From Multimedia Stream Models to GUI generation

Jesús Bescós, José M. Martínez, and Guillermo Cisneros

Grupo de Tratamiento de Imágenes,
Dpto. Señales, Sistemas, y Radiocomunicaciones,
E.T.S. Ing. Telecomunicación, Universidad Politécnica de Madrid, E-28040 Madrid, Spain
e-mail: {jbc,jms,gcp}@gti.upm.es

ABSTRACT

This paper is centered on the description of a model that generalizes multimedia data flows handling including complete behavior and interaction mechanisms, hence allowing full integration of GUIs generation --GUI components are upgraded to interactive media items-- into the same unified model. It aims to reinforce portability, reusability, and quick development of multimedia applications.

A picture of previous and of current state-of-art in multimedia application development clearly shows the need for standard abstractions in this field. Current work in this direction leads to a discussion on generic application structure (objects, semantics,...) and on different approaches to reach platform independence and efficient object sharing (formal representation languages, interpreted programming languages, distributed environments,...). At this point, we present a basic model based on several stream-based models and implementations on multimedia data flows, and built on the basis of the source-stream-sink paradigm. It follows with a detailed explanation of the unified (common to all media) abstract basic stream from which all monomedia flows (including GUI elements) are derived: stream setting-up (source-sink adaptation, and negotiation), flow control procedures, stream sensibility, behavior pattern,... The model presentation ends up with the introduction of the multimedia stream that performs synchronization and inter-stream communication tasks, and channels all sensibility, from/towards its managed streams, and allows for the design of an application generator. Then it deeps into the definition of the abstract class hierarchy that guides the model implementation. Finally, several implementation issues are addressed and some practical achievements are described.

Keywords: Multimedia streams, GUIs, object oriented design, user interaction, application interpreter.

1. INTRODUCTION

Application development, and particularly GUI design, led, until not so many years, to a nearly bloody fight between programmers and I/O devices. The evolution and quick spreading of multitasking operating systems speeded up the lucky arrival of the nowadays well known *window environments* whose SDKs still offer never-ending sets of functions to manage spatial and functional relationships among windows hierarchies, message queues, and a great variety of graphical (fonts, color,...) and not so graphical (sound, pointing devices, keyboard,...) resources. This implied an actual revolution in programming philosophy and GUIs appearance that would pave the way for the introduction of object oriented techniques[1].

SPIE Vol. 2915 • 0-8194-2317-3/97/$10.00

Collections of general-purpose objects[i] that packed these low level management functions, along with more or less arbitrary guidelines to programming, appeared since then to help implementation tasks. This allowed for reusability, reliability, and development of portable applications.

In parallel with this, the increasing use of diverse media inside applications, somehow motivated by the availability of low cost equipment and network resources and the developer's need for abstractions[2] to manage them, launched the research on communication issues[3] (intra-medium synchronization and QoS) and the development of multimedia extensions[ii]. These allowed to control, in a quite non-standard way, more specific and usually proprietary devices, and provided the means to deal with other tasks like inter-media synchronization not usually solved by these devices. As it happened with window environments, work was focused on finding a solution to overcome the problems derived from the use of hardware-specific multimedia libraries and from communication lacks, that is to reach a consensus on a unique object oriented model[2], or any other high level programming framework[4]. Current implementations in this direction mainly include toolkits usually focused on one or several aspects (communication[5], inter-media synchronization[6,7,8], media composition[4] , distributed environments[5], etc.) and often inherently linked to an specific authoring tool[8].

Although object collections and toolkits are usually written in high level languages (HLL) and are oriented to ease application developers' tasks, authoring tools (often based in those toolkits) have originally been associated to multimedia presentations, either live or orchestrated, and intended for educators and other non-programmers, not skilled in conventional programming languages[8]. On one side, they use visual programming environments to help the definition of spatial and temporal relationships among monomedia objects, sometimes based on underlying formal specifications that allow for coherence checking. On another side, they usually allow the inclusion of GUI elements into the application to cope with user interaction, when supported.

Diverse specification schemes, mainly used to define inter-media temporal relationships, have emerged. They range from the use of time-stamped[6,9,19] data streams for continuos synchronization to the various formalized mechanisms for flow control: time-based[4,12,9] models, formalized in OCPN[10] and derived schemes, further evolved to allow for user interaction[11], event-driven[8,12] models, and other more or less innovative ones[7]. Recent works trend to solve each model lacks to reach common maximum functionality. Regarding user interaction and GUI generation, these tasks are usually carried out by either independent (case of toolkits) or integrated (in the case of some authoring systems[8]) GUI builders, and somehow linked to the multimedia presentation flow controller. However, they still remain as independent entities.

