CMIFed:
A Presentation Environment for Portable Hypermedia Documents

Guido van Rossum, Jack Jansen, K. Sjoerd Mullender, Dick C.A. Bulterman

CWI: Centrum voor Wiskunde en Informatica
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands
Phone: +31-20-592-4127, FAX: -4199
Email: {guido,jack,sjoerd,dcab}@cwi.nl

ABSTRACT

In this paper we discuss the architecture and implementation of CMIFed, an editing and presentation environment for hypermedia documents. Typically such documents contain a mixture of text, images, audio, and video (and possibly other media), augmented with user interaction. CMIFed allows the author flexibility in specifying what is presented when, using multiple simultaneous output channels.

Unlike systems that use a timeline or scripting metaphor to control the presentation, in CMIFed the user manipulates a collection of *events* and *timing constraints* among those events. Common timing requirements can be specified by grouping events together in a tree whose nodes indicate sequential and parallel composition. More specific timing constraints between events can be added in the form of *synchronization arcs*. User interaction is supported in the form of hyperlinks.

We place CMIFed in the context of the Amsterdam model for hypermedia documents, which formalizes the properties of hypermedia presentations in a platform-independent manner.

Keywords: Hypermedia, Multimedia, Editing environment, Synchronization, Scheduling, Portability, Heterogeneity, CMIF.

1. INTRODUCTION

Many hardware platforms currently offer exciting multimedia possibilities: full-motion video, hi-fi audio and full-color computer graphics and text can be put together in dazzling displays and demos. Unfortunately, it is often difficult to exploit these possibilities. Part of this difficulty is due to the nature of defining multimedia presentations — making "good" presentations is an art, not a simple mechanical activity. Another, more solvable, part of the problem is that the tools used to create multimedia presentations are usually hard to use and extremely hardware-dependent.

Several paradigms exist for manipulating multimedia documents. One is *scripting*, a method where a program-like description of a presentation is constructed. Like programming, scripting is very useful for small-scale presentations, but editing and manipulating large documents can be cumbersome without well-defined structuring facilities. In addition, the detailed specification of parallel activities is as difficult in most scripting systems as it is in most programming languages. Another common paradigm for structuring events in time is a *timeline*, where the start and end times (relative to some origin) of events are specified by the author. While this makes precise synchronization possible, the obvious disadvantage is that when one wants to replace a piece of a presentation with an equivalent but longer piece (say an alternative video shot), the times of all the following events have to be changed. In both scripting and time-lining, it is difficult to specify the timing requirements between events whose precise duration is variable or unknown till runtime (perhaps due to varying network delays) or because some data is generated at run time.

A run-time characteristic of multimedia is that few presentations have a purely linear structure. Presentations contain interaction points where the user can control what happens next, and when. A useful paradigm to follow is that of *hypertext*, which, when extended to other media than text, is often called *hypermedia* [9]. Unfortunately, rather that offering solutions to the specification problem for multimedia, hypermedia only makes matters more complex.

The Multimedia Kernel Systems project at CWI aims to develop solutions for hypermedia systems that combine the notions of structured documents, flexible run-time scheduling and hyperlink support. We want to make multimedia presentations more portable between platforms and at the same time easier to change. The focus of the project is to attack the problems caused by the use of multimedia in a distributed environment, especially synchronization between data streams coming from different remote sources. We believe that if the author has specified the timing requirements (*constraints*) in a platform-independent way, a distributed system will have a better chance of satisfying those constraints. A summary of our view of distributed, heterogeneous multimedia appears in [6]. The Amsterdam Multimedia Framework described there is the long-term target system for our research.

The "CMIF editor", *CMIFed* in short, is our first concrete result. It uses the CMIF (CWI Multimedia Interchange Format — pronounced see-miff) file format [5]. In the current paper we share our experiences in building and using CMIFed.

The paper contains three major descriptive sections and a conclusion. In section 2 we review the data model which underlies our system. In section 3 we describe CMIFed's user interface. Section 4 gives highlights from its implementation. In the conclusion we discuss related work and future research.

2. DATA MODEL

The data model underlying CMIFed, the CMIF hypermedia model, is a concretisation of the Amsterdam hypermedia model [9]. The latter is an extension of (and a critique on) the Dexter hypertext reference model [7] as well as the original CMIF multimedia model [5]. The (extended) CMIF file format closely follows the CMIF hypermedia model. In this section we briefly review the components of the CMIF model that are necessary for understanding the CMIFed editor/viewer.

