Open Distributed Processing, II (C-20)
J. de Meer, B. Mahr and S. Storp (Editors)
Elsevier Science B.V. (North-Holland)

A language for the specification of interactive and distributed multimedia applications

Paulo F. Pinto[a] * and Peter F. Linington[b]

[a]Inesc, R. Alves Redol 9, 1000 Lisboa, Portugal, E-mail:pfp@inesc.pt

[b]Computing Laboratory, University of Kent, Canterbury CT2 7NF, UK,
E-mail:pfl@ukc.ac.uk

This paper describes a model for distributed multimedia applications and a specification language based on this model. The applications involve the composition and synchronization of multimedia objects, and their interaction with the user and the environment. Objects are autonomous entities which have a behaviour, in terms of the set of operations they offer to the environment; the mechanism for synchronization with these objects is based on the communication of typed events. A single mechanism integrates user interaction with run-time control of the distributed system, allowing a natural interplay between them.

The language specifies compositions by exploiting the concepts of the model. It captures the characteristics of multimedia interactions using an adaptation of process algebras, and includes a (procedural) functional part to define data structures and to provide consistency checks.

The prototype system implemented consists of a compiler which translates the language expressions into a state machine, and a central interpreter which orchestrates the composition and synchronization of the distributed multimedia objects.

Keyword Codes: IFIP TC6.4; IFIP TC6.1; I.1.3
Keywords: Application Layer Communication Services, Multimedia Services and Systems; Network Architecture and Design, Distributed Systems; Algebraic Manipulation, Languages and Systems.

1. INTRODUCTION

Multimedia data is now becoming common in distributed environments. Current practice, however, has grown out of local solutions and is still quite hardware dependent. Implementations generally consist of specific multimedia libraries linked directly with the applications. What is needed is an integrated family of system components capable of supporting distributed, multi-user designs.

Another issue is the choice of a suitable abstraction for multimedia data; a unique abstraction will not solve all the problems of handling multimedia, because any single abstraction will simply describe systems from a certain point of view.

*This research was partly sponsored by NATO and the EEC-ESPRIT research programme.

An object oriented approach is attractive, but still has some limitations. Objects are related to each other by class hierarchies, which are generally based on implementation concerns, and so provide a particular view. Some common themes in selecting abstractions are [32]:

- application aspects (creating abstractions suitable for a set of applications. E.g. an object which produces images for presentation as part of a document);
- device aspects (objects have a set of device control operations. E.g. objects with operations such as *Play*, etc.);
- media type aspects (objects are organized by the properties or quality of the data. E.g. a *music* object is a subtype of an *audio* object);
- communication aspects (highlighting the communication processes within the system. E.g. objects to represent stored media during transfer).

Choosing a single one of these aspects would clearly solve certain classes of problem but would obscure the analysis of others. Unification of the various aspects to give a single carefully designed set of subtyping relations would generate systems which are too fragile to support major enhancements to the class hierarchies – a situation which is very undesirable when describing multimedia systems which are constantly evolving.

This paper proposes a model for the construction of distributed multimedia applications. It is still not a complete solution, in that it concentrates on certain types of application, but the approach taken is quite general; applications are defined as a set of interacting multimedia objects, so that objects influence each other as the application runs. A typical application would be the presentation of multimedia documents which are specified as the composition of several basic objects.

In selecting abstractions to support the integration of multimedia components, the model takes a multi-faceted approach. Objects are not simply related to each other, by a single aspect, but can participate in various views of the system and each view has its own composition rules and subtyping relations. These relations are recorded independently of the objects concerned by a system-level Type Manager. This maintains a profile for each object, defining its relevance in each view. Any checks on the object (consistency, etc.) can be performed by system entities (compiler, network manager), without reference to the object itself and independently of the programming language in use. The type manager maintains a number of separate type systems expressing the fact that

- object functions are accessed by a typed interface expressed in terms of operations (such as Create, Play, or more application oriented operations like Zoom, etc.);
- objects have a behaviour; they are active, typed, entities, which can provide typed events (sequentially or in parallel) to the environment.
- an object can produce its multimedia signal using various different transfer syntaxes (PCM audio with 8 bit samples, PCM audio with 16 bit samples, etc);

The second contribution of this paper is in the way multimedia behaviour is specified, or programmed. A language, based on LOTOS [4] [14], has been defined to express the flows and types of information. It is strongly typed, in all the available views, and so consistency checks can be performed at compile time.

First, the necessary multimedia composition operators are considered: temporal, s-patial, general purpose, etc. Next, a suitable structural framework for the individual composition relations is chosen, examining hierarchies, graphs, networks, reference lines, etc. Finally, a powerful way of describing the relations for constructing applications is built from the previous facilities.

This paper reviews related work in the area, and then describes the model and the language. It outlines an implementation of the system and draws conclusions. The implementation assumes the Open Distributed Processing computational model as a basis, using object invocations for both data and control communication. This functionality was provided in the prototype by the ISA/ANSA [1] [2] ANSAware system.

2. RELATIONS BETWEEN OBJECTS AND DESCRIPTION TECHNIQUES

Early multimedia systems [33] [8] [27] defined composition mainly at a spatial lev-el. They used hierarchical structures to compose documents and there was no explicit language to describe the placement of the various objects. The placement was performed by completely integrated tools (formatters, editors, etc.) sharing a common notation. On the other hand, temporal dependencies between objects were included in a non-integrated way. These objects were activated explicitly (by user buttons) and the synchronization performed only at the beginning and at the end of the object. MINOS [8] presented some further mechanisms for synchronization, defining composed data types which provided predefined compositions of basic types.

