

A FRAMEWORK FOR EVENT-DRIVEN HYPERMEDIA PRESENTATION SYSTEMS

ROGÉRIO FERREIRA RODRIGUES, LUIZ FERNANDO GOMES SOARES

*Laboratório TeleMídia – Departamento de Informática – PUC-Rio
R. Marquês de São Vicente, 225 – Gávea – 22453-900, Rio de Janeiro, RJ, Brazil
E-mail: rogerio@telemidia.puc-rio.br, lfgs@inf.puc-rio.br*

LEANDRO MARQUES RODRIGUES

*Embratel – Rua Camerino, 96, sala 307 – 20080-010, Rio de Janeiro, RJ, Brazil
E-mail: lmrodri@embratel.com.br*

This paper proposes a modular organization for hypermedia presentation systems. In this context, an adaptable framework is presented for helping the design of hyperdocument presentation tools. An interface for exchanging messages between the document execution controller (formatter) and presentation tools is specified. This interface allows a simple control of temporal and spatial synchronization relationships among objects and internal regions of objects, independent of the presentation tool playing these objects. This facility will allow different presentation tools to cooperate among themselves. The model also foresees the incorporation of existent media players into hypermedia presentation systems through the design of adapter modules. As an example, this paper also describes the implementation of adapters for Java Media Framework (JMF) players. This implementation is used to validate our proposal and to identify common aspects that can be reused in new presentation tool implementations. Finally, it is important to mention that the framework proposed is independent of the hypermedia conceptual model semantics considered by presentation tools and formatters.

1 Introduction

Hypermedia presentation systems must be able to show different media type objects and should try to assure that the specified relationships among these objects are respected. A modular design of this kind of system consists in separating the presentation tools (content players) from the kernel element responsible for presentation control, usually named *hypermedia formatter* [4, 16].

This approach is mainly necessary on systems that present documents based on hypermedia models with a rich expression power, such as those of Madeus [10], NCM [15] and SMIL [18]. These models allow the specification of adaptative documents. Madeus and NCM permit specifying flexible duration for object presentation, offering basis for a document elastic time computation [2, 14]. NCM and SMIL allow defining documents whose presentation can be chosen based on platform characteristics (e.g. bandwidth, available devices, etc.) and on the user skill and preferences (e.g. level of knowledge, language, etc.). Since in all cases the adaptability may require the implementation of complex algorithms dependent on a global view of document presentation, it is reasonable to have a special module,

apart from the presentation tools, dedicated for these tasks. Besides elastic time computation and presentation characteristic selection, content prefetch, temporal and spatial consistency check, relationship evaluation and action scheduling are some other task examples suitable to be assigned to hypermedia formatters.

In general, the interface between presentation tools and formatters should standardize how tools notify the formatter of certain occurrences during an object presentation. For example: that some content-region (*anchor*) presentation has finished, or that an anchor was selected, or that the object position on the screen has changed, etc. In addition, the interface should also specify how a formatter requests presentation tools to perform actions over the objects, such as play, pause, change the bit rate, increase the sound level, etc.

Standardizing the interface for message exchange between the formatter and presentation tools has some important advantages. First, it provides interoperability between designers of presentation tools and formatters, which will be able to work independently. Second, it offers support for the implementation of presentation relationships among objects played on different tools.

Figure 1 suggests a modular organization for hypermedia presentation systems. Obviously, it may be interesting to use existent media players with particular interfaces not compatible with the one required by the formatter, demanding the development of special modules to make the necessary adaptations. In this case, the presentation tool will be constituted of an adapter and a player.

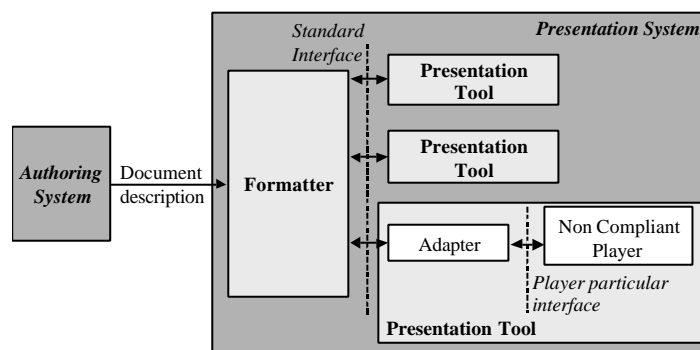


Figure 1 – Hypermedia presentation systems with modular organization

This paper proposes an adaptable framework for hypermedia presentation systems, which aims to allow formatters to easily control temporal and spatial synchronization relationships among objects, independent of the tool presenting them. However, it is important to mention that this paper will not focus on issues related to the internal development of hypermedia formatters, which can be found in [2, 4, 14]. The framework adaptability resides in the possibility of defining new types of actions and events to the presentation system through simple extensions

and in its independence from the hypermedia conceptual model semantics, which will be confined in the formatter and presentation tool implementations.

Some ideas presented can be found in the Sun Java Media Framework (JMF) [17]. This framework offers a basic structure for continuous media player implementations. Nevertheless, some important aspects of JMF API are not suitable for hypermedia presentation tools. In order to bypass this problem, adapters for the JMF players have been developed, as an example of our proposed framework application. The adapter implementations have been important not only to validate the proposed framework, but also to identify elements that could be reused in other presentation tool implementations.