The CMIF hypermedia model describes a multimedia document as a tree which specifies the presentation in an abstract, machine-independent way. This specification is created and edited using an authoring system; it is mapped to a particular platform by a viewing system. In our current implementation, CMIFed performs both

ACM Multimedia 93 /6/93/CA, USA

functions, but they could also be done separately (a viewing-only system is currently under construction).

Note that it is possible to 'over-specify' a document by defining something that can not be universally implemented, e.g. color images (which may lose nuances when mapped to a B/W monitor). While some would feel this an error, we consider it a feature: by providing mapping from documents to particular hardware platforms in the viewing system, it is possible to define documents that can be viewed on heterogeneous platforms, thus providing a means to support portable documents.

2.1. Nested presentations

CMIF allows a document, or presentation, to be recursively built from a number of subpresentations. In general, a presentation is either a *composite* presentation (one that contains other, nested presentations) or an *atomic* presentation (one that does not). A CMIF document thus contains a tree whose leaves are atomic presentations. (This is analogous to the subdivision of a book into chapters, sections, etc.)

2.2. Events

An atomic presentation is a collection of *events*. An event is the smallest fragment of media data that is manipulated within the system. It is usually a small fragment of video, audio, image or text data. The model is not concerned with the contents of this data, only with certain media-specific properties of it, such as width, height and/or duration, and the occurrence of *markers* in the data. Markers can be used to attach hyperlinks, and also timing constraints in the case of dynamic media.

Events refer to the actual data via pointers. These pointers can be filenames or some other kind of reference, depending on the storage type (e.g. keys in a database). The use of pointers makes it possible to use the same data multiple times without increasing the storage requirements. The media-specific properties and markers are stored together with the data. Presentation-specific properties are stored with the event. In general, we distinguish a three-level hierarchy here: an *event descriptor* describes the event as it occurs in the presentation (e.g. its channel assignment), a *data descriptor* describes the static properties of the data (e.g. format and dimensions), and the *data file* holds the raw data.

2.3. Channels

A *channel* is an abstraction for a group of properties shared by some events of the same media type. Each event is assigned to exactly one channel. Each channel has a media type, which must match the media type of the events assigned to it. Other properties of a channel depend on its media type; e.g. an audio channel might specify the playback volume, an image channel may specify a screen position, and a text channel might specify a font name. Several channels may have the same media type; the author can use this to further structure the presentation (e.g. there may be two audio channels, one used for background music and one for spoken commentary). Multiple channels for screen-oriented media may be seen as multiple windows, possibly overlapping.

At run-time, channels form a convenient mechanism to choose between several parallel variants of a presentation, e.g. for presentations with spoken or written text in multiple languages, or to provide a sequence of still images as a lower bandwidth alternative for video, to be used on slower workstations. In order to support this, all channels have a flag which can be set to disable all events assigned to it.

2.4. Timing constraints among events

The events that compose an atomic presentation are not just a random collection, they are ordered by timing constraints. Two mechanisms are used to express timing constraints:

- *Parallel and sequential composition*. A first-order synchronization pattern is created by placing the events in a tree whose nodes specify sequential and parallel composition. This takes care of many common timing constraints, such as adding captions to illustrations, presenting several pictures simultaneously in different (sub)windows, or composing a video sequence from a number of shots. Composition can be nested arbitrarily, e.g. to add background music to a sequence of pictures with captions. (This is similar to the subdivision of a section in paragraphs, sentences, etc.)
- *Synchronization arcs*. More precise synchronization between events can be obtained by adding *synchronization arcs* (or *sync arcs*). A *sync arc* is a relation between markers on two events in the same atomic presentation, specifying a desired delay and allowed deviations from the desired delay. For example, when a video fragment and background music are combined in a parallel node of the tree, we may want to delay the start of the music by approximately two seconds. This can be accomplished by adding a sync arc from the start marker of the video event to the start marker of the music event, specifying a delay of two seconds plus or minus 20 percent, say. (This range can be used by the viewer to compensate for platform-specific delays.)