Hierarchical structures are widely used because they provide simple descriptions, which are easy to handle, and can be related directly to the process of stepwise composition or refinement. It was thus natural that these kind of structures should be adopted to describe temporal synchronization [28] [23]. Nodes in the tree are either basic objects or denote some relation between descendents. Typical relations are sequential and parallel presentation of descendents. Hierarchical structures, however, have some serious limitations. Firstly, the node is the atomic unit of synchronization and if intra-object relations are needed the object must be divided into components. This destroys the concept of a single abstraction for the object and prevents it from being used in other contexts with different styles of synchronization. Secondly, the hierarchy describes the composition from a certain viewpoint (a document with chapters, sections, etc.). When other kinds of description are desirable, such as a picture being displayed while there is text referring to it, the hierarchical description will have to be made using another viewpoint.

The new problems that temporal objects introduce, both in terms of new styles of composition and in terms of new application structures to handle them, has lead to a number of proposals. Unfortunately, almost all of them restricted composition to temporal synchronization, making the interplay with other relevant aspects of a system difficult (consider management of bandwidth of connections, user interaction, relations with static data, etc.). Two main approaches to temporal synchronization are [3] the **reference line** and **reference points**.

Synchronization based on a **reference line** [7] [11] [9] is a better approach than the use of hierarchical structures. It works by independently attaching all objects to a time line. Positive aspect are the lack of a fixed structure for relating objects, making it easier to keep a single abstraction for each one, and to reuse objects in different applications. However, the approach has limitations in situations where changes are made to the predefined scheduling. Actions, such as *Pause*, or *SlowDown*, are either performed within one object, leading to inconsistency of the composition, or on the interpretation of the reference line, affecting all active objects simultaneously. There is nothing in between. Because of this, solutions based on a time line are weak when modeling non-determinism. It is difficult, for example, to specify user intervention that *might* happen, or to model implementation delays or delays due to the distribution, or from user intervention changing the duration of an object.

Synchronization schemes based on **reference points** produce systems which solve all these limitations. In this approach objects generate control events which are related in some way to the multimedia signal they produce [16] [24] [30]. A specification of synchronization will relate the various events in a causal way. This approach still allows for modifications of objects to take place after the synchronization specification has been written.

Events provide a versatile system which can be used to integrate types of composition other than just temporal synchronization. If all synchronization activity is represented as events, multimedia applications become just special cases of concurrent programs.

However, synchronization based on reference points assumes that the objects generate events in a consistent way. Where necessary, objects must be constrained to generate certain series of events in their proper sequence. The solution is to have a language (and a supporting framework) which specifies the constraints and allows the checking and correction of any inconsistencies.

The choice of a suitable notation for the **specification of object composition** is another important issue. The notation must be able to express the various compositions easily. Some proposals already exist in the literature. In Muse [11], composition was considered in a general form, and not limited to the temporal aspect. It was recognized that there should be different ways to describe it: graphs and networks are used for coarse-grain and static composition; editors of reference lines express fine-grain synchronization; and *bindings* express relations between entities. All the structures are grouped together in a data model; however, they still form fairly disjoints sets.

The other proposals focus on temporal composition. They use formal, or quasi-formal, notations which can be parsed, or interpreted, by a scheduling manager. Although these notations are suitable for the restricted problem of intermedia timing, they are also used in other contexts with less success, such as in specifying user intervention and describing distribution.

One example is [20] which uses a Petri net based OCPN model to specify dependencies of intervals. It is suitable to program low-level synchronization but it lacks structuring support to solve more context-dependent synchronization: the aggregation concept for a Petri net is hard to visualize; it is difficult to associate type information with a set of nodes in the net (indicating, for example, that the type is video); it is also difficult to model user intervention that is possible, but not necessary, in certain intervals; and it does not support evolution of objects or time coercion facilities. In [21] a second notation (hierarchical) is used for spatial composition. The use of a second notation demonstrates the difficulty of specifying compositions other than temporal compositions using only one notation (or data model).

Another example is [12] where path expressions are used to describe synchronization of objects. This proposal is presented as an extension to the hierarchical document architecture, ODA [5] [13]. Path expressions can be seen as equivalent to Petri nets under certain conditions [19] and this approach shares the same drawbacks as the one listed above.

A different approach is the use of process algebras [10] [22] to describe compositions. Their suitability for the specification problem is assessed further below, but one use has already been reported in [30]. It uses a synchronous language, Esterel, to describe temporal synchronization between objects, based on events. The synchronous nature of the language also makes it suitable for low-level synchronization, producing clearer specifications than the Petri nets or path expressions because of the greater expressive power of the algebra. However, there are some limitations if a higher level description of the application is wanted, involving aspects other than just synchronization: communication between objects, more complex relations between parts of the objects than causality of time events, and specification of user intervention. The main limitations are inherent in the language used: the non-existence of an abstract data type component, particularly important for multimedia; and the difficulty of generalizing the operators and signals (events) to arbitrary types of interaction and data.

The use of process algebras enhanced with an abstract data type component, as in LOTOS, and with modifications where relevant to suit the needs of multimedia applications, avoids most of these limitations and provides a powerful framework to describe interactions at a higher level.

