Supporting Real-time Multimedia Behaviour in Open Distributed Systems: An Approach Based on Synchronous Languages

G. S. Blair[1], M. Papathomas[1], G. Coulson[1], Philippe Robin[1], J. B. Stefani[2], F. Horn[2] and L. Hazard[2]

[1]Distributed Multimedia Research Group, Computing Department,
Lancaster University, Lancaster LA1 4YR,UK. E-mail: mpg@comp.lancs.ac.uk

[2]Centre National d'Etude des Télécommunications, 38, 40 rue du Gal Leclerc,
92131 ISSY les Moulineaux, FRANCE. E-mail: stefani, horn, hazard@issy.cnet.fr.

Abstract

There is currently considerable interest in developing multimedia applications in open distributed systems. However, it is now becoming clear that existing architectures for open distributed systems do not support the particular requirements of continuous media types such as digital audio and video. This is particularly the case in the important areas of quality of service support and real-time synchronization. This paper presents results from the Sumo project which aims at supporting continuous media types within the framework defined by the draft Open Distributed Processing standard. The paper advocates the use of synchronous languages within this framework for specifying and implementing real-time synchronization and QoS monitoring. A computational model and the realization of an infrastructure supporting this view are presented.

1. Introduction

Distributed multimedia computing has emerged in the last couple of years as a major area of research. This work is motivated by the wide range of potential applications such as desktop conferencing, multimedia mail and video-on-demand services. Nevertheless, significant technical challenges remain before the potential of distributed multimedia computing can be fully released.

One of the major problems in distributed multimedia computing is heterogeneity. It is likely that most distributed multimedia environments will be heterogeneous, consisting of a number of different workstations interconnected by one or more types of network. The distributed systems community has addressed this problem of heterogeneity by developing platform and language independent architectures such as the Open Software Foundation's Distributed Computing Environment (DCE) and the Object Management Group's Common Object Request Broker Architecture (CORBA). There has also recently been standardization work, initiated by ISO and ITU, to develop draft standards for Open Distributed Processing (ODP)[18]. However, it now becoming clear that such architectures and standards do not provide adequate support for multimedia computing. This state of affairs is primarily due to the fast moving and technologically-driven nature of the field but it is also partly due to the lack of standards frameworks in which research efforts can be structured. The ODP reference model standard promises to be a good starting point in this direction.

This paper addresses the problem of supporting multimedia computing in heterogeneous environments. More specifically, the paper discusses extensions to the ODP draft standard to meet the specific requirements of multimedia (including continuous media types such as audio and video). The main focus of the paper is on the role of synchronous languages in such environments; a novel approach to supporting such languages is also proposed. The paper is structured as follows. Section 2 presents an extended ODP architecture for multimedia applications; the role of synchronous languages in this extended architecture is highlighted. Section 3 then examines synchronous languages in some depth, with particular focus on the synchronous language Esterel. Some requirements to support Esterel are then derived. Following this, section 4 presents a novel approach to supporting synchronous languages in an ODP environment. Secion 5 discusses related work. Finally, section 6 contains some concluding remarks.

2. An Extended ODP Environment

It has long been recognized in the standards community that Open Systems Interconnection standards (OSI) are primarily concerned with communication between end systems. In a distributed system, it is equally important to consider standards within end systems, thus allowing the full functionality of a distributed system to be described. The task of ISO's ODP standardization process is to define a Reference Model which addresses these wider issues.

The complexity inherent in this process is managed by partitioning the standard into five viewpoints: enterprise, information, computational, engineering and technology. Each viewpoint is a complete and self contained perspective of a distributed system in a language appropriate to the target audience. For example, the enterprise viewpoint is targeted towards business managers and organizational analysts. In our research, we have concentrated on the computational and engineering viewpoints. The computational viewpoint is essentially a programming language model of distributed objects and interaction. The engineering viewpoint, in contrast, described how the computational model should be realized at the systems level. In this section, we focus on the computational viewpoint and define extensions to the existing computational language for multimedia. We return to engineering issues in section 4.

2.1 The Proposed Computational Language

2.1.1 The Existing Computational Language

The existing computational language is based on a location-independent object model where all interacting entities are treated uniformly as encapsulated objects. Objects are accessed through interfaces which define named operations together with constraints

Multimedia 94- 10/94 San Francisco, CA, USA

on their invocation. Interfaces are described in an abstract data type language known as Interface Definition Notation (IDN).