The paper is organized as follows. Section 2 describes the main features usually found in hypermedia document models, which are important for the design of formatters and presentation tools. Section 3 presents the framework. Section 4 describes a framework instantiation example, highlighting its integration with JMF API. Section 5 discusses related work and Section 6 presents some final remarks and future work.

2 Hypermedia models with synchronization relations

Hypermedia models usually define documents as a set of basic elements, the *data objects* (or *nodes*), and a set of relationships among these elements, generally implemented by *links* or *compositions*¹.

The main attribute of a data object is its content. A data object can represent an atomic or a composite object. The atomic object, commonly denoted *media object*, can be specialized in subclasses like text, image, audio and video. A composite object, or simply a *composition*, permits grouping different data objects, atomic or composite, offering support for a structured document specification. It is important to note that presenting a composition may have different purposes [16]. For instance, it is usual to have a command to present a composition, which results in the presentation of their components, using the relation semantic embedded in the composition. As an example, presenting a parallel composition may mean playing its components at the same time. However, users might wish to view the document map, that is, the composition structure. In this case, the composition presentation acts as an ordinary media object presentation.

An important characteristic of hypermedia document models is how presentation parameters are specified for each data object, such as duration, screen position, background color and the presentation tool to be used. Some models define these parameters in specific entities, separated from the content and logical structure of the document. This is a very interesting reuse approach. First, because it allows

¹ Parallel and sequential SMIL elements are examples of compositions that define presentation relationships among their components.

applying the same presentation pattern to different data objects. And second, because it makes possible the exhibition of the same object with different presentation characteristics. We will call *descriptor* the entity that contains the data object presentation parameters.

Another useful feature of hypermedia models is allowing the specification of several alternative descriptors for an object, driven by user preferences, goals and knowledge, and also by features of the platform where the document is presented. As a result, a formatter can adapt the document presentation, selecting the more appropriate descriptor for each data object. Furthermore, it is desirable that descriptors can be chosen depending on the user navigation. Therefore, it is interesting to allow descriptors to be defined either as data object attributes or as relationship attributes. From the aggregation of a data object and a descriptor results the object that should be given by the formatter to the presentation tool, here called *representation object*.

How relationships, mainly those defined by links, are specified results on important hypermedia issues. For example, HTML documents have links embedded in their content, causing significant problems [1, 3, 12]. As a consequence, several hypermedia models separate relationships from object content [1, 5, 6, 7, 15, 18]. In these models, links are defined as first-order entities, defining relationships among representation object *events*.

An event specifies an occurrence in time that results from some action over a representation object attribute or content region (an *anchor*)². Common examples are the user selection of an anchor (*selection event*), the presentation of an anchor (*presentation event*) and the modification of an attribute value (*attribution event*). The exact notion of content region depends on the data object type. For instance, a set of samples could be a content region of an audio node, while a set of characters could be a text node content region.

Events have states and their states, as well as their state transitions, will be used in relationship definitions. Let us take a presentation event as an example, whose state machine is illustrated in Figure 2. Initially, it is in the *sleeping* state. It goes to and stays in the *preparing* state while some prefetching procedure of its information units is executed. At the end of the procedure, the event goes to the *prepared* state. At the beginning of the content region presentation it goes to the *occurring* state. If the presentation is temporarily suspended, the event stays in the *paused* state, while this situation lasts. If the presentation is abruptly finished, the event goes to the *aborted* state and immediately after it comes back to the *prepared* state. At the end of the presentation, the event comes back to the *prepared* state. Two situations can cause the transition from the *occurring* to the *prepared* state: the natural end of the presentation duration, or an explicit command to finish the exhibition. Event

² The set of attributes of a representation object is the union of the data object attributes and the descriptor attributes. The representation object content may be the data object content itself (or a version/copy of it).

duration can be intrinsic to the object content region or explicitly defined by authors, preferably as a descriptor attribute.

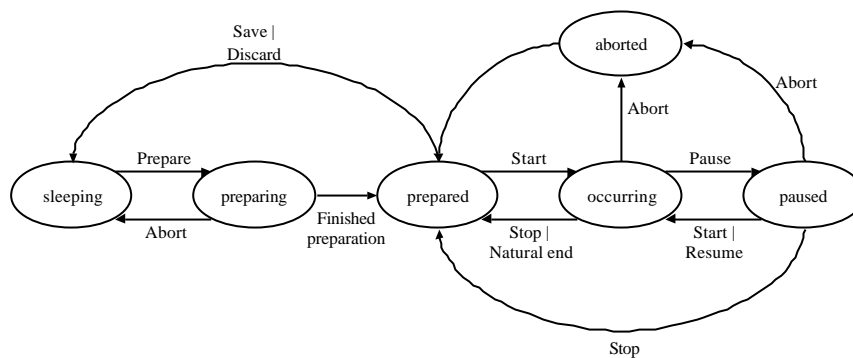


Figure 2 – Event state machine

It is important to note two aspects related to the state machine definition presented. First, there is one state machine for each representation object event. Thus, the presentation of a representation object as a whole is only a particular case of event state machine. The other aspect is that, depending on the hypermedia model, the event state machines can be simpler than that of Figure 2. For instance, in some models presentation events can be restricted to *prepared*, *occurring* and *paused* states; selection events (in almost all models) have only *prepared* and *occurring* as possible states.

