object based. However, a thorough assessment on how well they can be integrated is not yet possible as PREMO is in a very early stage of development.

HyTime is better developed and there are already some engines available; developing authoring tools based on HyTime to generate MHEG objects is also important to assess how well the two standards can be used together.

- *Integration with other tools:* a future development of this work, in particular by using a distributed environment would allow its integration with other multimedia research being carried out at the University of Kent.

  In particular, it is desirable to create MHEG documents representing video sequences with annotation, adapting the tool described in [Linington and Teixeira, 1993] to MHEG authoring; and to access the high speed ATM network and the existing video [Henshaw, 1994] and audio servers [Li, 1994].

- *Exposure to real-life problems:* the tests carried out were based on artificial situations. The best way to fine tune a system is to develop "real-life" applications. The use of the tools discussed above and the development of applications accessed by the public would provide invaluable data to assess the use of MHEG in practice.

8.5 Final remarks

The main points achieved by this work are summarised below:

The MHEG standard

Directly related to the work developed by ISO WG 12 this thesis has assessed:

- *Expressiveness:* MHEG has been shown to be capable of expressing "common" hypermedia documents.

- *Extensions:* MHEG has proved to be suitable for extension by providing a homogeneous structure into which new media can easily be accommodated.

This work also suggests that the *engine* should be explicit in the standard, and not assumed to exist and that it should be capable of applying actions to objects. This change makes the process of implementing a distributed architecture more efficient as actions can be distributed to processes handling objects without having to decode them twice as required by the current format. The need for double interpretation arises from the fact that the process that will handle the action depends not only on the target of that action but also on the type of action, as discussed in section 7.3.1.

The architecture proposed

The architecture proposed for implementing MHEG object has achieved the goals of:

- *Extensibility:* The architecture is extensible at several levels:
  - Kernel: a central point is responsible for messaging, both internally and externally. It provides transparency over the technology used for interprocess communication.
  - Client: processes are responsible for dealing with media specific data. Therefore new devices, media, and actions may be added with no impact on other parts of the system.
- *Open architecture:* third party applications may be used by the system allowing data encoded by proprietary algorithms to be controlled by a native application via a process acting as a "remote control".
- *Pre-processing hooks:* the structure provides adequate hooks for adding pre-processing to data made available to the system, although this feature is not currently required by MHEG.

These hooks can also be used by external scripting languages in order to define complex relationships between elements.

- *Software reuse:* the structure proposed for defining processes allows a large amount of software reuse when new media, devices and/or actions are introduced. Typically, the programmer has only to write the media specific code when extensions are added.

- *Performance:* the architecture does not add a significant overhead to the overall performance. The values measured are compatible with the hardware used.

Appendix A

Overview of MHEG Classes

This section provides a summary of the main characteristics of the MHEG classes. The quotation in *italics* in the beginning of each subsection is from the Working Group document and gives a one sentence summary of that class.

A.1 Mh-object

"Mh-object class provides the identification of MHEG objects"

Mh-object is an abstract class; i.e. no objects are instantiated from it but only from its subclasses. Mh-object provides (by inheritance) identification for all objects in MHEG hierarchy, such as the identification of the standard and its version, identification of the objects class, MHEG identifier of the MHEG object and general object information (name, owner, version, date, keywords, copyright, license and comments).

A.2 Action class

"Action class objects are used within a link object to describe the link effect".

The general structure of an action class object is shown in figure 1.1.

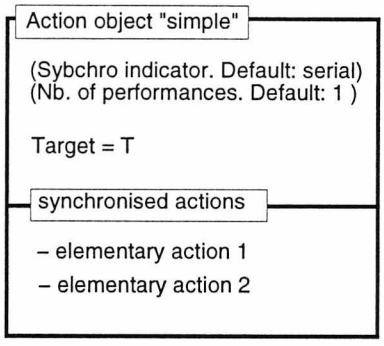

Figure 1.1: MHEG Simple Action Object Structure

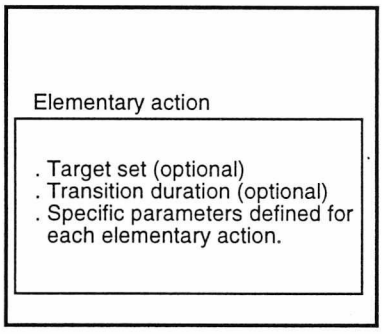

Figure 1.2: MHEG Elementary Action Structure

An action class object provides the following information:

- *Synchro indicator:* valid values are *parallel* or *serial.* The synchro indicator specifies the type of processing of the synchronised actions.
- *Target set:* an optional attribute that specifies the targets of the actions.
- *Number of performances:* specifies the number of performances of the synchronised actions. The default is one.
- *Synchronised actions:* the set of elementary actions (figure 1.2) and/or action objects (figure 1.1).