3. SUPPORT FOR DISTRIBUTED MULTIMEDIA APPLICATIONS

The modeling approach taken here to provide support for Distributed Multimedia Applications was to define an integrated system comprising a model, a language and tools[29]. Figure 1 illustrates the approach.

- The model contains (i) in the information viewpoint, an application level concerned with the definition of the various concepts, the components to implement them, and the way they interconnect; (ii) in the computational viewpoint, the invocation mechanism, the concurrency support, etc; and (iii) in the engineering viewpoint, the communication mechanisms used.

- The language reflects the capabilities of the model and conceptually can be divided into a configuration part with the definition of the elements and their configuration, and a specification (or programming) part which uses process algebra to describe the compositions. This specification part is expressed in terms of the concepts of the model and the configuration part makes the bridge between these concepts and the engineering objects that will implement them.

- The tools part of the system is a particular implementation of the model/language using a particular technology. It includes compilers, run-time managers, editors, etc. The prototype implementation used a *basic run-time system*, a *compiler*, a *run-time interpreter*, *editors*, *producers* and *consumers* of multimedia data.

4. THE MODEL

Most of the novel features of the model are in the application level. An object-based methodology was used for the description of the application because the types and processes of the language can straightforwardly be considered as objects and because encapsulation gives independence of the internal representations of the objects. The basic entity, a **multimedia object**, is half way between a data type view and a process view of information.

> *Definition:* A **multimedia object** is an active entity that conforms to a certain **behaviour** and may produce a medium-specific **stream**. The behaviour is represented by the events the object can generate and the operations it offers at its interface. The stream is characterized by a model of the medium (the signal), including the properties which control its transmission and presentation.

There is, in practice, an engineering sequence which breaks up the continuous nature of a multimedia signal (into bits, pages, frames, bars in a chart, cars passing by, etc.). A single abstract medium is capable of a virtually unlimited range of representations. Synchronization between objects is performed by *events* which can be associated with any of the possible representational sequences, providing a media-independent way of temporal labelling. The mapping between the sequence and the events is then a private matter for the object.

MM_Object – the producer

The computational concept associated with multimedia object is called an MM_Object. An MM_Object can produce (display) one or more multimedia objects, which are instances of its type. An MM_Object can be seen as composed of zero or more end components (*Sources* and *Sinks*) and a manager (see figure 2).

MM_Objects encapsulate the flow of multimedia signals, providing a control view of the distributed application. For instance a *Text* MM_Object generate text and can send events which relate to pages, mouse clicks, etc. A video-conferencing object has implicit synchronization of the various media and can produce events to trigger other application components (for example, to request the sending out-of-band documents). The specification is only concerned with stating the type of the signal flow and the type of the control information. The MM_Objects is a useful abstraction because it provides a single common way to work with any kind of multimedia data. Furthermore, when associated with a dynamic type system and an automatic generator of interface bindings the specification environment can work with a constantly evolving system of multimedia components.

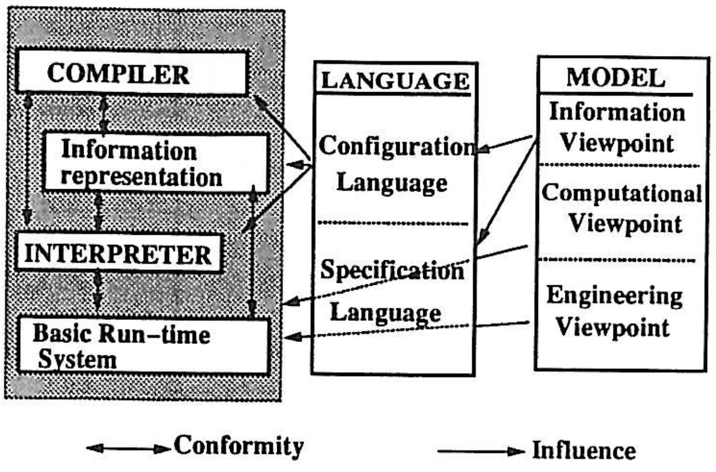

Figure 1. Integrated model, language and tools

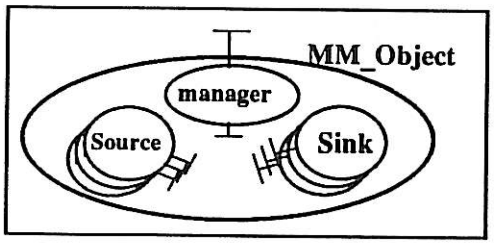

Figure 2. Conceptual structure of a MM_Object

Sources, Sinks and Convertors – Building Blocks

All MM_Object features are provided in a less media-independent form by specifying sources and sinks. Sources and sinks also have public interfaces which follow a common style [26] but their details are specific to the media types (for example, particular type of events are generated by the sink). They represent multimedia objects in an open distributed environment and can be used either via the concept of an MM_Object (as proposed here), or independently. Sources can be connected to sinks (using a network connection if necessary) either directly, if their transfer types are compatible, or via convertors which make them so. Transfer types define not only the representation media but also the way control over the transmission of the data is performed. This control can use a private communication mechanism or it can use events, making it suitable for inter-object synchronization.