Constraint and causality are the two main kinds of relations that can be defined in a hyperdocument specification. *Constraint relations* specify rules that should be respected during document execution (e.g.: if two objects are presented, they should finish their presentation at the same time, without clipping their content). In some cases, formatters will implement compile time algorithms in order to make presentation adjustments, for instance, modifying object duration. However, in order to maintain the constraints and, at the same time, to take into account presentation unpredictable occurrences (communication system delay, user interaction, etc.), it may be necessary that the formatter and presentation tools could interact at runtime. For example, the formatter might send a command to change a video frame rate due to transmission delays, in order to maintain a specified constraint.

Causal relations define conditions over events (usually over their states or state transitions) that, when satisfied, should fire actions on other (eventually the same) document events. These actions will cause new transitions on the event state machines. The following examples show some possible hypermedia relationships that could be defined, exploring the concepts presented.

Example 1: Let us assume a document containing an audio A_1 and a video V_1 . In this document, the audio should be presented during a specific part of V_1 , identified

by presentation event ev_1 . Let us consider that event ea_1 represents the presentation of A_1 as a whole. Figure 3 illustrates the desirable synchronization and how this would be defined using two causal links.

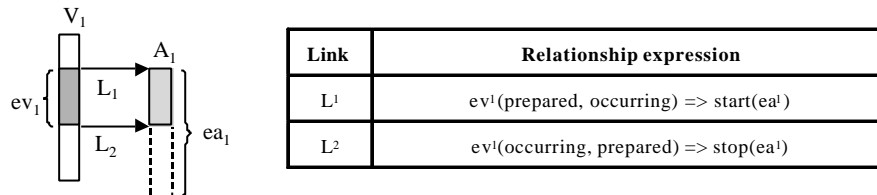


Figure 3 – Temporal synchronization with causal links

Example 2: Let us suppose a sequential composition containing two video clips (V_1 e V_2) that should be presented in this order. Events ev_1 and ev_2 represent the presentation of each video clip. Figure 4 shows the composition and how the relationship among its components would be expressed by event causality.

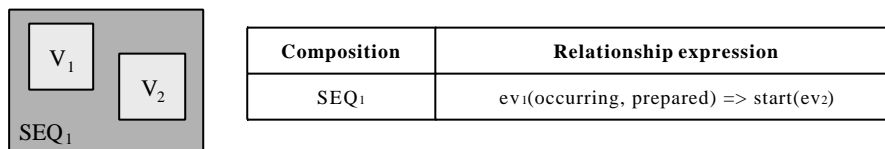


Figure 4 – Temporal synchronization with composition

Example 3: Let us now consider two data objects: a map of Brazil I_1 and a text T_1 containing information about the Rio de Janeiro state. The desired relationship specifies the presentation of T_1 with 200 x 150 pixels size, when a user clicks over the Rio de Janeiro area in the map. This relationship could be defined among a selection event ei_1 , representing the Rio region in the map, and a presentation event et_1 , corresponding to T_1 exhibition. The spatial presentation parameters could be defined in a descriptor associated to the relationship target. Figure 5 illustrates the relationship described and the relationship specification.

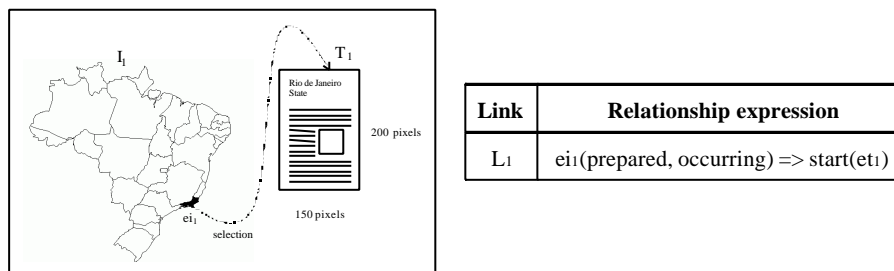


Figure 5 – Spatio-temporal synchronization specified with causal link

According to the discussion of this section, in order to respect the hyperdocument component relationships during a presentation, there must be a well-defined interface between the formatter and the presentation tools. Next section proposes a framework for standardizing this integration.

3 Framework for hypermedia presentation systems

Presentation tools take care of playing document representation objects. Usually, a presentation tool provides a user interface composed by a visual area for presenting the content data (if the media is a visible one) and a controller component with a set of elements (buttons, for example) for playing control, as exemplified in Figure 6. The controller component may allow executing actions like to start, to finish, to pause and to resume a continuous media presentation.



Figure 6 – Examples of presentation tool visual interface

A hypermedia presentation tool has two important tasks. It should run actions requested by the formatter and also notify the formatter of transitions occurred on event state machines that it is charged to monitor. As already mentioned, we propose an interface that aims to allow an independent development of formatters and presentation tools and to offer an integrated environment for hypermedia objects playing. We are going to use an object-oriented modeling approach, following the UML notation [13], to describe our framework. The diagram in Figure 7 brings the main information on the most important interfaces and classes for designing presentation tools integrated to hypermedia formatters.

