A Constructive Type Schema for Distributed Multimedia Applications

Paulo Pinto, Luis Bernardo and Paulo Pereira

Inesc / IST, R. Alves Redol, 9 P-1000 Lisboa, Portugal
{pfp, lflb, prbp}@inesc.pt

ABSTRACT

The constant evolution of multimedia technology puts strong requirements on the design of distributed multimedia systems if upgradability and extensibility are desired. A proposal for an object and a composition models based on a constructive type schema is described in this paper. Objects and compositions are aggregations of various small aspects to which the system reacts. As the system gets richer and more complete, new aspects can be easily added. Type information is left out of the objects and is used in the distributed system for various purposes. Simpler views of the models create environments where the construction of specific kinds of applications is easily performed. A visual editor is described for one of such views - distributed multimedia applications based on coarse temporal, spatial and logic synchronization.

Keywords
type categorization, multimedia composition, distribution, visual composition, active objects.

1. INTRODUCTION

Programming (or authoring) multimedia applications is not an obvious task and needs a powerful framework providing an environment where they can be constructed easily and quickly. Some issues should be addressed and solved by the framework:

The current state of the art in terms of equipments and signal encoding standards is characterized by a constant evolution. Features are permanently being added to the equipments creating new ways to handle data or interact with other types of data. In terms of signal encoding, better tradeoffs between quality of data and usage of bandwidth are also being obtained. This ongoing evolution prevents the use of a static type categorization for multimedia objects if the framework is to be used for a long time.

The framework should create an environment where the direct handling of temporal based signals should be avoided. This handling is computational consuming if performed by the application and most of the times useless for authoring purposes. The framework should also reduce the dependency on specific hardware by appropriate encapsulation. The set of authoring concepts should be kept small yet powerful enough to grab all the possibilities an author needs to use to compose applications. The current reality is the existence of complex system demanding a considerable technical knowledge of the objects by the author.

If distribution is also considered, some assumptions are not valid anymore. Examples are object availability or the facilities to interact with objects or to access them. On the other hand, it introduces new problems to be solved, such as bandwidth of channels or delays in reacting to commands. The framework should also solve these issues in an integrated way.

Still another aspect is the wide scope covered by the term *multimedia applications*, ranging from simple hypermedia documents with a hierarchical structure based on a text document, to more intricate relations including graphic animated objects or virtual realities. Optimal solutions for one case can be inefficient for others.

A solution to this problem, using type categorization extensively, is proposed in this paper. The paper only addresses part of the overall problem (the script language, and some implementation aspects are not fully described. See section 9). The solution is not unique for the entire problem. A common system architecture with different views is defined. Each view takes into account the specific characteristics of a certain kind of application. All views are based on a single object model which includes the allowed interactions and the definition of the active components. The paper is divided in three main parts: the following section describes the system architecture and the motivation to define it. Part two explains the object model. Both part one and part two are evolutions from the corresponding parts in [23]. The third part describes one of the views - multimedia applications based on coarse interaction between objects - and presents a visual-based tool to interactively create the application. Section 8 discusses some related work and the final section draws general comments on this work and on the future directions to be followed.

2. SYSTEM ARCHITECTURE

2.1. Motivation

Current frameworks for the construction of multimedia applications have native data type objects in a tightly integrated form concerning the functions available on the objects. New data type objects can still be integrated via modules which know how to handle the data but offer fewer features than the native

objects [19, 20]. A first requirement for the system architecture proposed here is the ability to recognize any kind of relevant function, for composition purposes, regardless of the nature of the object.

A second requirement is the ability to use any kind of interaction between objects both at data level (allowing for different kinds of configurations between object components called sources, sinks or converters of data [23]) and at control level (allowing for different kinds of participation other than triggering the start of other objects or the user watching the "play").

Most of the current systems were built with a specific kind of application in mind. The consequence of such an approach starts on the definition of the authoring concepts and goes until the relations between the components (usually using inheritance), emphasizing relations of certain features of the objects more than others. A different kind of application would have other ways to relate the objects between each other causing a reorganization of the inheritance chain and a major change on the system. Most of the times the differences are just on the way authors write specifications and not on the object model itself. Different tools to build applications could reuse the basic objects, as long as a constructive type schema for using their features is available. The system architecture described here has such a typed vision of the objects.