Back to the creation of multimedia applications, if we assume that a generic application can be seen as a group of interrelated objects, the solution to the development problems lies in the design of schemes and tools which improve and formalize definition, interpretation, presentation, and management of those objects. This means, first of all, that application programming would be reduced to a process whose first phase would consist on the selection, or design of the desired objets (shareable[13] if their coding follows standard rules); a second phase would consist on the association, to each object, of the data identifiers (i.e. URIs[14] like) to operate with; the final phase would define each object's behavior when interacting with either the user, other application objects, or the system that manages them. Second, any application execution would be achieved by unique *program* able to interpret and resolve the *instructions* coded in each object.

The aim of this paper is to describe a set of tools and abstractions that allow the programmer to systematize multimedia flows and user interaction handling. For this purpose, first, we discuss on generic application structure and its evolution. Second, we present an overview of the model, based on several stream-based models and implementations on multimedia data flows. The model presentation, built on the basis of the source-stream-sink paradigm, is followed by the description of the interaction mechanisms that allow for GUI integration. The model ends up with the presentation of the multimedia stream that performs synchronization and inter-stream communication tasks, and channels all sensibility, interaction, and error flow from/towards its managed streams. It follows a detailed explanation of the unified abstract basic stream API. Finally, several implementation issues are addressed and some practical achievements are described.

---

[i] OSF's Motif widgets, Borland's OWL, or Microsoft's MFC and their VBX, OCX, and recent Active-X objects.

[ii] Microsoft's Multimedia Extensions, Apple's QuickTime, IBM's MMPM/2, DEC's Xmedia,...

2. APPLICATION STRUCTURE

Following the ideas outlined in the introduction, we can describe the evolution in application development by distinguishing several aspects inside the execution process of an hypothetical multimedia application: the application objects, the application semantics, and the application data. This section briefly discusses on these subjects.

2.1. Applications as interrelated groups of objects

Object-oriented design of applications is nowadays a quite established policy. However, as commented in the introduction, we can find multiple object collections in the software market, each with its own programming *style*, and usually accessible via HLL APIs. Although some of these collections have even been ported to several platforms, they are far away from being fully portable, shareable, and reusable.

Great effort has been carried out to propose a standard mechanism that fulfilled the aforementioned properties. There are two main trends currently active in this field. The first one tries to code more or less complex objects by means of the definition of abstractions: superstructures (basic objects) and schemes that conform a document model. Once the desired objects are outlined, and depending on the referenced parts of them, the abstract definition is formalized via the appropriate representation language (ASN1, SGML, predefined events and actions, scripting languages,...). This direction is the one followed, for instance, by ODA and HyperODA[12] , MHEG[13], and HyTime[15]. Some authoring systems (whose abstraction ideas are common to these models) translate their outputs to these standard schemes, adding proprietary extensions to overcome someones lacks in synchronization definition[12]. The second trend starts from scratch, that is, uses a general purpose interpreted object oriented programming language, over which you can build whatever you desire, particularly complex objects. This low-level standardization is the one followed by systems like SUN's Java.

In short, both approaches reduce applications to feeding instructions into a virtual machine or parser whose functionality is supposed to be hardware independent. At this point its important to remark the evolution of application development models[16] that offer *standard* communication, management, information retrieval and presentation services, etc., hence providing a software platform over which you can design, for instance, a multimedia document presentation engine.

2.2. Application semantics

This is what actually defines what an application does. Basically we could define it as the set of relationships existing among all its component objects, a particular network of stimulus-response associations that makes them execute an specific action pattern: the final application objective.

2.3. Application data

Here we just want to stress the difference between basic objects and data. Following MHEG approach, we can understand basic objects as data structures that hide data formats heterogeneity by showing a unique interface, independently of whether they are *filled* or not with actual data.

3. THE STREAM MODEL

This section shows an overview of a stream model that includes user interaction mechanisms so that GUI generation can be integrated into the same unified model. Following this approach, the overall multimedia application can be defined using the same abstractions for both, traditional multimedia objects (video, audio, text, images,...) and typical GUI functionality.