Actions may also be nested (figure 1.3). Targets of nested actions default to the targets defined by the outer object. The attribute synchro indicator applies to all actions.

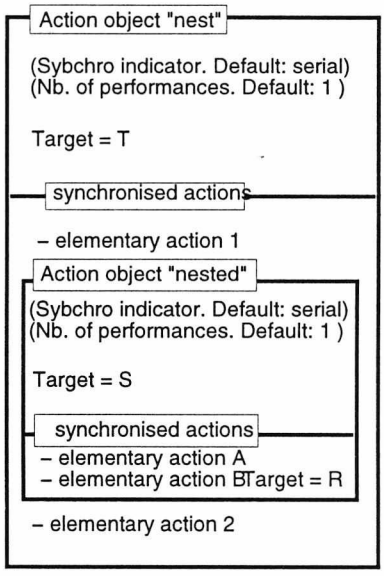

Figure 1.3: Nested Action Structure

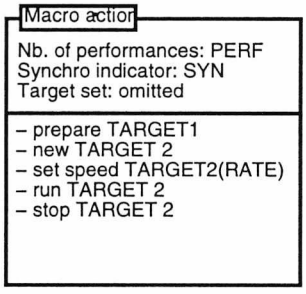

Figure 1.4: Macro Action Structure

The standard also defines macro actions (figure 1.4) to optimise the coding of complex or frequently used actions. Macros also provide for the sharing and reuse of complex behaviour. In the figure, the parameters to the macro are written in capitals.

A.3 Link class

"The link class defines a structure which defines a set of relationships."

MHEG links are used in two situations:

1. *In composite objects* where links can provide positions for other objects in time and space;

2. *In a stand alone manner* to allow creation and modification of general inter-object relations;

A.3.1 Characteristics of MHEG links

MHEG links have the following characteristics:

- *Directional:* they connect one object to one or more objects;
- *Conditional:* the actions defined are processed only if the conditions are satisfied. The conditions can be on the dynamic attributes of the source object or on external objects.

  All conditions are assumed to apply at the same time and the specification of the application should take into account the effects of sequential testing.

A.3.2 Link structure

A link is divided into two parts (figure 1.5):

1. *Link condition:* this describes the conditions that should be satisfied to provoke the *link effect.* The link condition specifies *trigger conditions* and *constraint conditions.*

   Constraint conditions define contextual conditions, such as speed or audible volume for example, that should be valid at the moment the link is triggered.

   Another difference between *trigger conditions* and *constraint conditions* is that a trigger condition specifies a "before" and an "after" condition while the constraint condition requires only the "moment" satisfaction of conditions.

   The following events may trigger a link:

   - Temporal events caused by timestones or delay;
   - Action events derived from any MHEG action;