Event – the interaction element

Typed events are the basic elements of interaction. Forcing events to be typed simplifies the writing of applications by creating windows of visibility over an otherwise flat event space, while at the same time helping to check the correctness of the system. Events (and their types) can fall into three different categories: object annotations, user intervention, or those generated by the platform, including exception handling. The integration of all these categories provides a versatile system, where all the *properties* which can be used for synchronization are treated in the same way. A *property* is some basis for synchronization; that is, it is anything that can be relevant to other objects. Examples are the rate of presentation of an object, the geometry of a window, the bandwidth of a network connection. In practical terms a property can be anything the object creator wants. Alternatives to this solution can be based on the attachment of some behaviour to the data types. For instance, in [31] and [3], exceptions are integrated into the presentation modes, thus using a limited number of built-in behaviours.

Figure 3 shows the various major event types that an object can send. Annotations are markers in the stream which were added with the help of special medium editors. The specification selects which of the defined events it wants to be visible.

One important category of events is called *ev_structural* and is used to subdivide an object, identifying internal time ranges. These ranges are not intrinsic to the objects and are only syntactic concepts in the language used for synchronization. *ev_structural* events are ordinary annotation events which are given a special status by a particular specification (figure 3 shows four ranges).

Only two events are required in all objects: the initial and final *ev_structural* ones – *Ev_Start* and *Ev_End*. This weak minimum requirement allows for the integration of objects that are not especially annotated for detailed synchronization. The source (or sink) can produce the *Ev_Start* and *Ev_End* events without any need to change the data signal. Interaction with such objects is, of course, relatively poor. A similar requirement for different levels of participation of the components was also reported in [17].

Policy – way of interaction

A model based simply on the ordering of events is still a poor basis for the expression of composition. It would be better to be able to relate two objects (or ranges) in time, and still be able to say that changes in one particular property in one object should be noticed by the other. A variety of modes of composition may be needed, independent of the object types. Consider, for instance, changes in the presentation rate for a composition of video and audio streams; one mode might always notify changes to the other object; another mode might only notify about Pause or Stop commands, and never for Slow_motion commands (since audio would become incomprehensible); etc.

Policy represents what an interaction based on a certain event type really does. In other words, policies and events dissociate functional interaction from communication mechanism. On the other hand, decoupling policies from MM_Object types makes new objects easier to create because they just have to implement the interface, and not all the interactions that they can possibly engage in. It also allows for different types of interactions with the same object.

5. THE LANGUAGE

The language has a process algebra component and a configuration one. The configuration contains the definitions of the MM_Objects, the definitions of the data sorts to be used in the calculus (int, char, etc) and the configuration of the connections. The association of MM_Objects (or ranges) with processes in the algebra provides a structuring schema to hold type information; this is a feature not seen in approaches using less structured formalisms such as Petri Nets [20] or path expressions [12].

The configuration part was designed specifically to meet the needs of the problem. The process algebra part was based on LOTOS, but modifications were made to express the specific multimedia interactions and to make the expressions clearer and more concise. The major points of modification are:

- In the original language, operators define concurrency and communication in terms of events. In the modified version, the types of events are taken into consideration (forming disjoint sorts), restricting the meaning of some operators. It is possible to relate two processes solely in terms of presentation rate changes, geometry changes, etc.

- Operators are also overloaded with some behaviour, and do not only represent synchronization, so that policy actions can be associated with the arrival of events. This is an effective shorthand to avoid writing the specification of policies explicitly using events.

- There is a distinction between events that *will* always happen and events that *might* happen. Modal expressions were defined to express this last kind of events. These expressions are only enabled when the MM_Object (or a range) with which they are associated is active. They create "regions" in time to allow certain actions. They can model optional user intervention.

The language is able to express compositions using three different paradigms, uniting them all in the same data model.

- Intervals – in this paradigm, multimedia objects are seen as intervals between which relations are established. This is the main view and MM_Objects are used both as data types with operations which can be referenced from the calculus, and as processes engaging in events.

- Pointers – this paradigm allows for the identification of points during the presentation of an object and then a reference (jump) to them from any point in the specification (If some other objects are active at that point they must be repositioned).

- Scripts – with this paradigm user code can be referenced from the specification. This is useful not only to express composition but also to add some special behaviour to an object. The integration is at the level of events, and the script is effectively a specification of an event handler (which may be either interpreted or compiled as an implementation choice).

A complete description of a multimedia interaction is called a *specification*. A simplified form of its structure is shown in figure 4 (the missing parts are *process definition*, which are similar to specification definitions, but provide a scoping feature, and the definition of local names as shorthands for operators. A full description is given in [25]):

A specification has a name, can be stored in a library and can be referenced from another specification. When referenced, it is used in a similar way to any other M-M_Object, including the selection of events to be made visible. All the internal events which can be used for external synchronization are listed in the *event list*. The *parameter list* is used to write parametric specifications. It can contain variables to pass data sorts, events or objects. The inclusion of events and objects allows for the creation of generic specifications that will synchronize internally with different objects. The *functionality* is used, as in LOTOS, both to express behaviour and for value passing, if the process terminates. Acceptable terminations are *noexit*, *exit* and *exit* $(t_a, \ldots, t_n)$, with t_i being data sorts or event types.