3.1. Monomedia and Multimedia streams

When talking about streams related to multimedia, it is not always very clear what exactly we are talking about. Throughout this paper we will refer to a *monomedia stream* as a functional unit (defined via a C++ class, and implemented in an independent thread) that contains, checks for compatibility, and connects a single source to a single sink, allows control of this connection via a unified public interface (a fixed set of functions independent of the media type), is able to receive user interaction messages (stream sensibility), and maintains basic timing parameters (local stream time) and error handling to allow for external synchronization and error management. The original concept follows the ideas initially outlined in Gibb's *active objects*[3], Steinmetz's *functional units*[4], the Tactus timing mechanisms[6], and others[11], further evolved to support full application development over them.

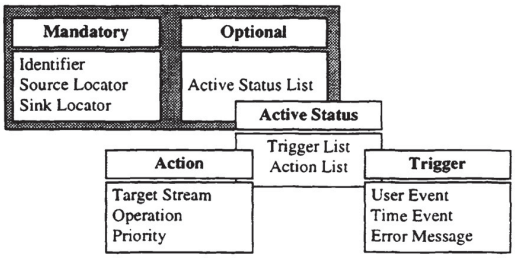

Fig.1: Monomedia stream definition

The *sink* and the *source* are independent functional units (both also defined as C++ classes) that make use of the available services[16] (communication, presentation, information, management,...) to *throw* or *extract* data to or from their location respecively. They provide a data abstraction for the monomedia stream to uniformly perform its flow control tasks.

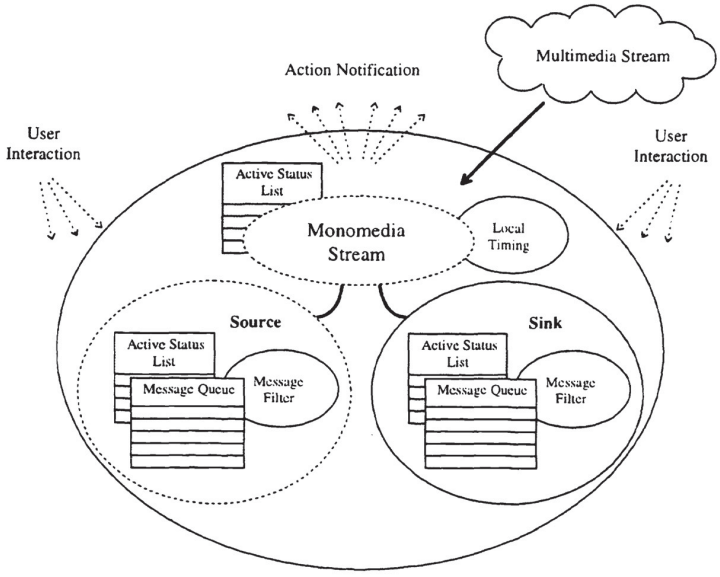

Fig. 2: Monomedia stream block diagram

A monomedia stream is defined (see Fig.1) first by a *stream identifier*, so that other streams can reference it; then, by two locators (URI based) describing the origin of data to *fill* the source, and the final data placement achieved by the sink, respectively; finally, it can optionally contain a list of *active status* composed each by a *trigger* list and an *actions* list. On one side, triggers are classified, attending to their provenance, into user actions (applied over the source or sink), timing events, and internal errors. Depending on the trigger type more conditions can be defined (for instance, a mouse input user event can have a sensible area associated) On the other side, allowed actions are composed of the target stream, the desired operation (all the available in the common public interface of monomedia streams plus any other available in each specific stream) and an optional priority flag.

To achieve the desired functionality (see Fig. 2) the monomedia stream classifies and distributes the active status according to their triggers types: the ones containing user events are *sent* to the sink or source active status lists, and the ones not associated to user interaction are kept in an independent list. From this information, both the sink and source generate their own message filters to increase efficiency in later message handling. Afterwards, the stream enters into a sensitive status, waiting for user events, error mesages, or temporal deadlines, and ready to perform any operation queried by the multimedia stream.