Activity takes place in the model when objects invoke named operations in the interfaces of other objects. Interface references are used to create either implicitly or explicitly a binding to the object supporting the interface. Interfaces may also be exported to a special object known as the trader. An object wishing to interact with a particular interface must then import the interface from the trader. At this stage, an implicit binding is created to the object supporting the interface. This process is summarized in figure 1.

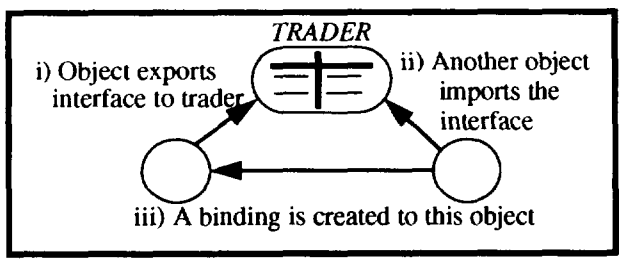

Figure 1 Trading and Binding in ODP

The ODP community has recognized the importance of multimedia and has recently proposed some extensions to the computational language to support the style of interaction required by continuous media communications. In particular, the language now supports the concept of stream bindings which represent a flow of continuous media data between stream interfaces. In addition, quality of service (QoS) annotations can be specified on stream interfaces.

We believe that the above extensions are necessary but not sufficient to support multimedia. From our analyses, we also perceive a need for:

- a more comprehensive treatment of quality of service specification (and subsequent management) in stream bindings, and
- support for real-time synchronization between different activities in a distributed environment.

To meet these requirements, we propose an extended computational model which provides linguistic support for QoS annotations and which also supports the concept of reactive objects. These concepts are described briefly below; further details and justification can be found in [10].

2.1.2 QoS Annotation on Interfaces

We propose that interface descriptions should be extended to include quality of service annotations which enforce constraints on the legality of bindings between interfaces. In more detail, QoS annotations consist of two clauses:

1. a *required* clause describing the level of service that the interface expects from its environment (e.g. other objects or the underlying system), and
2. a *provided* clause describing the level of service the interface can provide to clients.

For a binding to be legal, the two interfaces to be bound must be both type compatible and QoS compatible [22].

In our approach, the individual clauses are written in a real-time logic called QL. This logic allows the service provider to specify the required behaviour (including real-time behaviour) of events pertaining to the object (for example, the rate of emission of invocations from the interface). Further details of QL can be found in [22] which demonstrates how QL can be used to express a range of QoS properties such as throughput and jitter. The paper also considers issues such as the expressive power and decidability of the QL language.

2.1.3 Reactive Objects

Our second proposal is to add the concept of reactive objects to the computational language. Reactive objects are objects which have guaranteed real-time behaviour in terms of their reaction to incoming events and their generation of corresponding outgoing events. They are necessary to provide an element of real-time control in distributed multimedia applications. More specifically, there are two key areas where reactive objects are required in such applications:

1. *real-time synchronization*: reactive objects are required to control the precise timings of events in a mixed media presentation, e.g. to implement lip synchronization between a separate audio and video stream binding.
2. *QoS management*: reactive objects are also required to monitor the quality of service provided by the underlying infrastructure, to react to quality of service violations and to initiate 're-negotiation of the QoS currently provided (note that we are currently investigating the possibility of generating QoS managers directly from QL statements).

In addition, we propose that reactive objects should conform to the synchrony hypothesis (discussed below) and be implemented using a synchronous language. We believe that this style of programming has major benefits in real-time programming. This aspect of the computational language is discussed in more depth in section 3.

2.2 Applying the Computational Language

The current ODP computational language is entirely asynchronous. Objects communicate asynchronously (without time bounds) with other objects. In addition, objects react with indeterminate delay to incoming events. This is not acceptable for multimedia applications. The motivation for the changes to the computational language is therefore to provide more predictable real-time behaviour over both communications (bindings) and selected objects (reactive objects). This view of real-time systems is summarized in the following statement:

REAL-TIME = QoS-CONTROLLED BINDINGS + REACTIVE OBJECTS

In our enhanced language, an application consists of a number of interacting objects some of which are reactive and some of which are non-reactive. Reactive objects provide the necessary level of real-time control in the application. QoS annotations are then used where appropriate to place bounds on the real-time behaviour of communication between objects.