All presentation tools must implement the *HF_PresentationTool* interface, where the basic methods offered to formatters are defined. The *initialize* method receives, as parameters, the representation object to be played and the event list to be monitored. Presentation tools should discover the correspondent object anchor for each event, to be able to control the corresponding event state machines. The *prepare*, *start*, *stop*, *pause*, *resume* and *abort* methods receive, as parameter, a

presentation event contained in the event list passed to the *initialize* method. These methods will be used to cause transitions in the event state machines.

When a new representation object has to be presented, the formatter instantiates an object that implements *HF_PresentationTool* and calls its *initialize* method, setting the necessities parameters. Then, the formatter calls *prepare*, usually passing as argument the whole object content. At this moment, if the representation object contains a continuous media content, it may be interesting that the presentation tool start to fetch the content data that will guarantee the presentation beginning as soon as the *start* method is invoked. For example, if the representation object is a video, the presentation tool will prefetch its initial frames.

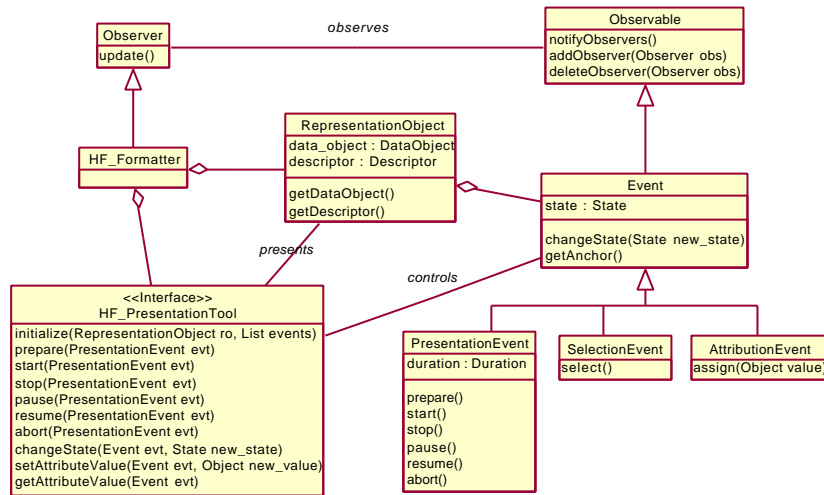


Figure 7 – Class hierarchy for presentation tools integration

The *setAttributeValue* and *getAttributeValue* methods allow a formatter to modify and to consult, respectively, a representation object attribute value. For continuous media, an important utility of *setAttributeValue* is to allow formatters to increase or to decrease the presentation rate, in order to try to ensure specified constraint relations.

We can note that the communication mechanism is very simple in the direction from the formatter to the presentation tool. However, it is important that each tool could understand the information describing representation objects and their events, such as object content, screen position, content region corresponding to each event and the type of each event (presentation, selection, etc.).

In the opposite direction, that is, from presentation tools to the formatter, the communication is based on the Observer design pattern [8]. This pattern specifies that every time the state of an *Observable* object changes, it notifies all objects that

have been registered as its observers (objects from the *Observer* class). An observer perceives this notification through a call to its update method. The formatter must behave as an event observer, being notified whenever any document event has its state changed. When this state transition occurs, the formatter fires all causal relationship actions whose conditions become true. For example, suppose video V_1 contained in the sequential composition of Example 2 (Section 2). When it finishes playing, its associated presentation event changes from the occurring to the prepared state. This transition results in a notification to the formatter, which will realize that the relationship condition is satisfied and then ask for the start of V_2 presentation. If the formatter implements algorithms for document presentation adjustments during runtime [14], it may also register itself to observe condition events in constraint relationships.

The framework presented in this section is independent of the hypermedia model semantics that the formatter and presentation tools rely on. It can be easily adapted to specific models. First, it can be extended by the creation of new event types. For example, a class to deal with the mouse over event or to deal with animation could be defined. In order to define a new kind of event, it is not necessary to modify the structure and relationships presented in Figure 7. It is enough that presentation tools recognize the new event semantics and its state machine. Another flexibility resides in the possibility of modifying the event state machine without changing the class diagram. The *changeState* method defined in the *Event* class and in the *HF_PresentationTool* interface allows the definition of new states and transitions. Evidently, a presentation tool implementation that actually controls the object and its event exhibition will need to know this new state machine.

4 Framework instantiation for Java Media Framework players

In order to validate the proposed framework, presentation tools for text, image, audio and video media types were implemented (in Java) and integrated with the HyperProp system [16]. The HyperProp formatter handles NCM [15] and SMIL [18] hypermedia documents. We chose to use JMF players in the implementation of HyperProp continuous media presentation tools due to the wide range of existent implementations and the JMF similarity with our proposal. This approach allowed the identification of interesting aspects that can be reused in future implementations of presentation tools. This section mainly focuses on the design of presentation tool adapters that take care of doing the necessary conversions between the interface of both frameworks.