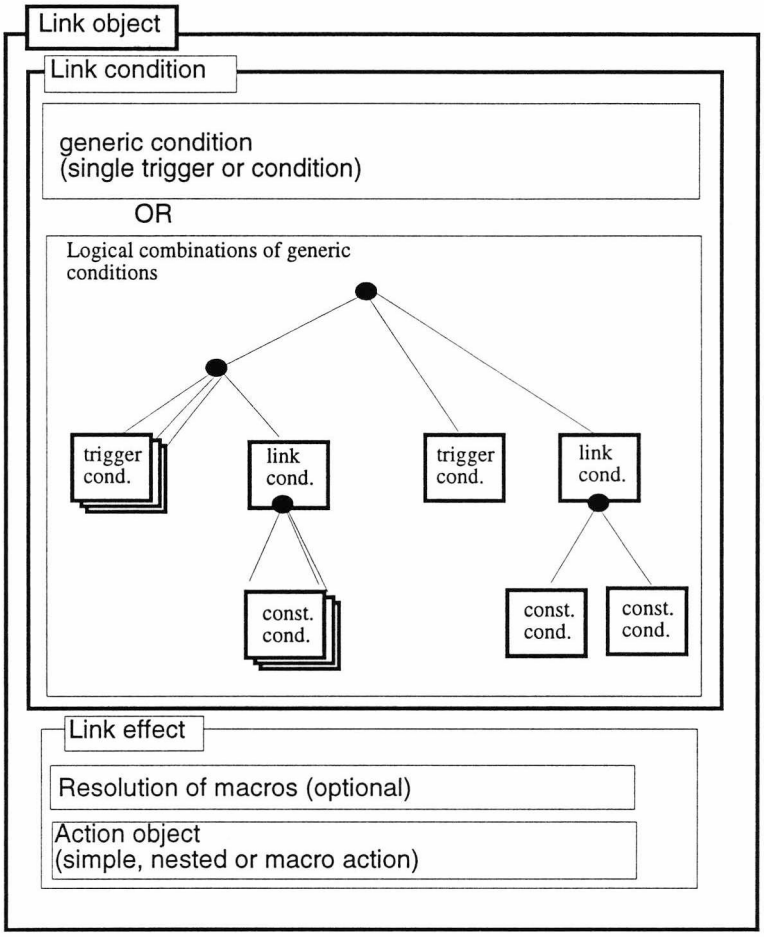

Figure 1.5: Link Object Structure

- Interaction events caused by user interaction.

2. *Link effect:* specify the processing that should occur when the link condition is satisfied.

A.4 Model class

The model class is an abstract class. The subclasses of model are the classes that define objects that are interchanged and that can be instantiated (possibly more than once) taking the class as a template. For example, a content object that contains an image is exchanged once but it may be instantiated more than once (different rt-objects) in several contexts. The creation of an rt-object does not affect the model object.

A.5 Script class

"The script class defines a container for complex relationships between MHEG objects and run-time objects, defined by a non-MHEG language."

Scripting languages are outside the scope of MHEG. Languages for specifying complex behaviour or compositions, such as in [Pinto, 1993] should be used, although in the future MHEG should add extensions for scripting language support. The standard, however, provides for the exchange of script objects. A script object contains:

- *The script classification:* an optional parameter that can be used to identify the type of script data.
- *A script hook:* which allows the identification of the external scripting language used. The *script hook* composed of an scripting language identification and a scripting language description.
- *The script data:* a inclusion or reference to the script itself.

A.6 Descriptor class

"The descriptor class defines a structure for the interchange of resource information about a set of other interchanged objects."

This class allows the description of interchanged documents so that presentation systems are able to adapt the available resources to the requirements for a given object, and it also allows an MHEG engine to determine whether it is capable or not of proceeding with a presentation.

It is not required that all interchanged objects also have a corresponding descriptor object.

A.7 Component class

> *"The component class models objects that may be interchanged within or across using applications."*

The component class is an abstract class inherited by *content class* and *composite class*. It contains no information and is in the hierarchy for clarity only.

A.8 Content class

> *"The content class contains or refers to the coded representation of media information together with a parameter set containing information required for content presentation."*

Content class is a model class object, i.e. it is interchanged by applications and it is used as a template for rt-component objects. A content object contains the following information:

- *Data classification:* an optional parameter that provides assistance in determining the type of the perceived media data.
- *Content original perception:* an optional parameter that provides the original spatial and temporal perception (e.g. original size, duration, etc.), and the original audible perception (e.g. volume and volume range).
- *Content hook:* the encoding/decoding mechanism used to represent the data.
- *Content data:* inclusion or reference to the actual data.

A.9 Multiplexed Content class

> *"The Multiplexed content class contains or refers to the coded representation of multiplexed media data together with a description of each*

multiplexed stream."

A multiplexed content object contains all the information a content object does, and in addition it provides:

- *Stream list:* an ordered list that describes the streams in the multiplexed data.

A.10 Composite class

"The composite class provides the support for associating multimedia and hypermedia objects."

A composite object contains the following information:

- *Composite behaviour:* a description of the sequencing and interrelationships between the elements of the composite, including initial behaviour.
- *The number of elements in the composition;*
- *List of composition elements:* a list of all sibling elements in the composition.

An element of a run time composite (`rt-composite`) is called a *socket*. A socket can be:

- *Empty:* when a null rt-component is plugged into it;
- *Presentable:* when an rt-component or rt-multiplexed content is plugged into it; or
- *Structural:* when a rt-composite is plugged into it.

A.11 Container class

"The container class provides support for regrouping multimedia and hypermedia data in order to interchange them as a whole set"

The objective of regrouping objects is to facilitate interchange in the sense that a group of objects may be transferred as one set. Because the interchange is done in one object only, the use of a container ensures that all components are included. A container object is not a *model object*, therefore no run-time instances can be created from a container.

Bibliography

[Adie, 1994] C. Adie. Network Access to Multimedia Information. Request for Comments (RFC) 1614, May 1994.