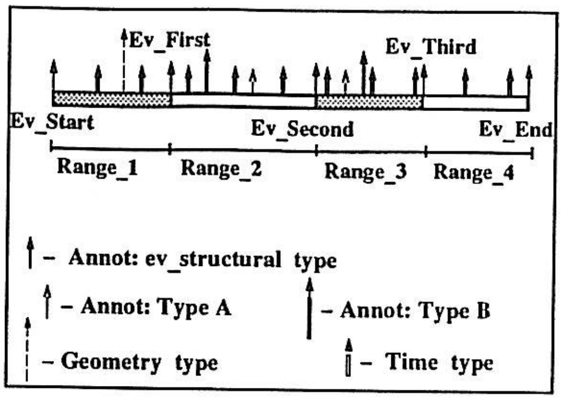

Figure 3. Different types of events

```
Specification Spec_name
     [ event_list ] ( parameter_list )
                 : functionality
 object definitions
behaviour
  main expression;
   AND (
      modal expression;
    OR
    ...
      modal expression;
endspec
```

Figure 4. Structure of the language

5.1. Configuration Part

The configuration part contains statements defining the MM_Object and other specifications used in the application (see figure 5).

Any ranges defined are identified together with the delimiting events. MM_Objects belong to types and share type property data (geometry of windows, volume of sound, etc.) which is stored in the system-level Type Manager. These values can be overridden in the definition statement.

The configuration part also contains statements identifying sources, sinks and convertors. Property data can be used to identify the component fully if the name alone is not enough (this is used for trading purposes, for instance). Convertors are devices which convert one type of data transfer stream to another type. They are placed in the circuit between a source and a sink using the information given in the *link* command.

5.2. Specification Part

This section contains a brief informal explanation of the meaning of each of the operators. A transition semantics was used to describe them in a more formal way [25].

Figure 6 lists the operators of the language. It contains both the traditional process algebra name (used to explain the semantics) and the keywords recognized by the compiler.

The main purpose of the operators is to express the relation between two (or more) intervals, and at the same time specify how each of the objects involved can influence the others. Intervals can be defined by MM_Objects or ranges.

The **interleave** operator, for instance, describes a parallel and independent situation. It states that the operands will run in parallel without any cross-influence, and the aggregated process finishes only when all the operands finish. $(A\_range1 ||| object) \gg A\_range2$ means that *object* and *range1* are played together, and only when both finish can the second range of A be played.

The **parallel** operator is similar to the interleave operator, but the operands can influence each other in some respects. In fact, this operator is a generator of operators. The syntax is $B1 \;\; | \, [t_0, p_0] \, ... \, [t_n, p_n] \, | \;\; B2$ and it means that the events belonging to

```
MMObject Type  Name  [with Sub$Structure
                      range1  "event_ini event_fin"
                      range2  "event_ini event_fin"
                      • • •
                     variable_name    variable_value
                     variable_name    variable_value
                     • • •                        ];
InLibrary           Name  [with Sub$Structure
                      range1  "event_ini event_fin"
                      range2  "event_ini event_fin"
                      • • •                       ];
Source     Type  Name  [with  property_list  ];
Sink       Type  Name  [with  property_list  ];
Convertor  Type  Name  [with  property_list  ];
Link  (object_name,  source_name
                 , | : { convertor_name, ... : }
                              sink_name  );
```

Figure 5. Declarations in the object definition part

OPERATOR	SYNTAX (LANGUAGE)	SYNTAX (COMPILER)
nullary		
inaction	STOP	STOP
successful termination	EXIT	EXIT
unary		
action prefix	g ; B	g THEN B
binary		
sequential composition	B1 >> B2	B1 THEN B2 B1 ACCEPT g IN B2
choice	B1 [ ] B2	B1 CHOICE B2
parallel general case	B1 | [type_0, policy_0] ... [type_n, policy_n] | B2	*user–definable*
interleave	B1 ||| B2	B1 INTERLEAVE B2
interrupted	B1 |> B2	B1 INTERRUPTED B2
modal	B1 AND B2 B1 AND (B2 OR B3)	B1 AND B2 B1 AND (B2 OR B3)
process instantiation	P	P

Figure 6. Type of expressions of the language

the types t_0, ..., t_n will be exchanged between B1 and B2 according to the policies p_0, ..., p_n, as was explained earlier. As the event types and policies are user defined, the parallel operator can relate objects in terms of virtually any kind of properties.

The **disabling** operator in LOTOS is useful to model exception handling, by stating conditions that, when met, will interrupt and terminate a process. In the current proposal exception handling uses the event mechanism, and the interruption features of the LOTOS disabling operator were adapted to provide some priority between the objects. It is used to model premature terminations of objects. For example, a video film clip associated with a page should be terminated when the reader turns the page over. The semantics of the operator states that the interrupting action will terminate all the objects in the current process except any object that is still referenced in the interrupting process. In this way the video film can last several pages while other objects do not. If the interruptable process terminates successfully, the interrupting process follows it in sequence.

The **action prefix**, and **sequential composition** operators are used to describe causal precedences based on reference points, in a more straightforward way.

5.2.1. Structure

Figure 4 shows that the specification part is composed of a *main behavioural expression* and possibly some *modal expressions*. This separation of behaviour expressions was defined to simplify the writing of compilers and to make the description more intuitive.

The *main expression* is essentially based on events of the *ev_structural* type, and states how the application should evolve when there is minimal intervention from the user, or from elsewhere. The expression can also have events of other deterministic types.