As can be seen in the class diagram of Figure 8, the presentation tools for discrete media, *HF_TextPresentationTool* and *HF_ImagePresentationTool*, directly implements the *HF_PresentationTool* interface, presented in the previous section. On the other hand, for continuous media, the *HF_PresentationTool* interface was not directly implemented by the tools, modeled by the *HF_JmfAudioPresentationTool*

and *HF_JmfVideoPresentationTool* classes. As these tools use the JMF players, it was necessary to develop specific adapters for each one of them (*HF_JmfAudioPlayerAdapter* e *HF_JmfVideoPlayerAdapter*). These JMF adapter classes inherit from the same class, called *HF_JmfPlayerAdapter* (which actually implements the *HF_PresentationTool* interface), where the common JMF adapter characteristics are defined.

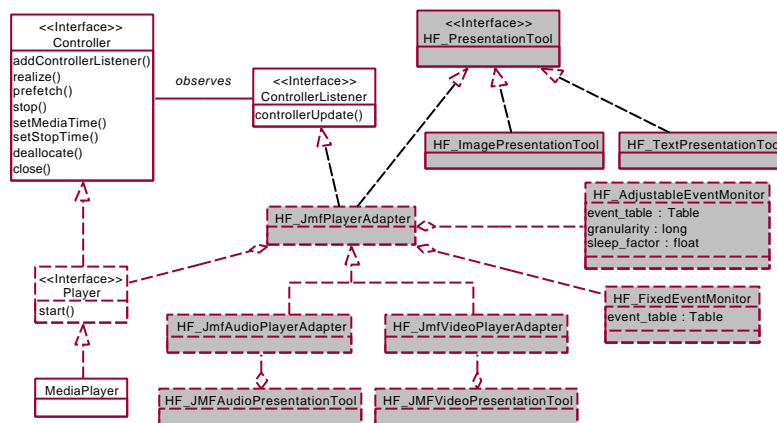


Figure 8 – Class diagram of JMF presentation tools

In the figure, classes and interfaces with white background are part of the Sun framework. The *Player* interface is implemented by JMF players (classes that inherit from *MediaPlayer*). A *Player* is responsible for processing a continuous media data stream and presenting it in the output devices, eventually offering some graphical components for user interaction. Every *Player* extends the *Controller* interface. A *Controller* defines a state machine, presented in Figure 9, which describes the JMF Player behavior [17] and, as a consequence, the presentation event of the entire media object that the player presents. Transitions in the state machine are notified to observers of the JMF player, registered through calls to the *addControllerListener* method. Every observer must be an instance of a class that implements the *ControllerListener* interface, since the notifications are made through the *controllerUpdate* method calls. Therefore, the *HF_JmfPlayerAdapter* class implements the *ControllerListener* interface, as it needs to monitor JMF player states.

Following Figure 9, a JMF player is created in the *unrealized* state. When the *realize* method is called, the player goes to the *realizing* state, starting the allocation of necessary resources. When this process ends, the player is aware of the media type to be presented and its state becomes *realized*. The *prefetch* method puts the player in the *prefetching* state, where it starts fetching the object content. When the player reaches the *prefetched* state, it is ready to play the media content. The *start*

method changes the player to the *started* state and effectively begins the presentation. A *started* JMF player returns to the *prefetched* state when its *stop* method is called, or when it reaches the end of the presentation or when the data transmission is interrupted. A player notifies its observers of state transitions, of any attribute value changing (e.g. duration and presentation rate modifications) and of its eventual destruction.

Besides the methods that cause state transitions (*realize*, *prefetch*, *start* and *stop*), the JMF player offers two other important methods: *setMediaTime*, to change the current position in the media stream presentation; and *setStopTime*, to set the position in the media stream where the presentation must be interrupted³.

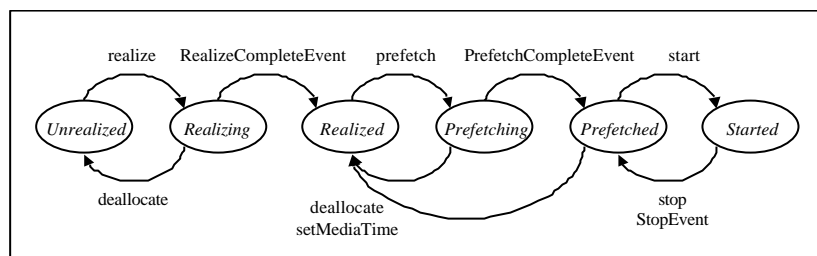


Figure 9 –JMF player state machine

As it can be observed, the presentation event state machine (Figure 2) and the JMF player state machine (Figure 9) have many similarities. Thus, the *HF_PresentationTool* and *Controller* methods are also similar. However, there is an important distinction between the sets of methods. *HF_PresentationTool* offers a finer granularity, since its methods are applied to representation object events. Actually, the adapter maintains a state machine for each representation object event. More restricted, the JMF player interface only allows its methods to be applied to the object as a whole. Therefore, the JMF player maintains only one state machine. One of the adapter roles is the conversion between the methods offered by both interfaces, as summarized in Table 1.