[Adobe, 1995] Adobe. Acrobat. *Adobe Magazine — British Edition*, (1):14–16, 1995.

[Aho *et al.*, 1986] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

[Akscyn and McCracken, 1984] Robert M. Akscyn and Donald L. McCracken. ZOG and the USS CARL VINSON: Lessons in System Development. Technical Report CMU-CS-84-127, Carnegie-Mellon University, Pittsburgh-PA, 1984.

[Apple, 1993] Apple Corporation. *Inside Macintosh: QuickTime, Apple Technical Library*. Addison Wesley Publishing Company, 1993.

[Barry, 1994] Jeff Barry. The HyperText Markup Language (HTML) and the World Wide Web: Raising ASCII Text to a New Level of Usability. *The Public-Access Computer Systems Review 5*, 5(5):5–62, 1994.

[Berners-Lee, 1993] Tim Berners-Lee. Hypertext Markup Language (HTML). Internet Draft `URL=ftp//info.cern.ch/pub/www/doc/htmlspec.ps`, 1993.

[Berners-Lee, 1995] (Editor) Tim Berners-Lee. URL Uniform Resource Locators. Internet Draft available at URL: `http://www.w3.org/hypertext/WWW/-Adressing/URL/url-spec.html`, 1995.

[Berson, 1992] Alex Berson. *Client/Server Architecture*. McGraw-Hill Series on Computer Communications, 1992.

[Birrel *et al.*, 1982] A. D. Birrel, R. Levin, R. M. Needham, and M. D. Schroeder. Grapevine: An exercise in distributed Computing. *Communications of the ACM*, 25(4):260–273, Apr 1982.

[Brown, 1986] P. J. Brown. On-line Documentation. Personal communication, Apr 1986.

[Brown, 1987a] P. J. Brown. Hypertext: The way forward. Technical report, University of Kent, Canterbury, Kent, 1987.

[Brown, 1987b] P. J. Brown. The differences between UNIX guide and OWL Guide. Personal communication, 1987.

[Brown, 1987c] P. J. Brown. Turning Ideas Into Products: The Guide System. In *Hypertext'87 Proceedings*, pages 33–40, Chapel Hill, NC, Nov 13–15 1987.

[Brown, 1989] Heather Brown. Standards for Structured Documents. *The Computer Journal*, 32(6):505–514, 1989.

[Brown, 1994] P. J. Brown. Adding Value to a Network Hypertext: can it be done transparently? In *European Conference on Hypermedia Technology 1994 Proceedings*, pages 51–58, Edinburgh, UK, Sep 18–23 1994.

[Buford, 1994] John F. Koegel Buford. *Multimedia Systems*. Prentice Hall, 1994.

[Bulterman *et al.*, 1991] Dick C. A. Bulterman, Guido van Rossum, and Robert van Liere. A Structure for Transportable, Dynamic Multimedia Documents. In *Usenix Summer Conference Proceedings*, pages 137–156, Nashville, Tennessee, June 1991.

[Bulterman, 1993] Dick C. A. Bulterman. Retrieving (JPEG) Pictures in Portable Hypermedia Documents. In Tat-Seng Chua and Tosiyasu L. Kunii, editor, *Multimedia*

Modeling, pages 217–226. World Scientific, Nov 1993. Proceedings of the First International Conference on Multi-Media Modeling.

[Bush, 1945] Vannevar Bush. As we may think. *Atlantic Monthly*, 1945.

[Bush, 1967] Vannevar Bush. Memex revisited. *Evolution of an Information Society*, 1967.

[Cardelli and Wegner, 1985] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys, ACM*, 17(4):471–522, Dec 1985.

[Carr *et al.*, 1995] Les Carr, Wendy Hall, Hugh Davis, and Rupert Hollom. The Microcosm Link Service and its Application to the World Wide Web. Personal Communication, 1995.

[Casey, 1994] Thomas Casey. MHEG, Scripts, and Standardisation Issues. In Wolfgand Hersnrer and Frank Kappe, editors, *Multimedia/Hypermedia in Open Distributed Environments (Proceedings of the Eurographics Symposium)*, pages 29–43, Graz, Austria, June 1994.

[CCITT, 1988] CCITT. *Recommendation X.500: The Directory — Overview of Concepts, Models and Service*. International Telecommunications Union, Place des Nations 1211 Geneva, Switzerland, 1988.

[Charles Petzold, 1992] Charles Petzold. *Programming Windows 3.1*. Microsoft Press, 1992.