The system must provide a very simple authoring conceptual model with only the necessary features for the selected kind of application. The features must be coherent, at type level, with a richer object model common to all views. This creates an easy authoring environment which can be used by non-expert people and still reuse a common system underneath. It is believed that multimedia applications will use autonomous objects in a distributed world which are able to handle data and have methods at interfaces to allow some degree of control. Ultimately, these objects can be defined using any kind of technique (definition of methods, software construction, etc.), but the way they are categorized with the type schema proposed here, makes the system usable regardless of future modifications or addition of features.

The last two main requirements are widely discussed on similar works and concern the encapsulation of hardware dependencies and the use of active objects to model the multimedia objects [8].

2.2. Architecture

The overall system architecture is shown in figure 1. Active objects implement the basic components acting as sources, sinks and converters. They cooperate in three different ways to accomplish the multimedia application:

(a) by transferring data streams from sources to sinks (possibly using converters on the way);

(b) by exchanging control data between them; and

(c) by invoking methods on each other.

The data stream part (a) is somehow left out of the model. Data streams are just typed entities which can be generated in ports of the same (or compatible) type and are consumed by similar ports. A type space for them is defined with relations of compatibility. In informal terms, compatibility here means the ability of the port to handle the data encoding and the mechanisms to transfer it (control methods associated with the transfer).

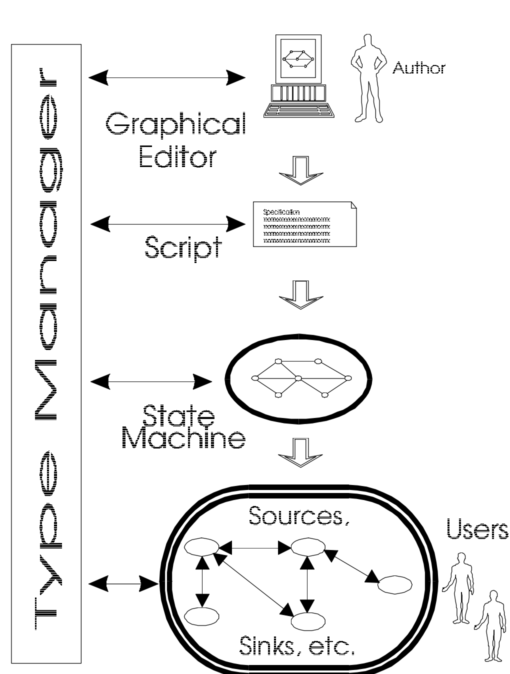

Figure 1 - System Architecture

The other two ways, (b) and (c), form an *abstract machine* with certain operations (the methods on the components) and data flows (control data). All this environment is strongly typed, checked at run-time by the run-time system and at compile-time (or specification-time) by the tools available. All control data uses messages, called *events*, between the components. Every event belongs to a type and has associated data. Methods are aggregated using concepts which facilitate type checking and extensibility (see section 3). A system-level *type manager* was used to gather all the type information in the system. This approach contrasts with others using type models of object-oriented programming languages. Some discussion of this issue is done later.

The *abstract machine* can be driven directly by an application if the interactions are simple, or it can be driven by a state machine generated from a script language, in more complex cases. It is expected that more than one script language will be used for multimedia applications. Some of the currently proposed ones [20, 12] handle certain aspects more than others and show limitations for certain cases [23]. However, if the abstract machine is general enough it can support different classes of script languages. Each class will correspond to a view of the overall system defined earlier. Although building an editor (or compiler) capable of using all the facilities of the abstract machine is possible, it would demand a high technical knowledge of the system from the author. Instead, by restricting the concepts to the type of application of interest, a simpler environment can be created.

The simpler environment of this paper, uses a graphical editor to interactively specify and compose the application. The editor generates a script language (see section 9) but could generate the state machine directly. Other kind of applications, such as hierarchical documents spreading out from a root text object (hypermedia style), would have different concepts on their editors.

As figure 1 shows, the type manager is used by all entities to make on-line checks on every operation the author wants to make.

3. COMPONENT OBJECT MODEL

There are two levels of objects in this paper. Objects at component level, and objects used at authoring level. Objects at authoring level are simplified views of the others but reflect most of their structure. This section describes the component level objects, and the next section covers the others.

Component objects model sources, sinks and converters.[1] The model described here is based on [24] and each object has a set of typed interfaces. This approach is in line with the Reference Model of Open Distributed Processing [13], and differs from those based on object-oriented programming languages where each object has a unique interface. With this approach it is still possible to think of an interface as composed of different parts assembled by the inheritance relation between objects. However, such use binds the particular meaning of inheritance of code to the unique sub-typing relation object-oriented programming languages have, preventing other useful usages.