The *modal expressions* are each active during the interval in which their guard event is defined. These expressions can also contain events of *ev_structural* type (except the guard, of course). This can happen if an entire object is defined in the expression, as for instance, in situations where the action of a user clicking on some button causes certain MM_Objects to be displayed. If some conditions are valid during the whole of the application, the pre-defined MM_Object *UI* (user interface) can be used. *UI* is always valid.

6. IMPLEMENTATION

A prototype system has been implemented to test some of the ideas of the model and language. The nature of the concepts in the model and the formal semantics of the language allow the control to be exercised in different ways. It can be centralized in an orchestrating entity (the interpreter), distributed in every MM_Objects, or have a mixed structure.

The current implementation uses a centralized orchestrator with some minor control performed directly by the MM_Objects. The language is compiled into a state machine and the orchestrator interprets this at run-time. State machines are the common choice to implement process algebras because they are both described by transition systems and a mapping algorithm can be defined to generate states from expressions [18] [6]. They also provide a powerful operational structure to manage the complexity of the behaviour of parallel communicating processes expressed by the language.

The implementation included three MM_Objects which produced text, image and active labels objects. In the case of the text object, rich text was expressed using a markup notation and is formatted when the objects are activated, using run-time information about the size of the page. This object has a simple structure and does not shift the text to support spatial composition by leaving "blank" areas where overlays can be placed. Provision of this feature would need a definition of a notation about the type of spatial information that objects need to get at creation time. The image object recognizes some image formats and uses a private coding to transport them. It supports the definition of figures as inner regions in the original figure. It is also possible to invoke transformation functions to be executed by the time the object is selected.

All objects are treated as continuous media; that is, they are created instantaneously but the time to transport the data is not nil. This suits the paradigm of intervals in the specification and also decouples data from control. Regardless of the time taken to transport the data, the object is ready to interact at control level immediately after creation (it can even be destroyed before all the data is shown). This is an advantage when the amount of data is significant, as in an image, for instance. In the case of the text object, the text is understood as a sequence of pages displayed with a presentation rate of zero. Pages are turned over when instructed to do so by the controller.

MM_Objects are implemented by independent sources but all the sinks, although independent, are integrated with the orchestrator. This integration allows the use of the same window environment and consequently a better performance for input and output related operations. The X-window system is used for user interaction and object display. The various objects get their input directly and if any input is relevant for inter-object synchronization, events are sent to other sources, sinks or orchestrator.

An MM_Object knows (or is told) exactly how to progress during the lifetime of the object. The knowledge is taken from the specification. Most of the times there is a causal effect between all synchronization events and the actions are just the start of new activities. When it is not the case, ie., an MM_Object has to wait for some event to come and it will arrive late, the MM_Object acts accordingly to the specification, possibly pausing the display of the signal.

The methods available on the MM_Objects are defined to support the necessary control of the application. Naturally, the compiler, the orchestrator and the MM_Objects are all consistent with each other, and the translation of the language performed by the compiler, depends on the interface of the MM_Objects. MM_Objects have a **generic** set of methods with the basic operations: claim a device, select an object, play the object, stop playing, etc. They also have a **specific** set of methods with media-specific operations. These operations are not known to the compiler, and the Type Manager is used to check their correctness. The current implementation uses a special event to transport the operation name and parameters. A better solution would be the creation of a stub invoker at run-time, but the Ansaware 4.0 did not support such facilities. The use of the Type Manager allows the compiler to work with evolving systems.

This kind of approach should be compared to the work of MHEG [16]. MHEG defines content and projector objects with instance data and manipulation functions. They correspond respectively to encoding and presentation functions. The philosophy of the work of the expert group is to define fully all possible multimedia objects, instead of creating crucial objects and letting the world be built around them by using subtyping relations. A noteworthy point in MHEG is the strong distinction between input and output objects. When interactions need both input and output (the most common case) they have to be performed by composite objects (called interactive objects) which have a heavy structure.

7. AN EXAMPLE

Figure 7 shows a working example of a multimedia document. Both the text and the image objects have sensitive areas of two kinds: one addressed at specification level and another that it is not. The behaviour of the object when these areas are pressed is different. In the first case, the object sends an event which has been registered beforehand, while in the second there are only internal actions, which are not used in inter-object synchronization.

This example assumes a user control panel in the application with buttons that send events on behalf of the *UI* MM_Object. Another assumption is that the application starts up, creates all the necessary objects, and then the interpreter blocks waiting for an indication to start interpreting the state machine structure.

In the example, everything is then blocked waiting for an indication from the user. Eventually the button *UI_Start* is pressed. When the event arrives the object *Book1* starts by displaying its first page. When the user turns over the first page[2] the event *P1* arrives and the *map* object (which is an image) is displayed. The presentation stays like this until the user turns over the second page. At that moment the map is terminated and the third page is presented, together with a label.