[Clark, 1992] Jeffrey D. Clark. *Windows Programmers Guide to OLE/DDE*. SAMS, 1992.

[Colaitis and Bertrand, 1994] F. Colaitis and F. Bertrand. The MHEG Standard: Principles and Examples of Applications. In Wolfgand Hersnrer and Frank Kappe,

editors, *Multimedia/Hypermedia in Open Distributed Environments (Proceedings of the Eurographics Symposium)*, pages 3–17, Graz, Austria, June 1994.

[Conklin and Begeman, 1987] Jeff Conklin and Michael L. Begeman. gIBIS: A Hypertext Tool for Team Design Deliberation. In *Hypertext'87 Proceedings*, pages 247–252, Chapel Hill, NC, Nov 13–15 1987.

[Coulouris *et al.*, 1994] George Coulouris, Jean Dollimore, and Tim Kinderberg. *Distributed Systems: Concepts and Design.* Addison-Wesley, 1994. Second edition.

[Danny Goodman, 1987] Danny Goodman. *The Complete Hypercard Handbook.* Bantam Books, 1987.

[Davis *et al.*, 1994] Hugh C. Davis, Simon Knight, and Wendy Hall. Light Hypermedia Link Services: A Study of Third Party Application Integration. In *European Conference on Hypermedia Technology 1994 Proceedings*, pages 41–50, Edinburgh, UK, Sep 18–23 1994.

[Digital Equipment *et al.*, 1993] Digital Equipment Corporation, Hewlett-Packard Company, HyperDesk Corporation, NCR Corporation, Object Design Inc., and SunSoft Inc. The Common Object Request Broker: Architecture and Specification. OMG Document 93.xx.yy Revision 1.2, Object Management Group (OMG), Dec 1993.

[Drapeau and Greenfield, 1991a] George D. Drapeau and Howard Greenfield. MAEstro — A Distributed Multimedia Authoring Environment. In *Usenix Summer Conference Proceedings*, pages 315–328, Nashville, Tennessee, June 1991.

[Drapeau and Greenfield, 1991b] George D. Drapeau and Howard Greenfield. *MAEstro User's Guide*, 1991.

[Eckhard Moeller and Angela Scheller and Gerd Schürmann, 1990] Eckhard Moeller and Angela Scheller and Gerd Schürmann. Distributed Multimedia Information Handling. *Computer Communications*, 13(4):232–242, May 1990.

[Fokus, 1995] GMD Fokus. GLASS - Globally Accessible ServiceS — An Interactive Distributed Multimedia Presentation System. Available at URL: `http://www.focus.gmd.de/ovma`, 1995.

[Fountain *et al.*, 1990] A. Fountain, W. Hall, I. Heath, and H. C. Davis. MICROCOSM: An open model for hypermedia with dynamic linking. In A. Rizk, N. Streitz, and J. Andrè, editors, *Hypertext: Concepts, Systems and Applications (Proceedings of the European Conference on Hypertext)*, pages 298–311, INRIA, France, Nov 1990.

[Fujitsu and TechnoTeacher, 1995] Open Systems Solutions Fujitsu and TechnoTeacher Inc. *HyTime Application Development Guide*, May 1995.

[Goldberg and Robson, 1986] A. Goldberg and D. Robson. *Smalltalk-80 / The Language*. Addison-Wesley Publishing Company, 1986.

[Goldfarb, 1990] Charles F. Goldfarb. *The SGML Handbook*. Clarendon Press, 1990.

[Gray, 1991] Pamela Gray. *Open Systems: A Business Strategy for the 1990s*. McGraw-Hill Book Company (UK) Limited, 1991.

[Halasz and Schwartz, 1994] Frank Halasz and Mayer Schwartz. The Dexter Hypertext Reference Model. *Communications of the ACM*, 37(2):30–39, Feb 1994. Edited by Kaj Gronbæk and Randall H. Trigg.

[Halasz and Schwartz, 1990] Frank Halasz and Mayer Schwartz. The Dexter Hypertext Reference Model. In D. Benigni J. Molina and J. Baronas, editors, *Proceedings of the Hypertext Standardization Workshop*, pages 95–133. NIST Special Publ., Jan 1990.

[Handel, 1991] Rainer Handel. *Integrated broadband networks: an introduction to ATM-based networks.* Addison-Wesley, 1991.

[Hardman *et al.*, 1993] Lynda Hardman, Dick C. A. Bulterman, and Guido van Rossum. The Amsterdam Hypermedia Model: Extending Hypertext to Support Real Multimedia . *Hypermedia*, 5(1):47–69, 1993.