A multi-faceted approach was used instead. Each component object is composed of different *status*. A status is a set of internal state with associated logic. There are status for speed, geometry, brightness, volume, etc. If an object does not support the status for volume, for instance, it cannot interact with the exterior to change the value controlling the volume of the sound produced. In terms of the computational model, status does not introduce any new concepts but just a way to relate interfaces. A status forms a close set of features associated with some particularity of the system. However, it should be possible that the logic associated with a certain status could influence other status. This is an internal issue of the object and the programmer is free to implement whatever influences he wishes, as long as the necessary communication mechanisms are supported. An example of cross-influence is a status related with bandwidth of a channel (quality of service) generating a new geometry for the object to reduce the amount of data transferred, and sending the new geometry information to other objects that are interested. Another internal issue is how inheritance can be used to form status. It is a feature related to the construction of objects not relevant at composition level.

---

[1]A convertor is used to link incompatible sources and sinks without modifying them, or to provide access to the data stream using another data type (for instance, grabbing a frame from a video stream). Whether the convertor is a first class computational object in the system or a facility bound to the communication infrastructure, is not so relevant. In this paper the important feature is the existence of an operational interface to control it.

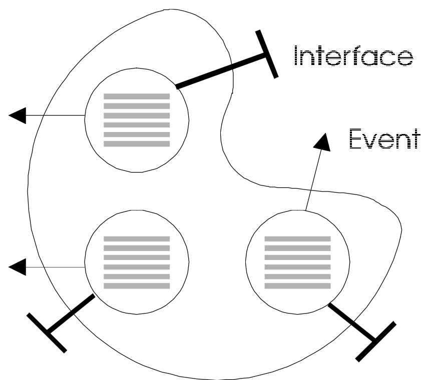

Figure 2 - Internal structure of a component object

There is a set of operations associated with each status - its interface - and a set of event types (Figure 2). For instance, a timestone event type would be associated with the annotation status. The logic of each status can be triggered by internal changes in the object or by external invocations on operations associated with the status. The type manager holds information relating status-events-interfaces and which status an object supports.

The complete interface of a component object is thus the set of all interfaces of the statuses it supports.

It was already said that objects interact. Interaction consists of invoking methods on others and sending events to others. The traffic of events is based on interests: objects register interest on the occurrence of some kind of events in other objects. They can be interested on changes of speed, or even on when an object finishes displaying the stream of data.

This concept of status is very useful for four main reasons:
(a) avoids the need to carefully design a common interface to all objects in order to use them efficiently. There is no need to define empty procedures (such as SetVolume for slides), or overload operations with a less intuitive meaning;
(b) avoids the requirement of having to design the interface of a medium type in such a way that it will cope with any future evolution of the technology of the components (new features available) minimizing its disturbance on the overall system. With this constructive type schema an old type, with an added status, is all that is needed;
(c) allows that tools made at a certain version of the system can still work with objects having features not known at that time.
(d) multimedia systems can be constructively built with objects being added as they are needed.

Just as status and events are typed entities, so are sources, sinks and converters. As it will be seen, a multimedia object is ultimately a combination of cooperating sources, sinks and converters. A type in this space represents a certain set of status and the form of dealing with the events (i.e., the logic). For instance, a certain video source which would never generate the event representing the end of the movie (leaving that task for the sink), would belong to a different type from a source that generates this event after sending the last bits of the stream. This type information will be helpful in deciding if two components can be connected together, and is stronger than just the statement of compatibility between data stream types.

The system being built has special component objects which are templates for the instances of the type of source, sink or converter wanted. It is a similar technique as the concept of factory in ODP [13] and ANSA [1]. The instances created will have some context data, called *property data*, regarding the specific object (name of files, geometries, references to interfaces of data stream ports, etc.). The template object only supports one status, *ComponentLife*, having state variables and the operations *prepare* and *dismiss*. *prepare* creates an instance of that type with specific property data values as initial information. The invoker of the operation is the only user of the new instance and can share its usage with other components if so wishes. *dismiss* is one way of aborting that instance.