Sync arcs are not powerful enough to model the requirements of lip-sync audio and video, unless one were willing to add markers to all video frames and to the corresponding points in the audio stream, and a corresponding number of sync arcs between them. This form of synchronization is important enough to warrant a special feature: a *continuous sync arc* between markers (usually the beginnings) of two events specifies that the delay range specified in the sync arc is to be maintained for the remaining concurrent duration of the events.

2.5. Hyperlinks

Some multimedia presentations must be *interactive*, i.e. some form of user input can be used to control the path through the presentation or its pace. For this purpose we support hyperlinks. Conforming to the Dexter model [7] we have separate anchors and links. An *anchor* is part of a media data item. Its representation depends on the media type; e.g. it can be a region in an image or a number of characters in a text event. A *link* is a directed connection between two anchors: the source and destination anchors (we currently don't support higher-order links).

The destination of a link may also be a composite node within an atomic presentation, or an entire atomic or composite presentation. Source and destination of a link need not be in the same document (file).

2.6. Attributes

The model supports a general notion of *attributes*. These are used to specify properties like file names, durations, user options and so on.

Presentations, events, channels, composite nodes, sync arcs, anchors, and links all have an attribute list, which is a table mapping names to values. The type of the value depends on the attribute name only. The meaning of an attribute depends on what kind of object it belongs to (e.g. events have different attributes than channels), and may also depend on its media type (e.g. image cropping does not apply to audio events).

Attributes with unrecognized names are ignored. This is for the benefit of other applications than CMIFed which might also manipulate CMIF files (e.g. a platform-specific viewer).

3. USER INTERFACE

In the previous section, we considered the abstract aspects of CMIF. This section discusses the interface provided by CMIFed for authoring CMIF documents. A more complete description of the user interface and examples of how it is used can be found in the companion paper [10].

In general, any authoring/viewing system for multimedia should include a interactive, WYSIWYG interface to the document. Since CMIF documents allow an author to specify presentations that may not be supported on all platforms, the editor/viewer also needs to provide a mapping facility to a particular platform. Unfortunately, these are not easy concepts to combine.

The nature of multimedia data, as well as the CMIF model for multimedia presentations, make it difficult to adhere to the multimedia equivalent of WYSIWYG editing. (Taken literally, this would mean that the time dimension represents itself. “Scrolling” the time dimension would then be equivalent to fast-forward or reverse playing, and selecting a position in time would require playing (part of) the presentation and hitting button at the right moment.) In addition, in order to support portability and ease of authoring, the CMIF model does not require the author to specify the exact timings for events; instead, timing constraints are defined using the presentation’s tree structure and sync arcs.

In order to provide flexible document specification and a previewing facility, CMIFed provides three “views” on a presentation, each highlighting a different aspect of it. These are: the *hierarchy view*, the *channel view*, and a preview (or *player*) view. The author can open and close each view independently of the others, and it is possible to have all views on the screen simultaneously (screen real estate permitting). The three views are described in the following subsections.

3.1. The hierarchy view

CMIFed’s major editing view is the *hierarchy view* (see figure 1). It is used both for viewing the tree of nested presentations, and for viewing the tree of events within an atomic presentation. The tree is not drawn as the usual mathematical tree diagram, but rather as a nested set of boxes representing tree nodes. The root of the tree is the outermost box, and containment is used to indicate ancestry relations. The advantage of this representation is that we can represent parallel and sequential composition by different placement of the boxes: boxes arranged from top to bottom are part of a sequential composition, while boxes placed in a left-to-right arrangement are composed in parallel. Nested boxes are suppressed when they would become too small to contain text. Double clicking on a box zooms in on that box, revealing previously suppressed detail; the path from the root to the zoomed-in box is shown as a stack of buttons and can be used to zoom out again (not shown in the example).

The standard principles of object-oriented user interfaces apply: the user can click on any box to select it, and editing operations applicable to the selected type of box can be selected from a number of menus. There are three groups of commands: commands to insert new nodes, cut and paste commands (which can move or copy entire subtrees), and commands that display additional information about the selected object (such as the full list of attributes). There is also a generic “edit” command which asynchronously invokes an external editor to edit an event’s data. The editing program invoked in this way depends on the media type and the choice of editor can be configured by the author. CMIFed does not have built-in editing modules for each media type, since good editing programs for most media types already exist or are bound to appear soon, and authors will want to continue using their favorite editor when preparing data for CMIFed. CMIFed can read most common data formats and is easily extended to support more.