As mentioned above, synchronous languages have a particularly important role in this model of computation in providing a vehicle for the development of reactive objects. The rest of the paper examines this role in more detail and looks at implementation strategies for supporting synchronous languages in a distributed system.

3. The Use of Synchronous Languages

3.1 Motivation and Benefits

Synchronous languages are based on an assumption known as the *synchrony hypothesis* which states that the reaction of a system to external events takes no time. This assumption leads to notations that are well suited for specifying the behaviour of a reactive system [4][16].

Although all synchronous languages rely on the synchrony hypothesis they may come in different flavours. Lustre[7] and Signal [17], for instance, are synchronous dataflow languages. Esterel [5] on the other hand is an imperative parallel synchronous language.

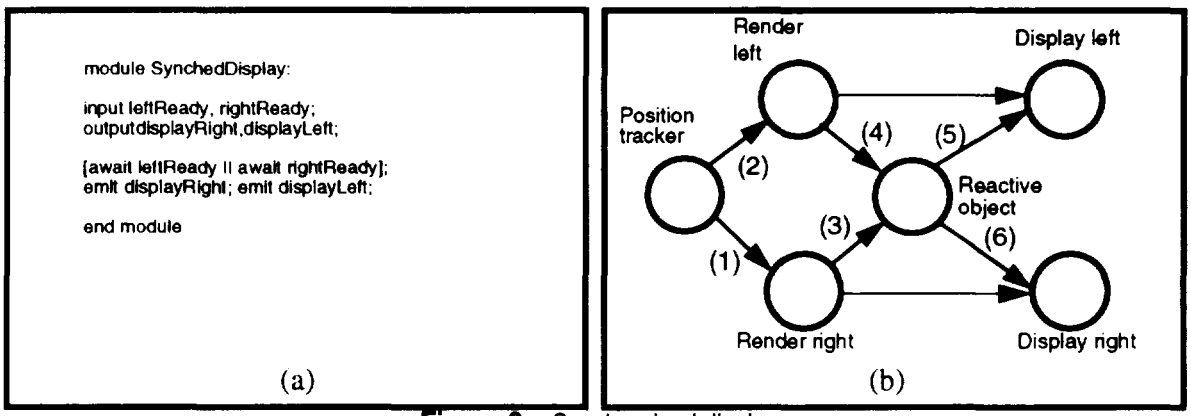

Figure 2 Synchronized display

Some particular benefits from the use of synchronous languages for programming reactive systems are:

- They support high level constructs allowing a concise description of the complex relationships between events governing the behaviour of a reactive system.
- They have a clearly defined semantics with respect to temporal execution and allow the behaviour of reactive systems to be specified formally [5].
- Programs in such languages are deterministic. They can be translated into a deterministic finite automata that can be used for an efficient implementation. The automaton may be also used to formally analyse behavioural properties and derive execution bounds.

Although a particular synchronous language, Esterel, is used in the rest of this paper the issues addressed are general and the research can be applied to any synchronous language.

3.2 Programming Reactive Objects in Esterel

3.2.1 Introducing Esterel

An Esterel program consists of a set of parallel processes that execute synchronously and communicate with each other and with the environment by sending and receiving signals.

The signals present at each instant are broadcast instantaneously to all processes. Signals may carry a value, in which case they are called *valued* signals, or be used just for synchronization, in which case they are called *pure* signals. Several occurrences of a valued signal may be present at the same instant. In this case their values can be combined using a value combination function. If no such function is specified for a valued signal, several occurrences of the signal at the same instant is considered as an error. A program may also communicate with its environment through sensors. A sensor has a value defined at each instant. However, there is no signal associated with a sensor. It is also not possible to get notified of a change to its value.

To comply with the synchronous hypothesis, the execution of statements takes no time. This allows the construction of programs with temporal paradoxes. However, the presence of such paradoxes is detected by the Esterel compiler. For instance, the program fragment below tests for the absence of a signal, **someSignal**, and then emits this same signal. However, as statements take no time, this means that the signal has to be both absent and present at the same instant.

```
present someSignal else
emit someSignal
end
```

3.2.2 Examples

The most interesting and relevant parts of Esterel are the statements used to specify temporal behaviour and synchronization. In this section we present some examples illustrating the expressive power of the temporal statements for some synchronization problems in distributed and multimedia applications. These examples also used to drive the discussion on the runtime support environment for Esterel in sections 3.5 and 4.2. The examples are used for presentation purposes and they are not meant to be complete or realistic.