Play ( )
Stop ( )
Inform ( )
Destroy ( )
ReceiveEvent ( )
RegisterStatusInterest ( )
UnregisterStatusInterest ( )

Ev_Ready
Ev_Start
Ev_Stop
Ev_End

Figure 3 - Composition of the *Context* status

All instances of the components have the status *Context* which gives the basic functions for the system to run. So, all component objects have, at least, these functions. The reference to its interface provides unique identification of the component in the system. Figure 3 shows the operations and events associated with the status Context. Apart from the obvious operations to play, stop and destroy the instance there are operations to receive events (a kind of dispatcher), to provide the references of interfaces of other status the object supports, and operations to allow other objects to register (and unregister) interest on event types. As this context exists on all objects the events associated with it are implemented by every component. They are *Ev_Ready* (as a reply to the operation *prepare* because the operation is not blocking to the caller), *Ev_Start* (to allow others to know that the object started), *Ev_Stop*, and *Ev_End* (to announce the termination of a play).

Other statuses are defined and are used to handle specific aspects of the object. For instance, the status *Volume* has the operations *setVolume*, *changeVolume*, and *getVolume*, and can send an event of the type *Ev_Volume* with the relevant data. A more complex status is the *Annotation*. Annotations (or timestones) in objects provide them with an internal structure for composition purposes[2]. Annotations can be subdivided in different subtypes (still visible to the type manager): *subtype_A*, *subtype_B*, etc. with data values such as "*yellow car*

[2]The way this is really implemented is not so relevant. The data stream can be considered unstructured and out-of-band (or in-band) signalling can be used to transport the annotations, being identified by some component responsible to send the events.

passing", "*plane taking off*", "*chapter 3*", "*frame 34012*", etc. When the object is playing and an annotation is passed, a corresponding event can be sent to any object that showed interest on it. The event is *Ev_Annotation* with the corresponding values for subtype and values. The status Annotation has placement operations to access the position of the object presentation from the exterior - *gotoAnnotation*, *getLocator* - and specific operations to register (unregister) interest on the events.

It should be noted that this object model fits well in standardization efforts such as MHEG [11], or in other models constructed using object-oriented programming language type models [18]. What is highlighted here is the way the system looks at the operations available on the objects. For instance, MHEG has a corresponding set of operations analogous to the ComponentLife and Context status called *preparation* and *presentation actions*. Furthermore, it has a flat type space of actions (methods) which can be seen through the glasses of this type model with all the advantages described earlier.

We claim that this model, associated with a powerful script language (not described here), is general enough to implement most of the distributed multimedia applications.

4. MULTIMEDIA OBJECT MODEL

The simpler set of concepts defined on top of the general component model is used to construct multimedia applications based on coarse spatial, temporal and logic synchronization between multimedia data types. The main concept is a *MM_Object* (multimedia object) which is an active entity that starts, has a lifetime, and ends, handling, or not, multimedia data [23]. Figure 4a shows the simplest object with the events *Ev_Start* and *Ev_End*, thus unable to interact at a finer grain than the whole object. Multimedia objects are also created from a unique template (common to all objects) using a status called *Life*. Life has similar operations to those of ComponentLife, but includes the identification of the components to be used and their configuration in terms of media streams.

Multimedia objects are also typed entities. A type in this space represents a set of status implemented by the object, and a list of components which can be put together to generate it. For instance, a video of a certain type can be originated in a disk and be displayed on the screen of a workstation. The sink could be changed to a TV set sink with the same kind of features (status). However, if the video comes from a camera, then it is categorized as a different multimedia type because the camera does not support changes in speed (slow down reproducing). In the first case (changing from the screen of the workstation to the TV set) both sinks must have an identical way to handle the aggregated logic of the respective status.

Multimedia objects also have the same structure of status and events. The difference is that the handling can be done by the multimedia object itself, or can be delegated to a component object. This is totally transparent to the user of the object because he asks for references to interfaces and the ones he is given back are really the ones on the components. An example of direct handling by the multimedia object is the *Context* status whereas the handling of Speed is generally performed directly by a component. In complex cases, with configurations different from one to one, the status in the multimedia object could have to perform consistency checks on the statuses of the components it controls.

Thus, the operations offered by the multimedia objects are directly related with the status supported by the components and so are the events. There are two main different kinds of events for composition purposes: *deterministic* (mostly related with annotations) and *non-deterministic* (all the others). Deterministic events are strongly bound to objects and always occur in the same way each time the object is played. Non-deterministic model actions that might or not occur (such as user intervention). Figure 4b shows a multimedia object with some deterministic events (bigger arrows) and some non-deterministic ones which can be triggered by buttons inside the object, or system features such as changes in the quality of service.