[Hardman *et al.*, 1994] Lynda Hardman, Dick C. A. Bulterman, and Guido van Rossum. The Amsterdam Hypermedia Model: Adding Time and Context to the Dexter Model. *Communications of the ACM*, 37(2):50–63, Feb 1994.

[Henshaw, 1994] Paul Henshaw. UKC ATM Video FileStore Application. Working paper, University of Kent at Canterbury, 1994.

[Herman *et al.*, 1994] I. Herman, G. S. Carsonn, J. Davy, D. A. Duce, P. J. W. ten Hagen, W. T. Hewitt, K. Kansy, B. J. Lurvey, R. Puk, G. J. Reynolds, and H. Stenzel. PREMO: An ISO Standard for a Presentation Environment for Multimedia Objects. In *Proceedings of the ACM Multimedia'94 Conference*. ACM Press, Oct 1994. Available by anonymous ftp.

[Hill and Hall, 1994] Gary Hill and Wendy Hall. Extending the Microcosm Model to a Distributed Environment. In *European Conference on Hypermedia Technology 1994 Proceedings*, pages 32–40, Edinburgh, UK, Sep 18–23 1994.

[Hill *et al.*, 1992] Gary Hill, Rob Wilkins, and Wendy Hall. Open and Reconfigurable Hypermedia Systems: A Filter-based Model. Technical Report CSTR 92-12, University of Southhampton, 1992.

[Hopper, 1990] Andrew Hopper. Pandora: An Experimental System for Multimedia Applications. *ACM Operating Systems Review*, 24(2), Apr 1990.

[Howard Rheingold, 1985] Howard Rheingold. *Tools for Thought.* Computer Book Division/Simon & Schuster, 1985.

[ISO, 1986a] ISO. *ISO-8879: Formal Public Identifiers*, 1986.

[ISO, 1986b] ISO. *ISO9070: Registration procedures for public text owner identifiers*, 1986.

[ISO, 1988] ISO. *ISO 9069 Information Processing — SGML Support Facilities — SGML Document Interchange Format (SDIF)*, Sep 1988.

[ISO, 1989] ISO. *ISO 8613 Information Processing - Text and Office Systems - Office Document Architecture (ODA) and Interchange Format.* 1989.

[ISO, 1992] ISO. *ISO/IEC DIS 10744 Information Technology — Hypermedia/Time-Based Structuring Language (HyTime)*, 1992.

[ISO, 1994a] ISO. *Information Processing Systems — Computer Graphics and Image Processing — Presentation Environments for Multimedia Objects (PREMO) ISO/IEC JTC1/SC24 Comittee Draft 14478-4.1*, Jul 1994. Part 1: Fundamentals of Premo.

[ISO, 1994b] ISO. *PREMO Multimedia Systems Services Working Draft ISO/IEC 14478-4.2/199x(E)*, 1994. Second Draft September 1994.

[Jakob Nielsen, 1990] Jakob Nielsen. *Hypertext and Hypermedia.* Academic Press, Inc, 1990.

[Jones and Hopper, 1993] Alan Jones and Andrew Hopper. Handling Audio and Video Streams in a Distributed Environment. *Operating Systems Review*, 27(5):231–243, Dec 1993.

[Koegel *et al.*, 1993] John Koegel, Lloyd W. Rutledge, John L. Rutledge, and Can Keskin. HyOctane: A HyTime Engine for an MMIS. In *Proceedings of ACM Multimedia 93*, Aug 1993. Available by anonymous ftp.

[Kohl *et al.*, 1994] J. T. Kohl, B. C. Neuman, and T. Y. Ts'o. The Evolution of the Kerberos Authentication Service. In F. M. T. Brazier and D. Johansen, editor, *Distributed Open Systems*, pages 78–95. IEEE Computer Society Press, 1994.

[Labs, 1994] Creative Labs. *Text-to-Speech User Manual*, 1994.

[Lampson, 1986] B. E. Lampson. Designing a Global Name Service. In *Proceedings of the Fifth ACM annual Symposium on Principles of Distributed Computing*, pages 1–10, Calgary, Canada, 1986.

[Leggett and Schnase, 1994] John J. Leggett and John L. Schnase. Viewing Dexter with Open Eyes. *Communications of the ACM*, 37(2):76–86, Feb 1994. Edited by Kaj Gronbæk and Randall H. Trigg.