3.2. The channel view

Another view on a presentation that CMIFed can give the author is the *channel view* (see figure 2). This is a diagram resembling a traditional timeline editor: time flows from top to bottom through the diagram and a number of columns represent the different (logical) channels used by the presentation. Boxes in each column represent events assigned to that channel. The placement and size of a box are indications of the start time and duration of the event. An important difference with a timeline editor, however, is that the times shown by the diagram are not directly specified by the author: they are derived from the timing constraints (and consequently only an approximation of the real times experienced when running the presentation).

The channel view also displays CMIF’s sync arcs and allows the author to create and edit them. Sync arcs are shown as arrows connecting two boxes.

When the user has made an editing change, for example by changing the structure in the hierarchy view or editing a sync arc, the effect of the change is immediately shown in the channel view by adjusting the placement of the affected boxes. (The channel view can’t be used for structural changes itself, since it does not display the composite nodes of the tree.)

The channel view provides a convenient overview of the channels used in a particular presentation (hence its name). A channel is represented as the diamond-shaped “title box” of its column. This box can be selected and commands can be applied to request additional information about a channel, to delete a channel, or to create a new channel. Channels can also be temporarily disabled, e.g. to support alternative versions or to suppress time-consuming operations during previewing.

Finally, the channel view is used to “animate” the execution of a presentation. When a presentation is active in the player, the boxes

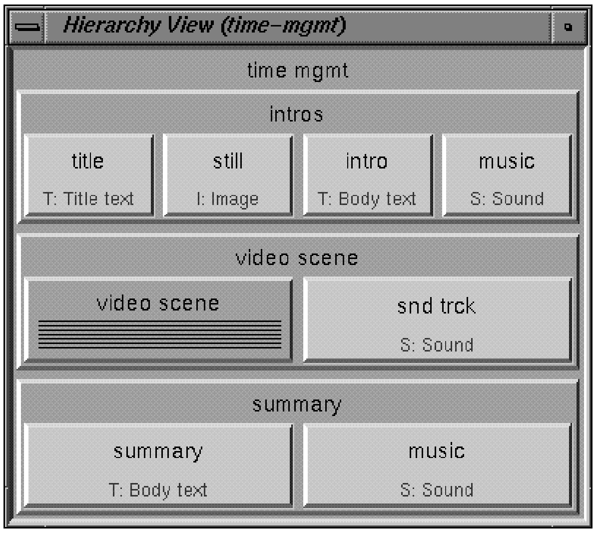

Figure 1. Hierarchy view example

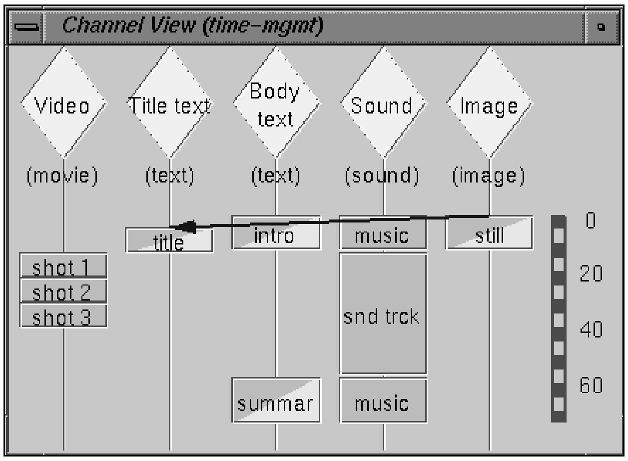

Figure 2. Channel view example

representing current events are highlighted in the channel view. Different highlighting colors are used to indicate different phases in the execution of an event: ready to be armed, arming, armed, and playing (explained further in section 5.2). This is a useful tool for “debugging” a presentation.

3.3. The player

The third view on the presentation, the *player*, shows the effect of mapping the abstract document to a particular platform. The player also allows the author to edit the layout-oriented aspects of a presentation, such as the geometry of the windows used by screen-oriented channels, and for drawing anchors (attachments for hyperlinks) on images.