Figure 4(a). Simplest MM_Object in terms of composition

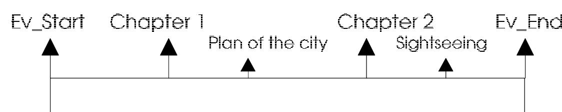

Figure 4 (b). MM_Object with different types of events

5. TYPE MANAGER

The type manager implemented is a general repository of types and could well be implemented using the architecture of the X.500 Directory Service [4]. The main difference towards other proposals for multimedia environments was already stated and is the decision of not using the inheritance technique of object-oriented programming languages. Firstly, because inheritance relates types binding the meaning of reusage of code to it. This aspect was avoided in [8] for instance, by defining all upper classes as abstract and, in fact, what is being inherited is specification of interfaces. Secondly, because there is the need to understand relations in a wider scope relating entries belonging to type spaces eminently different. Why then should a single kind of relation be overloaded with different meanings depending on the objects, instead of just having a different relation for each case? The meaning of a subtype relation will be shared by all entities that handle it. Finally, because if an object-oriented programming language type model was adopted, access to type information at system level

would have to be made using the specific language or a replica of the model would have to be created maintaining all the semantics defined in the language.

The current repository of types has type names as entries and defines a set of attributes for them: attributes are seen as property data related with the types (a default geometry or configuration for multimedia objects, for instance), and as types of relations between the types to support compatibility rules (e.g., a certain component can implement a certain multimedia object, or two data streams can be connected).

The type manager is used by the entire system: the graphical editor uses the information stored in the type manager to know about the existing types of multimedia objects available to the author, and to validate variations to the existing objects made by the author; the compiler uses the type manager to perform consistency verifications during the compilation, so as to prevent most of the errors to occur at run-time; the interpreter of the state machine, and the run-time system use the type manager to know the specific codes assigned to the types used and to perform run-time checks. The type manager is also used to get signature information about events or operations which are not built into the system (because they were defined at a later stage).

The type manager stores information about the following types:

- multimedia object types
- component object types
- data stream types
- status types
- event types
- operation types
- annotation types

The information on operation types, for instance, is very similar to the one existing in computational models such as ODP.

6. COMPOSITION MODEL

In this approach there is only one composition model with the distinction between the control part of the application, which uses events and operation invocations; and the data part, related with the flow of the data streams and component configuration.

Events and operation invocations (which were restricted to one way flows of information - requests without replies) are atomic actions in the composition model. A model with such characteristics can be appropriately specified with a process algebra type of language. A first proposal for a language, described in [22], used an adaptation of LOTOS [3, 14] taking advantage of its aggregation between data types and process algebra. A discussion of the advantages of such a choice is out of the scope of this paper. Some of the disadvantages motivated the definition of different operators with semantics more suited to the specific

requirements of multimedia interactions. This is a currently on-going work and will be reported in the near future.

The same approach of the object model is followed here: a general composition model and simplifications for specific views. In order to understand the features of the graphical editor a brief description of the simplified model is done here.

There are basically two kinds of compositions which then have multiple forms by parameterisation: *parallel* and *sequential* composition. Parallel composition relates intervals in terms of their lifetimes (a lifetime of an interval is the period when the interval is active, for instance, an object playing). A lifetime can possibly be infinite if there is a loop inside. The parallel operator can be parameterised:

- in terms of the semantic of termination - the parallel finishes when the first finish, or the last finish, or a disjunction (or conjunction) of a certain set of operands finish;
- in terms of the influence in the duration of the objects - all objects keep their natural duration, should change to the duration of the biggest, or the smallest, or to the duration of a certain operand;
- in terms of inter-influence between the operands - while the relation is active they should exchange information about a certain status (speed, geometry, etc.)

Sequential composition is a way to express causal relations. It is a view of the composition based on points. The semantic states that a certain condition is only active after another condition happens. In practice it is a waiting state for an event from another object.

There is an important feature associated with composition which makes the editor extensible to status it does not know about. A sequential relation can be just that (a relation), in which case the object is waiting to receive the corresponding event from another object; or it can have an action associated with it, in which case the event is not sent but the action is called. This feature works in a similar way for the parallel case. Most of the other proposals bind a meaning (start or play) to a sequential relation. Thus, inter-influences can be expressed by just having the status, in which case all events associated with the status circulate between the operands, or by denoting an action (belonging to the operational interface of the status), in which case it is the operation that is called.

7. VISUAL COMPOSITION

A graphical editor was built to ease the task of application authoring. The concepts were restricted to the essential and techniques of drag-and-drop were used. Figure 5 shows the appearance of the editor with a small specification written in it.