A multimedia stream is here referred as a superstructure that manages all the monomedia streams that owns. It is in charge of the execution of all the actions queried by its managed monomedia streams. For this purpose (see Fig.3), it maintains global timing parameters (application time), a list of the launched monomedia streams (threads), and two message queues: the *asynchronous queue*, a priority ordered one which keeps actions coming from user events and error messages, and the *synchronous queue*, a time ordered one for time events generated actions. Once the multimedia stream receives its definition (a list of monomedia streams that may conform a full application) it fills its synchronous queue (the asynchronous one will obviously be empty), and begins to get the actions from its queues and execute them.

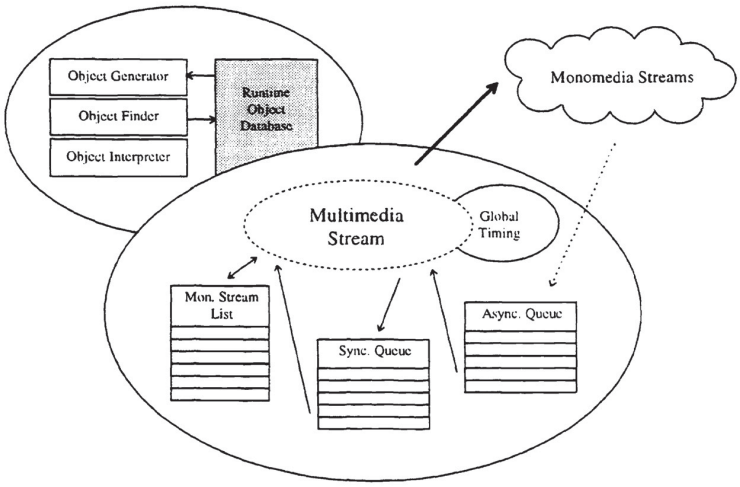

Fig. 3: Multimedia stream block diagram

3.2. User Interaction and GUI generation

Starting from the ideas described in the previous subsection, its easy to conclude that, based on the definition of the stream sensibility, and selecting an appropriate sink that *throws* animated images to the screen, and a compatible (as will be explained in section 4, compatibility depends on sink, stream, and source capabilities) source that *extracts* an image sequence from its data locator (either local or remote), it is possible to imitate the behavior of most existing GUI elements (buttons, scroll-bars, menus,...). Moreover, this approach, along with the provision of collections of proper more intelligent sinks,

streams, and sources (currently under development at our premises), allows the application programmer for the adoption of more innovative[17] and object-oriented GUIs[18].