An example of the player view is shown in figure 3. The player displays a control panel and additional windows for screen-oriented channels. The control panel controls the presentation through buttons similar to those used on CD players: stop, play and pause. It also displays the current and total nominal playing time and has an interface to change user options and to switch channels on or off.

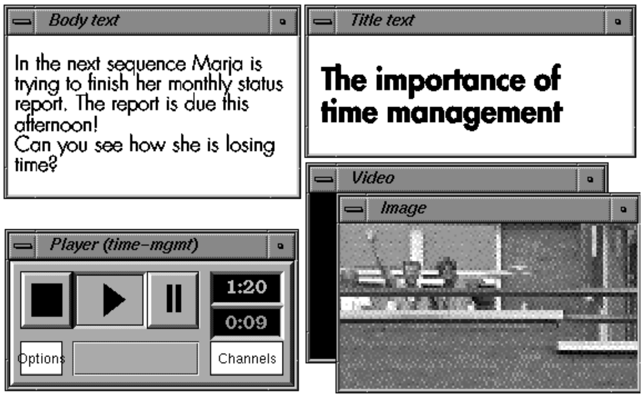

Figure 3. Player example

The player normally shows one window per screen-oriented channel (video, image or text). The window geometries are stored as channel attributes. When the viewer resizes such a window the corresponding attributes are automatically updated.

Window managers tend to add fancy borders and title bars to windows. Often a presentation looks better when it does not pop up a multitude of seemingly randomly placed windows but a single large window containing a number of panels. This can be accommodated by defining a “background” window, in which channel windows are placed as subwindows. The background window can then be dragged around by the user without affecting the relative geometries of its subwindows. (This is not shown in the example.)

While the player’s control panel is the main preview interface, it is also possible to start playing a presentation or part thereof directly from the hierarchy and channel views. This is a very useful previewing facility, since the author can (from the hierarchy view) preview any subtree of a presentation.

4. IMPLEMENTATION

In this section we discuss highlights from CMIFed’s implementation, with an emphasis on the novel aspects of the player implementation.

The programming model used to implement CMIFed is object-oriented and event driven. Interactive objects define *callback methods* that are called from a main event loop when input events (not to be confused with CMIF media events!) relevant to them are detected.

For editing views like the hierarchy and channel views, this model is well-known and we need not explain their implementation in detail. Most of this section is therefore concerned with the implementation of the player, which poses some new problems due to its (soft) real-time nature.

The player is responsible for presenting individual media events in a timely fashion. It has to satisfy the timing constraints specified by the author within the limits of the hardware platform and operating system supporting it. An additional user-oriented requirement is that the application must keep its interactive character at (almost) all times.

4.1. Callback scheduling

In order to maintain the application’s interactive character, the player must use the event dispatching mechanism. For this purpose we have added a timer mechanism to the main event loop and built a standard scheduler queue on top of this. The timer allows us to schedule a single call to a particular callback at a set time in the future. The scheduler queue allows different objects to use this facility independently.

The timer is manipulated to implement the player’s “pause” feature: to pause a presentation, the clock is temporarily stopped, so callbacks that are already queued will be held up. (Unfortunately this is not all that it takes to implement pauses, since some output devices, e.g. sound drivers, have an internal clock that must also be stopped and resumed.) It is also possible to implement slow-motion and fast-forward by changing the clock speed. (Reverse motion would require more work...)

The clock thus runs in a mixture of real and virtual time which we call “nominal time”: it runs in real time multiplied by a factor specified by the user (through the pause and possible slow/fast buttons). It follows that there is no exact relationship between the current position in the presentation and the clock time: wherever the timing constraints allow a variable delay, the real time used up by the delay will differ between successive viewings of the same presentation, and the relative start/end times of events will also differ.

4.2. From timing constraints to a graph of time dependencies

The player uses a simple but effective algorithm to satisfy timing constraints. (This will have to be improved in order to implement maximum delays, which require “foresight” or hard guarantees from the operating system about the time needed to access the data for the next event.) The initial phase of the algorithm is executed when the user requests that a presentation is played and precedes the actual playing. The time it takes is hardly noticeable (but if it were a problem, results of this phase could be precomputed).