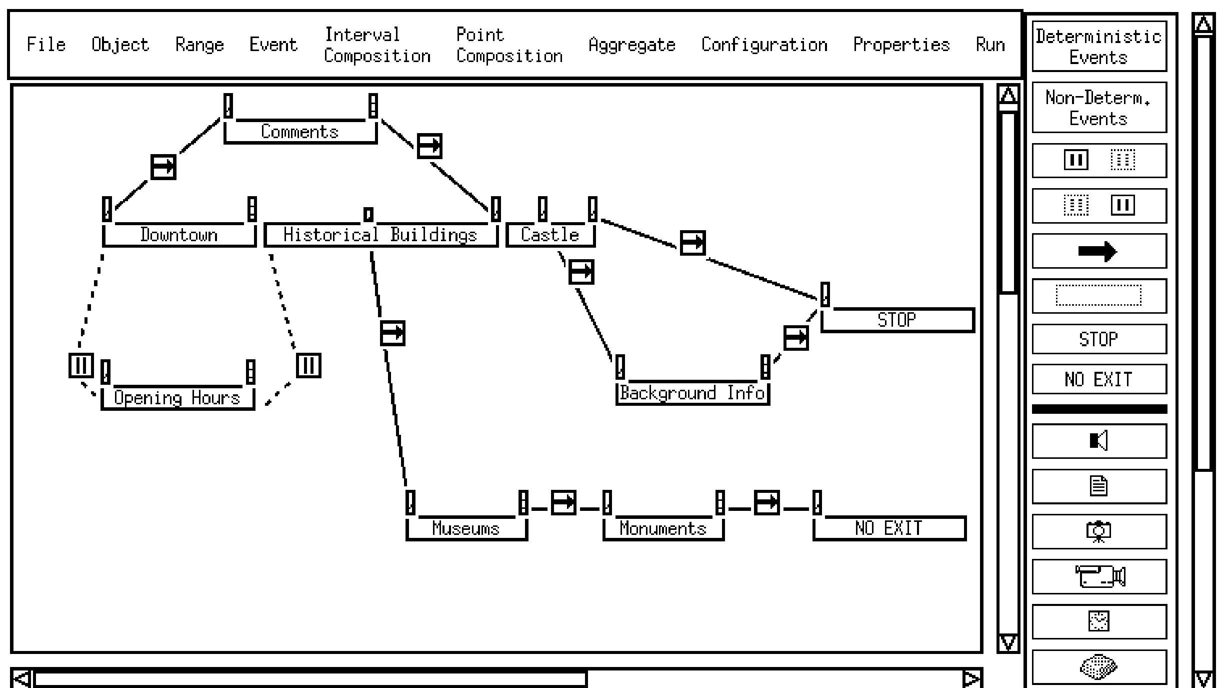

Figure 5 - Graphical Editor

The editor consists of a command menu bar, a working area, and a palette for events, operators and the most common multimedia types. The author begins by dragging a multimedia object from the palette (lower part, beneath the black separator) and dropping it in the working area. The new object gets a default configuration from the type manager. Customization can be made using the *Configuration* and *Properties* menus (*Configuration* permits alterations to the sources, sinks and converters to be used, and the setting of their topology. *Properties* is used to define the name of the files, the various geometries, etc.). The dropped objects get their *Ev_Start* and *Ev_End* events automatically. Objects can have *ranges* defined in them. A range was a syntactic feature to make the algebraic expressions in the language more readable, but was maintained. Ranges are separated by annotated events (deterministic nature). Further events can be inserted on the objects by dragging them from the palette. Deterministic events have a bigger icon than non-deterministic ones.

Parallel and sequential composition are also created with the drag and drop technique and then parameterised by the menus *Interval Composition* and *Point Composition*. The menus *Object*, *Range* and *Event* are used to do simple operations such as moving icons, changing their sizes or names, or deleting them. There is an aggregation feature to combine different objects in a composite one (a way of refinement). Again the drag and drop technique is used and the menu *Aggregate* sets the corresponding relations between events belonging to internal objects and those defined to be visible to the exterior. Finally, the menu *Run* is used to interactively run the specification.