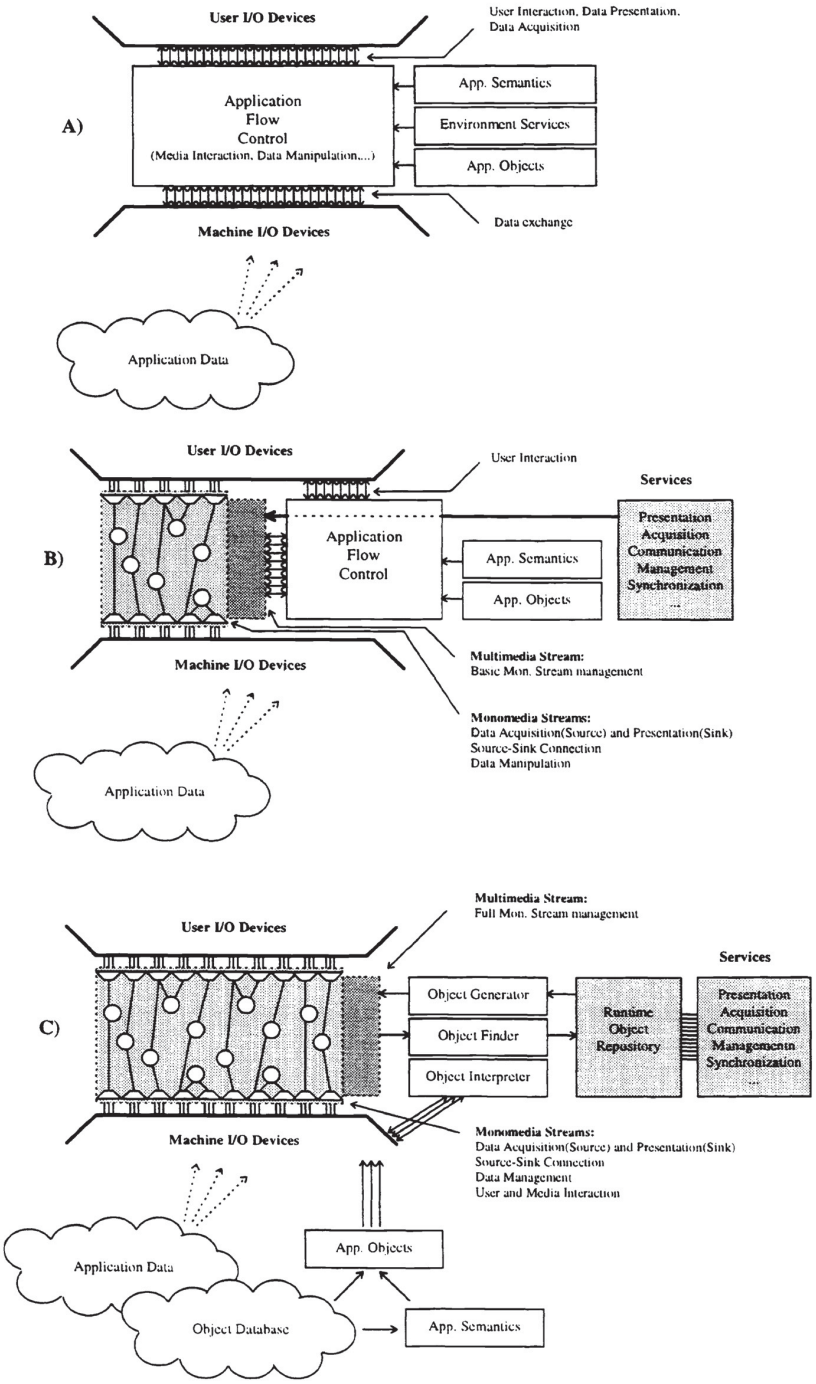

Fig. 4: Application development models

3.3. Streams and Application development model

Traditional and even some current 'object-oriented' multimedia application development, either supported or not by complex programming environments, usually leads to applications within whose code (HLLs in the best case) it is nearly impossible to separate or even distinguish any application semantics (often blurred by never-ending crossed or recurrent function calls), I/O devices handling (performed via proprietary APIs of a specific operating system or extended toolkit), GUI management (which is surprisingly found everywhere), and other frequently performed tasks (network usage, data management, etc.). This approach (see Fig. 4a), although currently the most straightforward and efficient (from a performance point of view) in most cases, makes quite difficult to maintain the application, and makes quite unlikely to achieve portability, reusability, sharing, etc.

As explained in the introduction, most efforts to overcome this problem have been directed towards the definition and implementation of programming frameworks, or object oriented models, and towards the normalization of basic services offered to hide platform dependencies. These achievements usually provide a solution to *standardize* management of multimedia data flows which can hence conform a quite independent and separated part (or the whole application in a simple non-interactive multimedia presentation) of the overall application. However, GUI management, semantics, object generation, etc., are still masked under the application flow control. Although in the case of using multimedia toolkits the application is usually still programmed by means of an HLLs, this level of abstraction allows to build quite efficient authoring systems which, via integrating typical GUI elements into their authoring environments, clearly ease the development of complex interactive presentations. This is the situation depicted in Fig. 4b, which presents an intermediate step in the design of the stream model presented in this paper: monomedia streams (depicted as source-stream-sink connections) are still insensitive, available stream types (file to screen, network to disk,...) have to be *a priori* known (at compile time) by the application, available services are used at several abstraction levels, and overall application definition via a formal representation language is not yet easy to afford (although some authoring systems manage to write formal standard definitions[8]).

Fig. 4c depicts a generic multimedia document interpreter which executes the application, defined by the semantics attached to these documents, by means of the stream model presented in this paper. To achieve it, first, streams have been expanded with user interaction capabilities so that GUI elements can be modeled with the same abstract notation as any other typical multimedia flow; second, the abstract class hierarchy (see section 4.1) over which all streams, sink, and sources are built, allows for external run-time object creation (see section 4.3) so that implementation new streams or upgrading of existing ones does not require any modification nor recompilation of the multimedia stream generator, an actual application generator.

4. IMPLEMENTATION ISSUES

Implementation of the base classes (C++ has been used) has been carried out over Microsoft's MFC object collection. These classes don not deal with an specific media type. They are media-independent. Hence, only when strictly necessary, direct calls to the Win32 API have been made (i.e. accurate timing provided by the Multimedia Extensions), so that a certain degree of portability can be achieved. In the case of derived sink, source, and stream classes, which are part of the object repository (see Fig. 4c) and may deal with specific media types and proprietary devices direct calls to dedicated system or vendor libraries have been used.