The adapter initialization reflects in the JMF player initial preparation, represented by a call to its *realize* method. The *prepare* action causes three player method calls. The *setMediaTime* method sets the media stream initial position, which corresponds to the start time of the presentation event passed as parameter. Method *setStopTime* sets the presentation finish position, which corresponds to the event stop time. If the event is related to the object as a whole and the duration is not specified, method *setStopTime* will not be called, and the presentation will continue until its natural end or until an explicit interruption. After the definition of

³ Actually, the *setMediaTime*, *setStopTime* and *stop* methods are defined in the *Clock* interface, from which the *Controller* interface inherits. However, this and some other JMF interfaces and classes were not considered to avoid making diagram overcrowded.

the start and end positions, the *prefetch* method is called to start the content fetching. The *start* action results in a call to the JMF player *start* method, while the *stop* action causes a call to the player *stop* method, followed by the release of the allocated resources. The *pause* action causes simply a call to the player *stop* method, without changing the current presentation point. The *resume* action restarts the presentation from the point it was interrupted, through a new call to the player *start* method. The *abort* action finishes the presentation, releasing the resources used by the player, similarly to the *stop* action.

Table 1 – Conversion between actions requested by formatter and methods called in JMF players

<i>HF_PresentationTool</i> methods	Related methods in the JMF player
<i>initialize(rep_obj, event_list)</i>	<i>realize()</i>
<i>prepare(E)</i>	<i>setMediaTime(T_i) + setStopTime(T_f) + prefetch</i>
<i>start(E)</i>	<i>start()</i>
<i>stop(E)</i>	<i>stop() + deallocate() + close()</i>
<i>pause(E)</i>	<i>stop()</i>
<i>resume(E)</i>	<i>start()</i>
<i>abort(E)</i>	<i>stop() + deallocate() + close()</i>

T_i = start time for the presentation of E, T_f = end time for the presentation of E

In the reverse communication direction, the adapter is responsible for receiving the JMF player notifications and translating them to the event state machine model of the formatter. The main types of JMF player notification are *PrefetchCompleteEvent*, *StartEvent*, *StopByRequestEvent*, *StopAtTimeEvent*, *EndOfMediaEvent* and *ControllerClosedEvent* [17]. Table 2 presents the mapping of each JMF player notification to the corresponding transition(s) in the presentation event state machine.

Table 2 – Conversion between JMF player notifications and presentation event state transitions

JMF player notifications	Notification meaning	Transitions in the presentation event state machine
<i>PrefetchCompleteEvent</i>	End of prefetching	<i>preparing</i> → <i>prepared</i>
<i>StartEvent</i>	Start of presentation	<i>prepared</i> <i>paused</i> → <i>occurring</i>
<i>StopByRequestEvent</i> as a reply to <i>stop</i> as a reply to <i>pause</i> as a reply to <i>abort</i>	Pause, stop or abortion of presentation	<i>occurring</i> <i>paused</i> → <i>prepared</i> <i>occurring</i> → <i>paused</i> <i>occurring</i> <i>paused</i> → <i>aborted</i> → <i>prepared</i>
<i>StopAtTimeEvent</i>	End of presentation set time	<i>occurring</i> → <i>prepared</i>
<i>EndOfMediaEvent</i>	Natural end of presentation	<i>occurring</i> → <i>prepared</i>
<i>ControllerClosedEvent</i>	Player closing	<i>occurring</i> <i>paused</i> → <i>aborted</i> → <i>prepared</i>

4.1 Monitoring presentation events

During a presentation event defined by a region *R*, other presentation events defined by sub-regions of *R* can occur. Continuous media presentation tools should be able

to signalize each beginning and end of the external and internal presentation events. This feature allows the definition of synchronization relationships among temporal sub-regions of media objects, as illustrated in Example 1 of Section 2. In the example, the whole video presentation may correspond to the external event while ev_1 may correspond to an internal event. Nevertheless, this functionality, which is easily offered by our proposed interface, is not trivial to be obtained through the JMF player API. Hence, this adaptation becomes an important task for the *HF_JmfPlayerAdapter* class. Indeed, it is this adaptation that transforms the unique JMF state machine in the several event state machines of a representation object presentation.

When initialized, the adapter receives a list of events that have to be controlled during the presentation. There are two modes to specify times delimiting the interval duration of presentation event regions. Region delimiters may be explicitly defined by fixed times relative to the start time of the object presentation (*fixed events*), without considering changes in the media exhibition rate. In this case, the adapter only needs to be aware of the system clock. On the other hand, interval edge instants may be associated to media content information units (*adjustable events*), like video frames or audio samples. In this case, the adapter must have an exact control of the content presentation rate, since jitters may happen.

The *HF_JmfPlayerAdapter* class has two monitors used as auxiliaries in the control of presentation events. The *non-adjustable monitor* (*HF_FixedEventMonitor*) handles the events whose regions are defined with fixed instant delimiters. On the other hand, the *adjustable monitor* (*HF_AdjustableEventMintor*) controls the events whose regions are associated to content information units. Both monitors create event tables with each entry containing the following information: the event identification, the type of transition (beginning or end of the related region presentation), the expected time for the transition and whether the transition is enabled or not. The event table is sorted by transition expected times and the choice for enabling or disabling each entry depends on the presentation start point, as next explained.