The example shows a piece of video about a town, showing the downtown during a first portion, then historical buildings followed by a *traveling* in the castle. The parallel between downtown and opening hours specifies that a text containing the opening hours of the shops should be displayed while showing downtown. Some audio comments about the downtown and the historical buildings are heard. The video only starts the castle sequence after the comments are finished (the reference to the event in the video is just a way to reference a point in the video object). When the castle part is shown, a certain point deserves some audio explanation of background information. This is specified by a sequence operator from that deterministic event. Finally, there is a non-deterministic event which is active while the historical buildings are shown (The precise point in the object is irrelevant, and the notation only means any point between the two adjacent deterministic events). Every time the event happens (if it does at all) textual information about museums and monuments is displayed.

8. RELATED WORK

The topics of multimedia composition and object modeling are usually discussed separately and most often the contact points reside on the identification of the necessary interfaces to support the composition model. The main reason is that composition is mainly understood as simple temporal (and sometimes spatial) synchronization. If logic composition is also wanted or if some decision taking is dependent on the type of the objects both topics must be defined in the same framework. A representative example of such an environment is [17].

In the topic of multimedia composition alone most of the systems reported show certain limitations that the proposal of this paper overcomes. Process algebras decouples the description of the application from any predefined structure such as a hierarchy [25, 20], or a timeline [5, 10, 8] widening the scope of applicability to different kinds of applications. Having logical events as atomic actions creates the ability to perform future editing facilities on components which are hard (or impossible) to achieve when using Petri Nets [16, 21, 26], timelines, synchronous process algebras [27], or final formatted notations such as MHEG. [9] is an example showing a peculiar way of supporting editing facilities, based on the TEX´s glue concept. It should only be seen as another proposal for object composition algorithms as it lacks other aspects of multimedia composition to be considered a complete multimedia framework.

In the topic of multimedia object modeling most of the proposals try to *completely* define the interfaces of the objects, making the model static with little possibilities for futures extensions or modifications. Nevertheless, constant evolution is a key issue in multimedia. The object-oriented technique is the most popular choice using inheritance [18] or specializations and meta classes [15]. Such choices make the framework vulnerable to major disturbances if the changes are considerable. *Constructive* approaches can simply create new

versions of objects out of the set of available interfaces and still work with old ones, as the type information is outside the object. A notable example of such an approach is similar to the one in this paper [7] and objects are defined as sets of specific interfaces. It lacks however a connection to a type model in order to use that information in the tools. The MADE project [2] goes further in the definition of objects and provides visibility of features available inside of the objects (e.g. concurrency mechanisms). It is perhaps a far too long way in specifying composition as it can introduce implementation details. The MHEG notation can fit well in a constructive framework because it only states how actions can be defined, and the user is free to define the ones of his needs.

[18] integrates both topics. However, three composition models were defined (data flow, activity and temporal) with some overlapping of concepts. This can make consistency checks difficult. The data flow composition model merges what was called here data steam types and part of the control data. The temporal composition model uses a timeline based technique (which has inherent limitations) to determine durations of objects. It conflicts with the reference point based description used in the activity model (this one broadly similar to the one of this paper). The choice in this paper was to parameterise the parallel operator with indications of relative size of operands, keeping it concise with the reference point approach.

Finally, visual composition was proposed in [17] using a point based notation, and a less rich set of choices of compositions (for instance the sequential composition is always associated with the action *start*). Petri net based visual programming can also be found in [6].

9. CONCLUSIONS AND FURTHER WORK

This paper proposed a general solution to create frameworks for the construction of distributed multimedia applications. The constructive approach based on *status* in conjunction with a system wide type schema make the system extensible and sufficiently general for a wide range of distributed applications. The existence of conceptually simple views of the framework permits an easy environment for the construction of specific kinds of applications and still reuse general multimedia components in the system. The type schema proposed requires very little from the objects allowing the usage of standard objects such as MHEG, or similar (providing a minimal shelf is built). The existence of type information at system level allows the tools of the framework to produce correct specifications of the applications and is by no means a restriction to the process of software construction. The meanings of type, subtype relation and compatibility rules are valid in a smaller scope, only shared by the components of the system which have to handle them. The information is kept outside the objects not influencing their construction.

Important issues are still unfinished and not entirely clarified. The prototype graphical editor needs some completing work in aspects of interactive test of the specification. Data streams are still implemented using RPC operations instead of new concepts of continuous data supported directly by the infrastructure. There is little integration between the concept of status and controlling operation interfaces on the data stream types. If statuses can also be defined inside the controlling part of the data streams the composition can be richer.

Issues not covered so deep in this paper which need more clarification are: the complete definition of a script language fitting the generality of the object model; the concrete experience in using high-speed networks covering issues such as quality of service, time resilience of data compressing algorithms; topologies different from 1:1; and dynamic reconfiguration of data flows and topologies.