4.1. The class hierarchy

Fig 5. shows a diagram of the four basic C++ classes, along with a description of their main functionality. The CMMStream class (identified with the multimedia stream object) is the only one that can be *instantiated*. The other ones, CMSink, CMStream, and CMSource (respectively implementing a sink, a monomedia stream, and a source) are abstract classes, that is, its medium-dependent public interface is composed of pure virtual functions; hence, they act as base classes to derive media-specific dealing ones.

It is important to remark that the CMMStream class *knows* nothing about any medium-specific dealing object, nor nothing about sinks nor sources. As described previously (see Sec. 3.1) it maintains a list of monomedia streams objects, but only can access them via their base class (CMStream) public functions. Hence, the multimedia stream is also media-independent. This is a main requirement to implement an external object repository and therefore to achieve independence between the application interpreter and the application definition (objects, semantics, and data).

Another main point in the design process is the fact that whereas both the multimedia and monomedia stream classes are base classes (not derived from any other), the sink and source abstract classes are both derived from a basic window class, in order to implement a message map that allows for user interaction (screen input/output for specific sources/sinks, and user event reception).

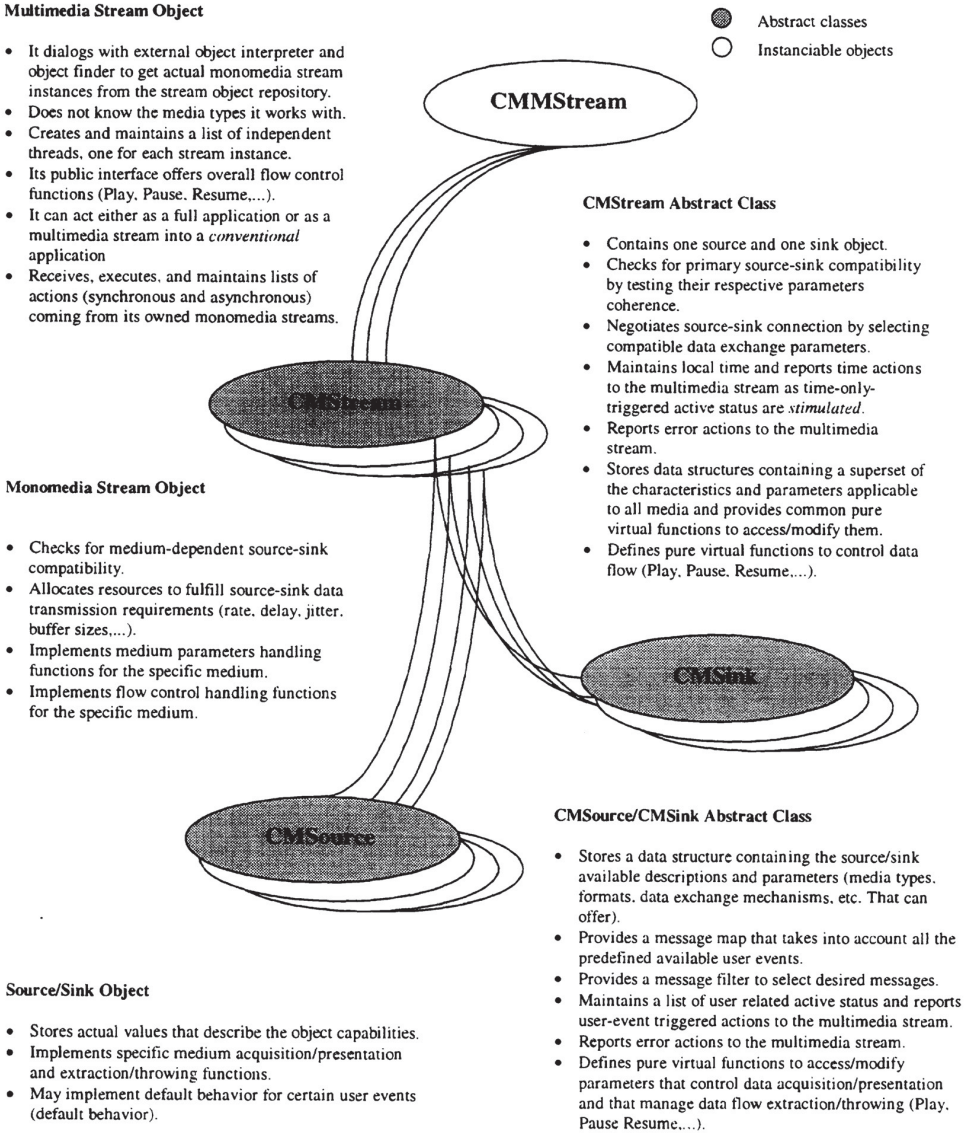

Fig. 5: Stream model class description

4.2. Inter-stream communication

The multimedia stream centralizes all communication among monomedia streams. Direct communication between two monomedia streams is hence not allowed. All communication is implemented into the base abstract classes, so that medium-dependent streams do not have to deal with this subjects (object inheritance).

Whenever the multimedia stream receives from the object finder a monomedia stream instance, as a result of the execution of a particular action, it launches it into a new thread. This is done independently of whether the specific stream performs any other multithreaded tasks. Hence, operations requested (by actions) on one or several monomedia streams are signaled via one of the available *synchronization objects* (events, mutex, semaphores,...) of the Win32 API.

Asynchronous messages (user, error, time) sent to the multimedia stream when an active status is triggered are performed directly via function calls to the classes that manage the synchronous and asynchronous multimedia stream queues.

4.3. Runtime object generation

Available medium-specific streams (fixed sink-stream-source triplets) are all stored into one or several external DLL (see Fig. 4c). This receives the source and sink locators (from the object interpreter) and starts looking (object finder) for a stream whose negotiation process is successful. Once found, returns a pointer to the allocated object (object generator) *casted* to an object of the base abstract class (CMStream). Hence, the media-independent multimedia stream will reference