The *PARALLEL* operator was defined as being a parallel composition in which the

[2]It is a logical page, so it can be a part of a physical page.

```
MMObject Paged_Text     Book1     with Sub$Structure
                                           Second_Portion    "P2  P3"
                                           Third_Portion     "P3  Ev_End"
                                  Data$Id     "text.txt" First$Page  "1"   Last$Page      "4"
                                  Geom$X   "0"       Geom$Y    "0"   Geom$Height "600"  Geom$Width"500" ;
MMObject  Image         Map       with Data$Id     "image/f22454"
                                  Geom$X   "30"      Geom$Y    "381" Geom$Height  "80"  Geom$Width"400" ;
MMObject HS_click_die   Label     with  Label        "A Label"
                                  Geom$X   "430"     Geom$Y    "51"  Geom$Height  "30"  Geom$Width"60" ;
Source    PgdTxtSrc       Source1 ;                 Sink     PgdTxtSnk        Sink1 ;
Source    ImageSrc        Source2 ;                 Sink     ImageSnk         Sink2 ;
Source    SimpleHS_Src Source3 ;                    Sink     HS_ClickDieSnk Sink3 ;
                           Link   (Book1, Source1, Sink1     );
                           Link   (Map,   Source2, Sink2     );
                           Link   (Label, Source3, Sink3     );
DEFINE   PARALLEL  | [ev_geometry, pol_inform] |
behaviour
  UI ? #UI_Start   THEN Book1 ! #Ev_Start   THEN Book1 ? #P1  THEN map  INTERRUPTED
     (Second_Portion  PARALLEL   label)  THEN Last_Portion   THEN   EXIT  ;
  AND (
        UI ? #UI_Abort     THEN   ABORT   ;
        OR
        A := ( map ? x : ev_geometry        THEN Book1 ! x   ;   A ) ;
        )
endspec
```

Figure 7. Example of a specification

synchronization gate is the geometry of the objects. The policy *Inform* sends all changes in the geometry of one operand to the other operand.

The *label* object belongs to a type that has the following behaviour: when it is clicked it sends an event and terminates. The parallel composition finishes when the *label* finishes and the user turns over this page. When both happen, the fourth page is displayed, and when this finishes (ie. the user turns it over) the application stops.

This example only contains objects which are static in time. During their presentation the user can make alterations to the geometry by means of a **geometry panel** provided for that purpose (the window manager facilities were used). There is one such panel with each visual object. If the objects were not static, such as, for example, audio and video, **rate panels** would also be provided to enable the corresponding parameters to be changed.

The last part of the specification is the modal part. The first expression states the fact that the user can abort the program at any instant by pressing the button *UI_Abort*. The second statement says that all events of type *ev_geometry* which are sent from the *map* should go to the object *Book1*.

8. CONCLUSIONS AND FURTHER DIRECTIONS

A first conclusion of this work is that there is clearly the need for abstractions which can handle the diversity of functions that multimedia objects have. These include abstractions at an **authoring level**, demanding only the necessary amount of information to express actual interaction, and abstractions at **engineering level** to support facilities offered all the way through to the communication technology. The nature of the interactions and the nature of the functions performed depend on the level where the abstractions are made. As was noted before, there is no unique solution to the entire problem of handling multimedia data.

The choices made here, however, seemed to cover a wide range of uses. At the authoring level, the definition of event types to categorize the output of objects in terms of control, unifying time, contextual information, user code, etc., seems versatile enough and reduces the hard-wired knowledge that elements have of each other (such as awareness that a certain feature happens 27 seconds after the start of a video clip). The integration of these events into a general communication mechanism opens up interaction with any entity in the system.

At the engineering level, the event selection mechanism, consisting of the registration of interest in types of events, seems general enough to support synchronized behaviour of objects in distributed environments.

In language terms, the concurrency of the algebra makes specifications very simple to read and write. The type of language also provides a property based description rather than a causal one. For instance, the specification of the existence of an event which can happen during a certain interval is straightforward (this is a common situation when modeling user interface). The handling of time becomes quite easy with the concept of process and the way interactions are specified. Intervals are a good abstraction for continuous media and the algebra provides a consistent treatment of actions when objects cease to exist. This should be compared with other approaches, such as the path for event delivery in Hypercard, for instance, which is statically defined and changes suddenly when objects terminate. In Hypercard, this kind of synchronization can only be achieved with events defined at a global level (not in the individual scripts).

The integration of MM_Objects with a dynamic system-level Type Model, is very useful in incorporating specific behaviour into objects, and in allowing for extensions when new versions are created.

Finally, a formal basis for the language proved crucial in building a coherent system. It helped to establish the essential mechanisms clearly, and could form the semantic basis of more user-friendly authoring tools. Graphical interfaces should be considered, but it is important that the various relations are defined in a precise way. One possibility is the use of an adaptation of the formal extension to LOTOS, G-LOTOS [15]. However, pure process algebras do not provide the best authoring environment for specifying some types of multimedia document. It is rather difficult to express additional kinds of information such as spatial dependencies between objects. In this work, the problem was avoided by deriving spatial precedence, of the windows, from the ordering of the operands. The geometry events of the X system were intercepted at application level and integrated in the event mechanism.

Although the main concepts were defined and the implementation proved they are consistent, some clarifications and improvements are still needed. Some of them overlap with the current work on ODP.

- (i) the **interface of MM_Objects** must be completed to represent a true open multimedia object in a distributed environment. Sources and sinks will have to be changed accordingly;

- (ii) ODP already uses the modeling technique of **subtyping** extensively with various meanings: conformance of interfaces in the trader, invocation mechanism during binding and in the computational model, etc. Multimedia applications require such techniques. To make it really usable in a distributed system (by compilers, network managers, etc.) an independent way to store and access type information, and manage the many different meanings bound to subtyping should be defined. Consequently, there is already a strong requirement to have a Type Manager service in ODP;

- (iii) **concurrency**. For the particular case of a specification written in a concurrent language, the direct use of concurrent primitives in a distributed system could be advantageous. These kinds of primitive are still in a very preliminary stage in the current distributed architectures and should be clarified in the ODP work. [1] proposes path expressions to control concurrency but this is not yet implemented in Ansaware;

- (iv) **multimedia access point, MAP**. A clear definition of MAPs is another issue – MAPs are generalizations of the session access points, such as plugs and sockets in ANSAware, to support multimedia streams. This paper has discussed a model which abstracts away from the engineering detail, and there was no strong requirement to define such access points. However, MAPs would require transport mechanisms more suited to bulk data transfer, with various quality of services, than the current operation invocation in the ODP computational model.