The adapter can be asked for starting the object presentation from any point of its content, that is, from any of its presentation events. Depending on the start position, some event state machines should not be considered. Table 3 summarizes the rules adopted by JMF adapter monitors, which establish the presentation event transitions that are enabled in the monitor event table. These rules are based on the interval relation between the event requested to be presented (event α) and any other object presentation event (event β). We suppose $[T_{\alpha i}, T_{\alpha f}]$ and $[T_{\beta i}, T_{\beta f}]$ as the intervals respectively specified for α and β .

Event monitors are implemented as threads started by adapters. The non-adjustable monitor has a simple functioning. When started, the monitor gets the expected time of the first enabled entry in the table and it goes to sleep until then. When the monitor wakes up, it simply changes the event state *occurring* for

beginning transition and *prepared* for end transition), looks for the next enabled table entry and returns to sleep until this entry expected time.

Table 3 – Rules for enabling presentation events

Temporal Relation	Relation Illustration	Rule
$T_{\alpha i} > T_{\beta i}$ OR $T_{\alpha f} \leq T_{\beta f}$		The beginning and the end transitions of α will remain enabled. Event β edge transitions will be disabled.
$T_{\alpha i} \leq T_{\beta i} < T_{\alpha f}$ AND $T_{\alpha f} < T_{\beta f}$		All transitions will remain enabled. However, the end of α will cut off β presentation. Thus, event β will be aborted.
$T_{\alpha i} \leq T_{\beta i}$ AND $T_{\alpha f} \geq T_{\beta f}$		All transitions will remain enabled and will be notified at their respective expected times.

The adjustable monitor has a little more work to do. The object content presentation rate can suffer variations caused by unpredictable factors such as communication and operational system delays. This makes impossible for the monitor to exactly preview its sleep time duration until the next table entry expected time. Moreover, the JMF player API does not offer a way for its observers to be notified when specified exhibition internal points are reached. Thus, the monitor has to poll the JMF player in order to control adjustable event presentations. It does the polling making successive calls to the JMF player *getMediaTime* method. If this check is done with a high frequency, there will be a wasting of memory and CPU resources. Otherwise, with a low check frequency, imprecision on notification instants may happen.

Aiming to find a customizable and balanced solution, the adjustable monitor has two other attributes, *granularity* and *sleep-factor*, which can be set up. *Granularity* defines the monitor sample rate, while *sleep-factor* is an adjustment factor that can vary from 0 to 1. When the monitor finds an enabled entry, it goes to sleep by an interval calculated as the maximum value between the *granularity* and the expression $[(t_{next} - t_{now}) \times sleep_factor]$. In the expression, t_{next} is the next enabled transition expected instant and t_{now} is the current playing time. Therefore, if *sleep-factor* is 1 and $[t_{next} - t_{now}]$ is lesser than the *granularity* value, the monitor will sleep during the exactly interval duration previewed for the next transition occurrence. With a value greater than 0 and lesser than 1, the monitor will successively sleep, during progressively shorter time intervals, as the playing time becomes closer to the next transition expected time, until the granularity value is

reached. With a 0 *sleep_factor*, the monitor will always sleep during the *granularity* value. It is important to note that, if the presentation rate suffer modifications, *granularity* and *sleep_factor* values will have to be proportionally corrected.

After each check, the adjustable monitor searches for transition expected times that have been reached, but not signaled to the formatter yet. If there is any, the monitor fires the corresponding event state transition, as the non-adjustable monitor does with fixed events.

The adapter also observes unpredictable presentation situations, like the modification of the content presentation position. When this occurs, the adapter asynchronously wakes up its monitors. The monitors then correct their current table entry, their current sleep time and the enabled table entries.

Finally, it is important to mention that the developed presentation tools are also capable of dealing with selection events over spatial regions and with anchors that combine presentation and selection events.

5 Related work

The main WWW browsers have a plug-in API, originally proposed by Netscape [11], which allows external applications to be incorporated to a Web client. Its main utility is presenting content formats not recognized by browsers. However, plug-ins are not hypermedia presentation tools but only presentation tools, since the API does not offer support for defining relationships among different plug-ins, and also among plug-ins and HTML pages. Actually, there are some mechanisms (e.g. LiveConnect and ActiveX) that allow the integration of HTML pages with plug-ins, and also with other client-side technologies, like scripts and applets. However, these mechanisms are browser dependent (Netscape or Internet Explorer) and based on hard code programming. Furthermore, a plug-in has its life cycle tied to the HTML page where it is embedded, which limits even more its use as a presentation tool.

Bouvin and Schade [3] propose extensions to the plug-in API with the main goal of allowing the creation of links between objects (mainly continuous media objects) independent of their presentation tools. Since the existent browser plug-in API could not be modified, they implemented a system, named Mimicry, using applet controllers to emulate these kinds of relationships. Their specific proxy, called DHMProxy, takes care of substituting plug-in markups, like *EMBED* or *OBJECT*, by references to their applet controllers. However, the system supports only the creation of links triggered by user interaction. It is not possible to define temporal synchronization relationships such as, start playing a video when a specific frame of another video is presented.