common-to-all-media (but specifically implemented for each) functionality by using functionality defined in this abstract class. Regarding functions specially needed or provided for a particular stream (geometric operations available for an image stream, for instance), a higher level protocol must be established between the stream designer (who describes via a formal language) and the specific stream implementation. For this purpose, the base abstract class includes a *dummy* function to allow sending of proprietary messages (and then launch specific actions) to an stream. This is something similar to the use, in multimedia document formal representation languages (see Sec. 2.1), of scripting languages when no predefined actions can perform the desired operation.

4.4. Stream and application examples

Regarding conventional (non GUI related) streams, several basic ones have been implemented to deal with persistent video, audio, still images, and text, and non persistent audio. However, their features are not relevant for this paper purpose.

Current available GUI streams are based on a still image stream (one image for icons, two for single or double state buttons, three or more for micons[18] and several state buttons,...) supplied with default behavior (for instance, automatic highlight when the mouse pointer is on), which we consider enough to achieve full user interaction.

Fig. 6 shows a video viewer, one of the applications developed using the presented stream model. It shows several GUI streams the are used to call the flow control functions of a video streams. The single multimedia stream (the application itself) is composed of:

- An *image stream* that displays the application background . This stream has no active status.
- A *micon stream* (derived from an *animation stream*), that displays our researching group logotype in an *elastic* way.
- Nine multi-state *button streams* (also derived from an *animation stream*). Six (the centered ones) display multi-image push buttons to control the video sequence playback. They have default behavior (their own Play function is called when the user clicks on them), and an active status, triggered by the mouse click, whose action targets the video stream and specifies the graphically described operation. The three remaining (the left ones marked with +,0,-) control parameters of the video sequence (audio and video *distance*).
- A video stream, which covers most of the application display area. It has much default behavior but no active status.

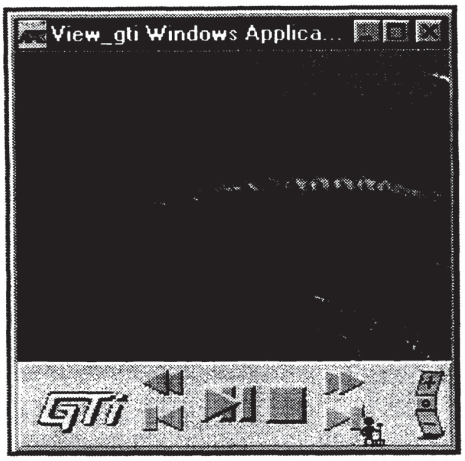

Fig. 6: An example application

5. CONCLUSIONS

This paper has presented a multimedia stream model for the development of full multimedia applications. The original concept follows the ideas initially outlined in Gibb's *active objects*[3], Steinmetz's *functional units*[4], the Tactus timing mechanisms[6], and others[11]. The main difference resides in a further evolution to support GUI elements integration into the same unified model, via the inclusion of interaction management into the stream classes, hence allowing for full application development over the same abstractions.