[Li, 1994] N. Li. A Distributed Audio System. In Wolfgand Hersnrer and Frank Kappe, editors, *Multimedia/Hypermedia in Open Distributed Environments (Proceedings of the Eurographics Symposium)*, pages 109–121, Graz, Austria, June 1994.

[Linington and Teixeira, 1993] P. Linington and C. Teixeira. Exploiting interactive Video and Animation in Distributed Environments for the Design of Hypermedia and and Graphical User Interfaces. In *Proceedings of the VI SIBIGRAPI*, pages 213–220, Recife, Brazil, Oct 1993.

[Linington, 1992] Peter F. Linington. Introduction to the Open Distributed Procesing Basic Reference Model. In J. de Meer and V. Heymer and R. Roth, editor, *Open Distributed Processing*, pages 3–13. Elsevier, 1992.

[Markey, 1991] Brian D. Markey. Emerging Hypermedia Standards - Hypermedia Marketplace Prepares for HyTime and MHEG. In *Usenix Summer Conference Proceedings*, pages 59–74, Nashville, Tenessee, June 1991.

[Matthews and Dobson, 1993] Martin Matthews and Bruce Dobson. *Networking Windows for Workgroups*. Osborne Mc Graw Hill, 1993.

[McAleese, 1993] Ray McAleese. *Hypertext: Theory into Practice*. Intellect Books, 1993.

[McArthur, 1995] Jeffrey McArthur. SGML Frequent Asked Questions. Frequently posted to `comp.text.sgml` newsgroup, 1995.

[Meyer-Boudnik and Effelsberg, 1995] Thomas Meyer-Boudnik and Wolfgang Effelsberg. MHEG Explained. *IEEE Multimedia*, 2(1):26–38, 1995.

[MHEG, 1991] Multimedia and Hypermedia Information Coding Expert Group MHEG. Information Processing - Coded Representation of Multimedia and Hypermedia Information Objects. Technical Report Working Document S.4, ISO/IEC, Oct 1991.

[MHEG, 1993] Multimedia and Hypermedia Information Coding Expert Group MHEG. Information Technology - Coded Representation of Multimedia and Hypermedia Information Objects (MHEG) - Part I: Base Notation (ASN.1). Technical Report Working Draft Version: 1.0, ISO/IEC JTC1/SC29/WG12, Feb 1993.

[MHEG, 1994a] Multimedia and Hypermedia Information Coding Expert Group MHEG. Information Technology - Coded Representation of Multimedia and Hypermedia Information Objects (MHEG) - Part I: Base Notation (ASN.1). Technical Report Preparation Document for DIS, ISO/IEC JTC1/SC29/WG12, Feb 1994. Version: V0 - Rennes Meeting.

[MHEG, 1994b] Multimedia and Hypermedia Information Coding Expert Group MHEG. Information Technology - Coded Representation of Multimedia and Hypermedia Information Objects (MHEG) - Part I: MHEG object representation; Base Notation (ASN.1). Technical Report Draft International Standard 13522-1, ISO/IEC JTC1/SC29/WG12, Dec 1994.

[MHEG, 1995a] Multimedia and Hypermedia Information Coding Expert Group MHEG. International Standard ISO/IEC DIS 13522-1 — Information Technology - Coded Representation of Multimedia and Hypermedia Information Objects

(MHEG) - Part I: MHEG Object Representation; Base Notation (ASN.1). Technical report, ISO/IEC JTC1/SC29/WG12, Aug 1995. Draft Version: Tokyo Meeting.

[MHEG, 1995b] Multimedia and Hypermedia Information Coding Expert Group MHEG. International Standard ISO/IEC DIS 13522-1 — Information Technology - Coding of Multimedia and Hypermedia Information Objects - Part 5: MHEG Subset for Base Level Implementation. Technical report, ISO/IEC JTC1/SC29/WG12, Aug 1995. Working Draft 13522-5.

[Microsoft, 1994] Microsoft. *Microsoft Visual C++ version 1.5 Books on Line*, 1994.

[Mullender, 1989] Sape Mullender. *Distributed Systems*. Addison-Wesley Publishing Company, 1989.

[Mullender, 1993] Sape Mullender. *Distributed Systems*. Addison-Wesley, 1993. Second edition.

[Multicosm, 1994] Inc. Multicosm. Microcosm: a Technical Overview. Technical Note, 1994.

[Needham, 1989] R. M. Needham. Names. In Sape Mullender, editor, *Distributed Systems*, pages 89–101. Addison Weslley – ACM Press Frontier Series, 1989.

[Nelson, 1987] Theodor Holm Nelson. *Literary machines*. Prentice Hall, 1987.