The algorithm begins by building a directed graph of timing dependencies. The nodes in this graph correspond to begin and end points of the interior and leaf nodes in the atomic presentation (or subtree thereof) being played, and the edges represent timing relations between two nodes, e.g. A must precede B by at least T seconds. The graph is initialized with edges that represent the timing constraints implied by the tree structure: e.g. in a sequential composition, the end of each child must precede the beginning of the next. Edges are also added to represent the durations of events, gathered by inspection of the data descriptors (or file headers).

The graph is then extended with edges representing the sync arcs present in the presentation. When interior markers in media events are used to attach sync arcs, new nodes are created for these markers, and new edges are added to link them to the begin/end and other markers of the same event. (Sync arcs with an end point outside the subtree being played are ignored.)

Two special nodes are added to the graph: the start and end nodes. The start node is connected with a zero-delay edge to the graph node representing the beginning of the root of the subtree being played, and the corresponding end is similarly connected to the graph’s end node.

4.3. Traversing the time dependency graph

Once this graph has been built the player enters real-time mode to "execute" the presentation. In this mode the graph's edges and nodes are being marked. Initially, the start node is marked. When a node is marked, all its outgoing edges are labeled with the current value of the clock. When the delay that such a labeled edge represents has passed, the edge itself is marked. The marking of edges that correspond to media events is triggered by the driver handling the media; edges corresponding to timing constraints are marked using the scheduler queue. When all of a node's incoming edges have been marked, the node itself is marked. When the graph's end node is marked, the presentation is finished and the player leaves real-time mode. (Deadlocks are possible if there are illegal timing constraints; these can be detected by checking that there is always at least one active driver or one entry in the scheduler queue.)

It is understood that the allowed deviations from the desired delays can be used for graceful degradation of performance, e.g. by stretching the specified range for some delays. (I.e., our "real-time" mode is *soft* real-time.)

The interactive nature of the player is maintained by implementing the marking algorithm as a collection of callbacks that can be entered in the scheduler queue.

Continuous sync arcs (used for e.g. lip-synchronous video and audio) must be implemented differently. The simplest solution, given their likely application, is to special-case them and let the audio and video drivers work it out bilaterally.

4.4. Other uses of the time dependency graph

Using sync arcs one can specify constraints that are impossible to satisfy, e.g. circularities. The graph of time dependencies can be used to detect these. A traversal in simulated time is used by the channel view to calculate the layout of boxes representing events; detection of circularities is a by-product of this traversal. Such a traversal can also detect other problems with the specification, e.g. multiple simultaneous events on the same channel.

4.5. Pre-arming

Because the CPU can work only at a finite speed, sometimes callbacks are called at a later time than scheduled. In the current version of CMIFed all delays are only *minimal* delays, where this is tolerable. However, unnecessary delays in a presentation are still a nuisance, so the player incorporates a mechanism to minimize delays. The key to this mechanism is the use of idle time to work ahead, e.g. to start filling a buffer with data that should be displayed in a moment. We call this "pre-arming" of an event. It should be obvious that the time needed for pre-arming depends both on the media type and on the size and location of the data (which may possibly be stored on a server across the network).

While the presentation is playing, there are two kinds of idle time: when the presentation is waiting for user interaction, or when the next callback in the scheduler queue is some time ahead in the future. In the former case the duration of the idle time is unknown; in the latter case an upper bound is known. We use this upper bound as follows: if an estimate of the time needed by a particular pre-arm action is known, this pre-arm action is only executed if sufficient time remains until the next scheduled event. Initial estimates are calculated based on heuristics involving the data size; when a pre-arm action is executed, the actual time it takes is saved and used as an estimate when the presentation is run again later. When several pre-arm actions are possible, the one is chosen whose results are needed first.

4.5.1 Multi-threaded pre-arming

An alternative method for pre-arming would be to use multiple threads. A lower priority thread can then start working on pre-arm actions while the higher-priority main thread handles active events and the user interface. This will require some subtlety when the main thread needs the results of a pre-arm action that has been begun but not yet finished by the pre-arming thread — possibly the priority of the pre-arming thread can be raised temporarily.

4.5.2 Pre-arming for hyperjumps

When a composite presentation uses hyperjumps extensively, pre-arming is less effective. We are considering an extension to the pre-arm mechanism which can work ahead on likely hyperjumps, e.g. by constructing the timing graph ahead of time and pre-arming the initial events. When a presentation is played several times, statistics can be stored about which hyperjumps are taken most often, to guide the selection of presentations to start working on first.