Example 1: Synchronized Video Display

This example concerns the synchronized display of the left and right eye frames on a stereo head-mounted video display. Figure 2 (b) shows the structure of this application. The "position tracker" object encapsulates a device and the computation needed to track a user's movements and output his/her position in a virtual world. The Render left and Render right objects maintain the model of the virtual world and render the left and right eye frames corresponding to the user's position. We assume that the current position is updated after it has been obtained by both of the Render objects at (1),(2).The frames are rendered in parallel on two separate workstations and time taken for rendering may vary for each workstation and frame. Figure 2 (a) shows the Esterel code of the reactive object that is used to synchronize the display of the left and right images. The signals **leftReady** (4) and **rightReady** (3) are used to communicate with the Render components. **leftReady** is sent by the Render left object whereas **rightReady** is sent by Render right to signal that the corresponding frame is ready. The parallel statement, ||, used in the program terminates only after both await statements have terminated. Then, the two emit statements are executed simultaneously, according to the synchronous computation model their execution takes no time, and the signals **displayLeft** (5) and **displayRight** (6) are sent to the Display left and Display right objects for displaying the corresponding frames.

Example 2: Video Decompression and Playback

In this example, compressed video frames are read from the disk, are decompressed and displayed at a fixed frame rate. Figure 3 shows the structure of the application. Object (a) reads compressed video frames into a buffer in memory. The compressed frames are decompressed by object (b) and displayed by object (d). The reactive object (c) coordinates the operation of the other components to maintain a fixed frame rate. From time to time decompression of a frame may take longer than the inter-frame display period. In this case the previous frame or a special frame are displayed instead.