GRiNS [9] is a SMIL player developed by Oratrix that supports hyperdocument presentation with spatio-temporal synchronization. However, it does not allow defining synchronization relationships among internal events of object content.

Different from the aforementioned work, this paper proposed an adaptable and generic model of interface for integrating hypermedia presentation tools and

formatters. Our proposal offers support for controlling hypermedia document presentations with fine inter-media temporal and spatial synchronization. Observing important functionality needed on several hypermedia presentation systems, this paper also proposed a framework to contemplate this class of applications. New presentation tools can be developed by the framework specialization, while other existing ones can be incorporated using adapter modules. The design of JMF player adapters was only an example of how the framework might be applied.

6 Conclusion and future work

This paper aimed to simplify hyperdocument presentation system implementation, proposing an adaptable model of interface for the interaction of hypermedia presentation tools and formatters, independent of the hypermedia conceptual model they rely on.

This work also proposed a framework that helps the design of new hypermedia presentation tools and the adaptation of existing ones. As an example, a set of presentation tools for continuous media was implemented integrating JMF players with our proposal. This integration required a mapping study, which made possible to perceive that the JMF API could be extended to permit a more fine and efficient inter-media synchronization mechanism. The implemented presentation tools were integrated with the HyperProp system formatter [16]. At present, we are working on the integration of other existent players to our system, and also to contemplate other hypermedia formatters with our model. We are also working on refinements to our framework in order to comprise formatter internal design.

References

1. Anderson K.M. "Integrating Open Hypermedia Systems with the World Wide Web". *VIII ACM International Hypertext Conference*. Southampton, England, April 1997, pp.157-166.
2. Bachelet B., Mahey P., Rodrigues R.F., Soares L.F.G. "Elastic Time Computation for Hypermedia Documents". *VI Brazilian Symposium on Multimedia and Hypermedia Systems - SBMídia'2000*, Natal, June 2000, pp. 47-62.
3. Bouvin N.O., Schade R. "Integrating Temporal Media and Open Hypermedia on the World Wide Web". *VIII International World Wide Web Conference*, 1999, pp. 375-387.
4. Buchanan M.C., Zellweger P.T. "Automatic Temporal Layout Mechanisms". *ACM Multimedia'93*, Anaheim, California, August 1993, pp. 341-350.
5. Carr L., Roure D., Hall W., Hill G. "The Distributed Link Service: A Tool for Publishers, Authors and Readers". *IV International World Wide Web Conference*, Boston, USA. 1995.

6. DeRose S.J., Maler E., Daniel R. "XML Pointer Language (XPointer) Version 1.0". *W3C Working Draft*, January 2001. In <http://www.w3.org/TR/2001/WD-xptr-20010108/>.
7. DeRose S.J., Maler E., Orchard D. "XML Linking Language (XLink) Version 1.0". *W3C Recommendation*, June 2001. In <http://www.w3.org/TR/2001/REC-xlink-20010627/>.
8. Gamma E., Helm R., Johnson R., Vlissides J. "Design Patterns: Elements of Reusable Object-Oriented Software". *Addison Wesley*, 1995.
9. "GRiNS Player beta for SMIL 2.0". *Oratrix Development BV*, September 2000. In http://www2.oratrix.nl/W3C/index_html.
10. Jourdan M., Layaïda N., Roisin C., Sabry-Ismail L., Tardif, L. "Madeus, an Authoring Environment for Interactive Multimedia Documents", *ACM Multimedia Conference 98*, England, September 1998, pp. 267-272.
11. Netscape Communications Corporation. "Plug-in Guide". 1998. In <http://developer.netscape.com/docs/manuals/communicator/plugin/index.htm>.
12. Rodrigues L.M., Antonacci M.J., Rodrigues R.F., Muchaluat D.C., Soares L.F.G. "Improving SMIL with NCM Facilities", *Journal of Multimedia Tools and Applications*, Kluwer Academic Publishers, 2001. (*to be published*)
13. Rumbaugh J., Jacobson I., Booch G. "The Unified Modeling Language: Reference Manual", *Addison-Wesley*, 1999.
14. Sabry-Ismail L., Layaïda N., Roisin C. "Dealing with Uncertain Durations in Synchronized Multimedia Presentations". *Multimedia Tools and Applications Journal*, Kluwer Academic Publishers, 1999.
15. Soares L.F.G., Casanova M.A., Rodriguez N.L.R. "Nested Composite Nodes and Version Control in an Open Hypermedia System". *International Journal on Information Systems (Special Issue on Multimedia Information Systems)*, Vol. 20, No. 6. Elsevier Science Ltd, 1995, pp. 501-519.
16. Soares L.F.G., Rodrigues R.F., Muchaluat D.C. "Authoring and Formatting Hypermedia Documents in the HyperProp System". *ACM Multimedia Systems Journal*, Springer-Verlag, Vol. 8, No. 2, March 2000, pp. 118-134.
17. Sun Microsystems. "Java Media Framework, v2.0 API Specification". 1999. In <http://java.sun.com/products/java-media/jmf/2.1/specdownload.html>.
18. "Synchronized Multimedia Integration Language (SMIL 2.0) Specification". *W3C Recommendation*, August 2001. In <http://www.w3.org/TR/smil20>.