REFERENCES

1. ANSAware 4.1. *System Programming in ANSAware.* Doc. RM.101.02. February 1993.
2. F. Arbab, I. Herman, G. Reynolds. An Object Model for Multimedia Programming. *Eurographics'93*.
3. T. Bolognesi and E. Brinksma. Introduction to the ISO Specification Language LOTOS. *Computer Networks ans IDSN Systems*, 14:25-59, 1987
4. CCITT Rec. X.501 (ISO/IEC JTC1/ SC21 ISO 9594-2), The Directory Models, March 1988.
5. S. Christodoulakis ans S. Graham. Browsing within Time-Driven Multimedia Documents. *Conference on Office Information Systems*, pages 219-227, March 1988.
6. S. B. Eun, E. S. No, H. C. Kim, H. Yoon, S. R. Maeng. Specification of Multimedia Composition and a Visual Programming Environment. *ACM Multimedia 93*, pp. 167-173.
7. G. Flurry. Multimedia System Services for a Distributed Environment. *Proceedings of the 4th Workshop on Oper. Syst. and Network Support for Digital Audio and Video*, Nov 1993.
8. S. Gibbs. Composite Multimedia and Active Objects. *Proceedings of the OOPSLA'91 Conference*, pp. 97-112. Association for Computing Machinery, 1991.
9. R. Hamakawa and J. Rekimoto. Object Composition and Playback Models for Handling Multimedia Data. *ACM Multimedia 93*, pp. 273-281.
10. M. Hodges, R. Sasnett, and M. Ackerman. A Construction Set for Multimedia Application. *IEEE Software*, January 1989.
11. ISO/IEC JTC 1/SC 29/WG 12. Information Technology - Coded Representation of Multimedia and Hypermedia Information Objects, February 1993.
12. ISO JC1/SC17/WG8, SMSL. Standard Multimedia/ Hypermedia Scripting Language.

13. ISO/IEC JTC1/SC21/WG7 N755, *Information Technology - Basic Reference Model of Open Distributed Processing - Part 1: Overview and guide to use*, January 1993.

14. ISO 8807. - Open Systems Interconnection: LOTOS - A Formal Description Technique based on the Temporal Ordering of Observational Behaviour, 1987.

15. W. Klas, E. J. Neuhold and M. Schrefl. Using an object-oriented approach to model multimedia data. *Computer Communications* vol 13, no 4, May 1990, pp.204-216.

16. T. Little and A. Ghafoor. Synchronization and Storage Models for Multimedia Objects. *IEEE Journal on Selected Areas in Communications*, 8(3):413-427, April 1990.

17. V. de Mey *et all*, Visual Composition and Multimedia, *Procedings Eurographics '92*.

18. V. de Mey and S. Gibbs. A Multimedia Component Kit. *ACM Multimedia 93*, pp. 291-300.

19. Microsoft. *Microsoft Multimedia Development Kit. Multimedia Viewer Developer's Guide*, 1.0 edition, 1991.

20. S. Newcomb, N. Kipp, and V. Newcomb. The HyTime: Hypermedia/Time-based Document Structuring Language. *Communications of the ACM*, 34(11):67-83, November 1991.

21. N. U. Qazi, M. Woo, and A. Ghafoor. A Synchronization and Communication Model for Distributed Multimedia Objects. *ACM Multimedia 93*, pp. 147-155.

22. P. F. Pinto. An Interaction Model for Multimedia Composition. PhD thesis, University of Kent at Canterbury, 1993.

23. P. F. Pinto, P. F. Linington. A language for the specification of interactive and distributed multimedia applications. *Int. Conf. on Open Distributed Processing 1993*, pp. 217-234.

24. P. F. Pinto. Interface Definitions for Multimedia Interfaces. Palantir Internal Report n.092, University of Kent at Canterbury, October 1992.

25. J. Postel, G. Finn, A Katz, and J. Reynolds. An Experimental Multimedia Mail System. *ACM Transactions on Office Information Systems*, 6(1):63-81, January 1988.

26. B. Prabhakaran and S. V. Raghavan. Synchronization Models for Multimedia Presentation with User Participation. *ACM Multimedia 93*, pp. 157-166.

27. J. Stefani, L. Hazard, and F. Horn. Computational model for distributed multimedia applications based on a synchronous programming language. Computer Communications, 15(2), Mar 1992.