4.6. Implementation language

The entire CMIF editor is implemented in Python, a very high-level interpreted, extensible object-oriented prototyping language [16]. This language has a number of practical advantages for this application, in particular it has a good interface to the graphics facilities and user interface toolkit on the initial target machine (the SGI Indigo workstation). Python's extensibility (with modules implemented in C) means that it is easy to add new interfaces to system libraries, and this has been used to efficiently handle the data formats and I/O devices needed for audio, image and video processing without losing the advantages of using a very high-level language (e.g. automatic garbage collection and powerful string handling operations, shorter and clearer code, and a much faster edit-run cycle).

The entire application is constructed as a set of classes representing the entities defined by the CMIF model (nodes, sync arcs, channels, etc.) as well as the user interface objects (views, windows, objects, active event managers etc.). Most classes are reusable in the form of a library for other applications that use CMIF presentations, e.g. a stand-alone CMIF player without editing facilities or a tool for creating multimedia management games. The extension modules written in C specifically for CMIFed can also be reused, since they provide only the low-level mechanisms for manipulating multimedia data — all policy decisions are made by code written in Python.

5. CONCLUSION

This section places CMIFed in the context of some related work, discusses future research, and places some concluding remarks.

5.1. Related work

There are many hypertext systems with some form of multimedia support. The Andrew Toolkit [15] and Intermedia [17] are well-known examples. However, these do not support synchronization between different media events: the support for continuous media like audio and video is often restricted to playing an audio or video clip when the user clicks on a button.

Multimedia systems addressing the issue of synchronization are less common, but some exist, e.g. Firefly [4] and Videobook [14]. Firefly shares some design goals with CMIFed, but uses the equivalent of sync arcs exclusively to specify synchronization, making simple composition tasks tedious. The Videobook system, while using a highly visual metaphor, is really a script-based timeline editing system. It is also weak in its higher-level composition facilities. Other systems using composition and/or sync arcs to create synchronized multimedia presentations (without hyperlinks) are described in [2] and [8].

Commercial systems addressing synchronization are generally of the timeline or scripting variety, e.g. [12]. Even though some of these systems have been ported between common platforms (e.g. Apple's QuickTime [1] is now also available under Microsoft Windows), they were never designed to support distributed systems.

A relevant international standard is HyTime [11], see also [13]. HyTime by itself does not specify a model, but supplies the syntax to describe different models, since it is intended to be used for a much wider range of applications. This generic language is powerful enough that a specific instance of it (a "HyTime DTD") could conceivably replace the current concrete CMIF language. The creation of such a DTD would involve translating the model described in section 2 into concrete HyTime tags. Whether this is worth the additional parsing complexity depends on the uptake of the standard.

5.2. Future research

The above sections have already suggested many possible topics for future research. Apart from the obvious improvements to the user interface and porting the system to other platforms, we intend to attack a variety of problems that we have experienced during the construction and use of CMIFed:

- Some constraints can only be satisfied by stretching or shrinking some media data, e.g. by playing a video sequence somewhat slower or faster, or by truncation or repetition of material. Whether this is possible is media-specific and platform-specific (e.g. some audio hardware only supports a few playback rates), but in any case the author must be able to specify what is acceptable. In the current version of CMIFed this is not supported; such constraints will cause parts of the presentation to be "late". We are considering the incorporation of ideas from [4], which suggests stretching and shrinking, and [3], which suggests several other ways of adapting the duration of media items to constraints. (This issue is closely related to a more general specification of sync arcs, using minimum, ideal and maximum delays.)
- There are several possible interpretations of what should happen when a link is followed: should the current presentation be canceled, suspended or continued? Currently we always cancel the active atomic presentation and start the new one in its place. In an extended version of the model, the author may specify what should happen when the link is created (including leaving the choice to the user). See [8] for ideas.
- Using a worst-case model of the run-time delays on a specific hardware platform (e.g. disk seek delays, network characteristics and CPU speed) one may verify whether a particular specification is compatible with that platform.
- CMIF documents are currently static: the author defines what options are open to the user at each point. There is however an important range of applications where user input is used to generate a database query whose results will be presented in hypermedia form. We intend to work on the automatic conversion of query results into new ad-hoc CMIF presentations.
- Using the Amsterdam Multimedia Framework [6] as a guideline, we intend to construct a distributed version of the CMIFed player which distributes knowledge about timing constraints to remote components in order to help satisfying them. This includes providing alternative representations of media data with different resource usage and corresponding different presentational aspects.