[Newcomb *et al.*, 1991] Steven R. Newcomb, Neil A. Kipp, and Victoria T. Newcomb. The "HyTime" Hypermedia/Time-based Document Structuring Language. *Communications of the ACM*, 34(11):67–82, Nov 1991.

[Newcomb, 1991] Steven R. Newcomb. Standard Music Description Language complies with hypermedia standard. *IEEE Computer*, 24(7):76–79, July 1991.

[Orfali *et al.*, 1995] Robert Orfali, Dan Harkey, and Jeri Edwards. Intergalactic Client/Server Computing . *Byte*, pages 108–122, Apr 1995.

[OWL International, Inc, 1988] OWL International, Inc. *Guide: Hypertext for the PC*, 1988.

[OWL International, Inc, 1992a] OWL International, Inc. *Guide User Manual*, 1992.

[OWL International, Inc, 1992b] OWL International, Inc. *Logiix Command Reference*, 1992.

[OWL International, Inc, 1992c] OWL International, Inc. *Logiix User Manual*, 1992.

[Pichler *et al.*, 1995] Michael Pichler, Gerbert Orasche, Keith Andrews, Ed Grossman, and Mark McCahill. VRweb: A Multi-System VRLM Viewer. In *The First Annual Symposium on the Virtual Reality Modeling Language (VRML 95) Proceedings (To appear)*, San Diego, California, December 1995.

[Pinto, 1993] Paulo Fonseca Pinto. *An Interaction Model for Multimedia Composition.* PhD thesis, University of Kent at Canterbury, 1993.

[Pleas, 1994] Keith Pleas. Distributed OLE promises cross-platform remote capabilities for linking component software. *Byte*, Nov 1994.

[Rada, 1991] Roy Rada. *Hypertext: From Text to Expertext.* McGraw-Hill, 1991.

[Raggett, 1993a] David Raggett. HTML+ (Hypertext markup format). Internet Draft available at URL: `ftp://ftp.nisc.sri.com/draft-ragget-www-html-00.ps`, July 1993.

[Raggett, 1993b] David Raggett. HTML+ (Hypertext markup language). A proposed standard for light weight presentation independent delivery format for browsing and querying information across the Internet, July 1993.

[Raggett, 1994a] David Raggett. A Review of the HTML+ Document Format. In *Proceedings of the First International Conference on the World-Wide Web (W3)*. Elsevier Science, 1994. Also available at `http://www1.cern.ch/-Papers/WWW94/dsr.ps`.

[Raggett, 1994b] David Raggett. Extending WWW to support Platform Independent Virtual Reality. In *Proceedings of INET'94*, pages 242–247, 1994.

[Shneiderman, 1984] Ben Shneiderman. Response Time and Display Rate in Human Performance with Computers. *Comp. Surveys*, 16:0–1, 1984.

[Silberschatz and Galvin, 1994] Abraham Silberschatz and Peter B. Galvin. *Operating System Concepts*. Addison-Wesley, 1994.

[Soley, 1990] Richard Mark Soley. Object Management Architecture Guide. OMG TC Document 90-9-1, Object Management Group, Framingham, MA, Nov 1990.

[Stenzel *et al.*, 1994] H. Stenzel, G. S. Carson, I. Herman, and K. Kansy. PREMO: An Architecture for Presentation of Multimedia Objects in an Open Environment. In Wolfgand Hersnrer and Frank Kappe, editors, *Multimedia/Hypermedia in Open Distributed Environments (Proceedings of the Eurographics Symposium)*, pages 77–96, Graz, Austria, June 1994.

[Tanenbaum, 1992a] Andrew S Tanenbaum. *Modern Operating Systems*. Prentice Hall, 1992.

[Tanenbaum, 1992b] Andrew S Tanenbaum. *Operating Systems: Design and Implementation*. Prentice-Hall International Editions, 1992.

[Tebbutt, 1991] David Tebbutt. "Inside Pandora's Box". *BYTE*, 16(12):116IS–59–116IS–66, Nov 1991.

[van Herwijnen, 1994] Eric van Herwijnen. *Practical SGML*. Kluwer Academic Publishers, 1994.

[William, 1991] Ian William. Hypermedia for Multi-User Technical Documentation. In Heather Brown, editor, *Hypermedia/Hypertext and Object-Oriented Databases*, pages 201–217. Chapman and Hall, 1991.

[Woodhead, 1991] Nigel Woodhead. *Hypertext & Hypermedia: Theory and Applications*. Addison-Wesley Publishing Company, 1991.