REFERENCES

1. APM. Architecture Report: The ANSA Computational Model. Technical Report AR.001.00, Architecture Projects Management Ltd, Cambridge, August 1991.
2. APM. *ANSAware 4.0 System Programmer's Manual*, February 1992.
3. G. Blakowski, J. Hubel, U. Langrehr, and M. Muhlhauser. Tool support for the synchronization and presentation of distributed multimedia. *Computer Communications*, 15(10):611–618, December 1992.
4. T. Bolognesi and E. Brinksma. Introduction to the ISO Specification Language LOTOS. *Computer Networks and ISDN Systems*, 14:25–59, 1987.
5. H. Brown. Standards for Structured Documents. *The Computer Journal*, 32(6), 1989.
6. L. Cardelli and R. Pike. Squeak: a Language for Communicating with Mice. *ACM Computer Graphics*, 19(3):199–204, 1985.

7. S. Christodoulakis and S. Graham. Browsing within Time-Driven Multimedia Documents. *Conference on Office Information Systems*, pages 219–227, March 1988.
8. S. Christodoulakis, M. Theodoridou, F. Ho, M. Papa, and A. Pathria. Multimedia Document Presentation, Information Extraction, and Document Formation in MINOS: A Model and a System. *ACM Transactions on Office Information Systems*, 4(4):345–383, October 1986.
9. S. Gibbs. Composite Multimedia and Active Objects. In *OOPSLA*, pages 97–112. Association of Computing Machinery, 1991.
10. C. A. R. Hoare. *Communicating Sequential Processes.* Computer Science. Prentice-Hall, 1985.
11. M. Hodges, R. Sasnett, and M. Ackerman. A Construction Set for Multimedia Application. *IEEE Software*, January 1989.
12. P. Hoepner. Synchronizing the presentation of multimedia objects. *Computer Communications*, 15(9):557–564, Nov 1992.
13. ISO 8613. Information Processing – Text and Office Systems – Office Document Architecture (ODA) and Interchange Format, 1989.
14. ISO 8807. Information Processing Systems - Open Systems Interconnection: LOTOS - A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour, 1987.
15. ISO 8807/PDAD1. Information Processing Systems – Open Systems Interconnection: LOTOS – A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour – Proposed Draft Addendum 1: G-LOTOS, 1989.
16. ISO/IEC JTC1/SC2/WG12. Information Processing – Coded Representation of Multimedia and Hypermedia Information Objects, October 1991.
17. C. Kacmar and J. Leggett. PROXHY: A Process-Oriented Extensible Hypertext Architecture. *ACM Transactions on Office Information Systems*, 9(4):399–419, October 1991.
18. K. Karjoth. Implementing Process Algebra Specifications by State Machines. In *Eighth International Symposium on Protocol Specification, Testing and Verification*, Atlantic City, New Jersey, 1988. IFIP.
19. P. Lauer and R. Campbell. A description of path expressions by petri nets. In *Conf. Record 2nd ACM Symposium on Principles of Programming Languages*, pages 95–105, 1975.
20. 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.
21. T. Little and A. Ghafoor. Spatio-Temporal Composition of Distributed Multimedia Objects for Value-Added Networks. *IEEE Computer*, 24(10):42–50, October 1991.
22. R. Milner. *Communication and Concurrency.* Computer Science. Prentice-Hall, 1989.
23. 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.
24. C. Nicolaou. An Architecture for Real-Time Multimedia Communicating Systems. *IEEE Journal on Selected Areas in Communications*, 8(3):391–400, April 1990.

25. P. Pinto. *An Interaction Model for Multimedia Composition.* PhD thesis, University of Kent at Canterbury, 1993.
26. P. F. Pinto. Interface Definitions for Multimedia Interfaces. Palantir Internal Report n. 092, 1992.
27. A. Poggio, J. J. Garcia Luna Aceves, et al. CCWS: A Computer-Based, Multimedia Information System. *IEEE Computer*, 18(10):92–103, October 1985.
28. 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.
29. A. Schill. Distributed Application Support: Survey and Synthesis of Existing Approaches. *Information and Software Technology*, 32(8):545–558, October 1990.
30. J. Stefani, L. Hazard, and F. Horn. Computational model for distributed multimedia applications based on a synchronous programming language. *Computer Communications*, 15(2):114–128, March 1992.
31. R. Steinmetz. Synchronisation Properties in Multimedia Systems. *IEEE Journal on Selected Areas in Communications*, 8(3):401–412, April 1990.
32. R. Steinmetz and J. Fritzsche. Abstractions for continuous-media programming. *Computer Communications*, 15(6):396–402, July/August 1992.
33. R. Thomas, H. Forsdick, et al. Diamond. a Multimedia Message System Built upon a Distributed Architecture. *IEEE Computer*, 18(12):65–78, December 1985.