Once achieved this unified model, it is easier to undertake the implementation of an application generator, which interprets stream definitions expressed via a formal representation language. A practical implementation has been presented, and some resulting examples described..

We have not aimed to establish any formal definitions for either describing the application components (streams) or the synchronization structure. These would, however, be tasks to implement in a future authoring tool based on these principles.

6. ACKNOWLEDGEMENTS

To our master's thesis students Julián Cabrera, Sonia Vergara, and Rita Vázquez for their valuable collaboration in implementation and design tasks.

Work partially supported by the Comisión Interministerial de Ciencia y Tecnología of the Spanish Government.

7. REFERENCES

1. J. Rumbaugh et al., *Object-Oriented Modeling and Design*, Prentice-Hall, 1991.

2. R. Steinmetz, J.C. Fritzsche, "Abstractions for continuous-media programming", *Computer Communications*, Vol 15, no 6, pp 396-402, July/August 1992.

3. R. Steinmetz, T. Meyer, "Modelling Distributed Multimedia Applications", in *Int. Workshop on Adv. Comm. and Appl. for High Speed Networks*, Munich, March 16-19, 1992.

4. S. Gibbs, "Composite Multimedia and Active Objects", in *Proceedings of the ACM Conference on Object-Oriented Programming Systems*, Languages, and Applications, pp 97-112, Phoenix, New York, 1991

5. T. Käpner et al., "An Introduction to HeiMAT: The Heidelberg Multimedia Application Toolkit", in *Proceedings of the Third International Workshop on Network and Operating System Support for Digital Audio and Video*, San Diego, CA, November, 1992.

6. R.B. Dannenberg et al., "Tactus: toolkit-level support for synchronized interactive multimedia", *Multimedia Systems*, Vol 1, pp 77-86, September 1993.

7. R. Hamakawa, J. Rekimoto, "Object composition and playback models for handling multimedia data", *Multimedia Systems*, Vol 2, pp 26-35, June 1994.

8. S. Eun et al., "Eventor: an authoring system for interactive multimedia applications", *Multimedia Systems*, Vol 2, pp 129-140, September 1994.

9. G. A. Schloss, M.J. Wynblatt, "Providiong definition and temporal structure for multimedia data", *Multimedia Systems*, Vol. 3, pp 264-277, November 1995.

10. T.D.C. Little, A. Ghafoor, "Synchronization and Storage Models for Multimedia Objects", *IEEE Journal on Selected Areas in Communications*, Vol. 8, No 3, pp 413-427, April 1990.

11. B. Prabhakaran, S.V. Raghavan, "Synchronization models for multimedia presentation with user participation", *Multimedia Systems*, Vol 2, pp 53-62, August 1994.

12. H. Khalfallah, A. Karmouch, "An architecture and a data model for integrated multimedia documents and presentational applications", *Multimedia Systems*, Vol. 3, pp 238-250, November 1995.

13. T. Meyer-Boudnik, W. Effelsberg, "MHEG Explained", *IEEE Multimedia*, Vol. 2, pp 26-38, 1995

14. "World Wide Web Specifications - URI", currently located at *http://www.ccs.org/validate/wwwspec5.html*.

15. J. Budford, et al., "HyOctane: A HyTime Engine for a MMIS", *Multimedia Systems*, Vol. 1, pp 173-185, February1994.

16. J.M Martínez, et al., "Functional Services for Application Development ", next paper in this volume.

17. J. Grudin, "Interface, an evolving concept", Communications of the ACM, Vol. 36, No 4, pp 111-119, April 1993.

18. J. Nielsen, "Noncommand User Interfaces", *Communications of the ACM*, Vol. 36, No 4, pp 83-99, April 1993.

19. D. Hehmann, et al., "Implementing HeiTS - Architecture and Implementation Strategy of the Heidelberg High-Speed Transport System", in *Proceedings of the Second International Workshop on Network and Operating System Support for Digital Audio and Video*, Springer, Heidelberg 1992.