5.3. Concluding remarks

CMIFed has been developed during the past two years and has been used to create several example presentations. While only a first step in the right direction, building and using CMIFed has taught us many lessons on all aspects of hypermedia systems, ranging from thoughts about the inadequacy of current operating systems to the development of new data models and editing paradigms, and we are excited about the direction in which the next steps will take us.

ACKNOWLEDGMENTS

The authors would like to thank Robert van Liere and Dik Winter for their work on the initial implementation and for many fruitful design discussions. We are grateful to Lynda Hardman, who successfully posed as a "naive" user, for her feedback, constructive criticism, and for her patience even when the system was constantly changing.

REFERENCES

[1] Apple Computer, "QuickTime", Cupertino, CA.
[2] *Gerold Blakowski, Jens Hübel and Ulrike Langrehr*, "Tools for Specifying and Executing Synchronized Multimedia Presentations", 2nd International Workshop on Network and OS Support for Digital Audio/Video, Heidelberg, Germany, Nov. 18-19, 1991.
[3] *Monica Bordegoni*, "Multimedia in Views", CWI Report number CS-R9263.
[4] *M. Cecelia Buchanan and Polle T. Zellweger*, "Specifying Temporal Behavior in Hypermedia Documents", ECHT '92 (Proceedings of the ACM Conference on Hypertext), Milano, Italy Nov 30-Dec 4 1992, 262-271.
[5] *Dick C. A. Bulterman, Guido van Rossum and Robert van Liere*, "A Structure for Transportable, Dynamic Multimedia Documents", USENIX conference June 1991 Nashville TN, 137-155.
[6] *Dick C. A. Bulterman*, "Synchronization of Multi-Sourced Multimedia Data for Heterogeneous Target Systems", 3rd International Workshop on Network and OS Support for Digital Audio/Video, San Diego, 1992.
[7] *Frank Halasz and Mayer Schwartz*, "The Dexter Hypertext Reference Model", NIST Hypertext Standardization Workshop, Gaithersburg, MD, January 16-18 1990.
[8] *Rei Hamakawa, Hidekazu Sakagami and Jun Rekimoto*, "Audio and Video Extensions to Graphical User Interface Toolkits", 3rd International Workshop on Network and OS Support for Digital Audio/Video, San Diego, 1992.
[9] *Lynda Hardman, Guido van Rossum*, *Dick C. A. Bulterman*, "The Amsterdam Hypermedia Model: extending hypertext to support *real* multimedia", Hypermedia, May 1993, 5(1).
[10] *Lynda Hardman, Dick C. A. Bulterman, Guido van Rossum*, "Structured Multimedia Authoring", Proceedings of ACM Multimedia '93, Anaheim, Auust 1-6 1993.
[11] International Standard Organization, "Hypermedia/Time-based structuring language", ISO 10744, 1992.
[12] MacroMind, "Director version 2.0", 1990 (dynamic media authoring tool for the Apple Macintosh).
[13] *Steven R. Newcomb, Neill A. Kipp and Victoria T. Newcomb*, "'HyTime' the Hypermedia/Time-based Document Structuring Language", Communications of the ACM, November 1991, 34 (11) 67-83.
[14] *Ryuichi Ogawa, Hiroaki Harada and Asao Kaneko*, "Scenario-based Hypermedia: A Model and a System", ECHT '90 (European Conference on Hypertext), November 1990, INRIA France, 38-51.
[15] *Palay, et al.*, "The Andrew Toolkit: an Overview", Proceedings of the USENIX Technical Conference, February 1988.
[16] *Guido van Rossum and Jelke de Boer*, "Interactively Testing Remote Servers Using the Python Programming Language", CWI Quarterly, December 1991, 4 (4) 283-303.
[17] *N. Yankelovich, B. Hahn, N. Meyrowitz and S. Drucker*, "Intermedia: The concept and construction of a seamless information environment", IEEE Computer, January 1988, 21(1) 81-96.