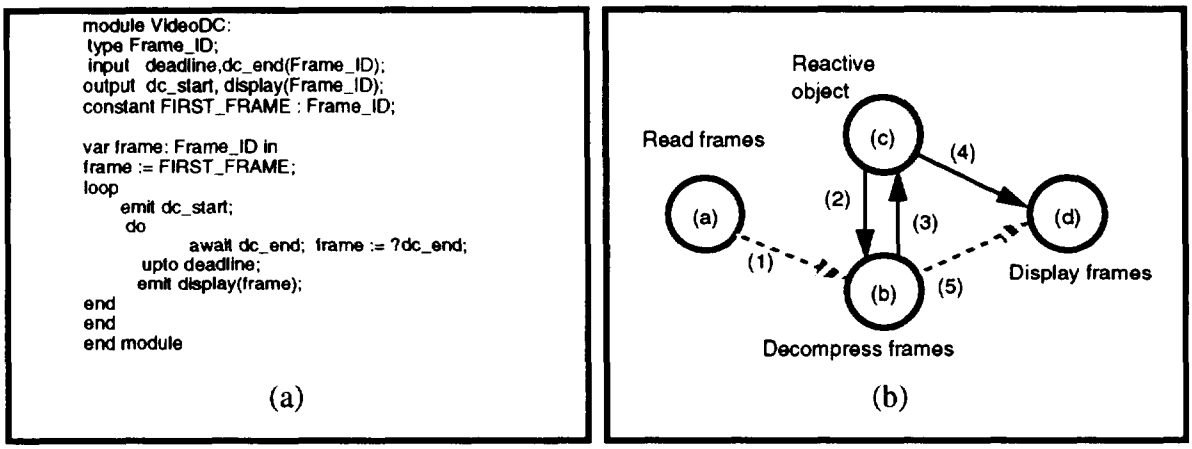

Figure 3 Video decompression

The program of the reactive object is shown in figure 2(a). We assume that object (a) has already read some compressed frames into memory when the reactive object starts executing and that it keeps doing so, so there are always some frames in memory. (1) and (5) represent the data flow between the attached components which could be realised through using shared memory. The signal **dc_start** (2) is sent to object (b) to start decompressing a frame. When it has decompressed a frame it sends the signal **dc_end** (3) with a value identifying the frame. The signal **deadline** is received when it is time to display the next frame; then (c) sends the signal display (4) to object (d). The value of the **display** signal identifies the frame to display. The "do-upto" statement is used to make sure that frames are displayed precisely at the required rate indicated by the receipt of the **deadline** signal. If decompression finishes before the occurrence of the **deadline** signal, this statement delays the execution of the program until it occurs. If not, it aborts the statement executed in its body when **deadline** occurs. The value that has been assigned to the **frame** variable controls whether a frame decompressed in time or a previous frame is displayed. The id of a decompressed frame is obtained as the value of the **dc_end** signal, **?dc_end**, or at the beginning of the program by the constant: FIRST_FRAME.

Example 3: Observing the relative timing of events

Here, we examine two short examples to illustrate the difficulty of determining whether external events should be presented simultaneously to an Esterel automaton.

Figure 2(a) shows an example of a program controlling a system used for a quiz game. After a question has been asked to two users, the one who pushes a button first gets the right to answer. The system determines which user was first and indicates the outcome. Note that it is also possible for the users to push the buttons simultaneously, in which case we have a tie.

The signals **button1** and **button2** are used by the environment to indicate that the respective user pushed his button. The multiple await statement used in this program causes the program to wait until an instant when the signal in some of the cases are present. This statement is deterministic: if the signals for several cases are present the first one is chosen. To keep the example simple we do not treat the case where none of the users push their button.

Figure 2(b) shows an example of a door control system requiring that two buttons be pushed simultaneously to open the door. The signals **button1** and **button2** are sent by the environment when the respective buttons are depressed. The signal **open_door** is sent to the environment to open the door.

```
module Quiz:

input  button1, button2;
output user1, user2, tie;
loop
  await
     case [button1 and button2] do
          emit tie;
     case button1 do
          emit user1;
     case button2 do
          emit user2;
  end
end
end module
```

```
module DoorControl:

input  button1, button2;
output open_door;

loop
     await [button1 and button2];
     emit open_door;
end

end module
```

Figure 4 Timing of occurrence of external events

The **await[button1 and button2]** statement causes the program to wait until an instant where both **button1** and **button2** signals are present. The presence of just one of these signal has no effect. The corresponding signal is lost and no signal is emitted by the program.

3.3 Requirements for Supporting Esterel

The following issues have to be addressed for using Esterel as suggested by the examples in the previous section:

- providing support for the execution of the Esterel program within an asynchronous computing system and providing an interface so that it may interact with it.
- establishing a correspondence between the Esterel notion of time and real time. That is, establish a relation between events in the distributed system and instants in the synchronous execution model. The term *anchoring* has been used for this in the literature [5].
- ensuring that the integration of the Esterel program within the asynchronous environment is done in such a way that the assumptions, derived from the synchrony hypothesis, are approximated in a satisfactory way in the distributed system.

The integrated system may be thought of as a virtual synchronous execution machine. The issue is then the correct execution of this machine. In 3.4 we discuss how to integrate an Esterel program in a larger distributed environment for constructing such a virtual synchronous machine. In 3.5 we discuss the requirements for executing distributed applications controlled by Esterel programs in such a way that the synchronous hypothesis is correctly approximated.

3.4 A Virtual Synchronous Machine

The Esterel compiler [11] translates programs into a module implemented in some host language (in our case C). This module implements an automaton representing the Esterel program. The interface of the automaton with its environment is realized by a set of functions, the most interesting of which are:

- *Input functions:* for each input signal declared in the Esterel program, the compiler generates a C function PROG_I_someSignal() where someSignal is the name of the associated signal. This function is called by the program using the automaton to indicate the presence of the associated signal at the current instant.
- *The automaton transition function:* A function named PROG() is called to cause the automaton to react. The signals present at this instant are the ones for which a call to their associated C input function preceded the call to PROG().
- *Output functions:* for each output signal declared in the Esterel program, a C function named PROG_O_someSignal(), where someSignal is the name of an ouput signal, should by supplied by the environment. These functions are called by the automaton transition function to indicate that the associated output signal was emitted as part of the reaction of the Esterel program to signals present at the current instant.

To integrate the synchronous module into a distributed system, it is necessary to provide some software that does the following: i) Gets notified of external events and maps them to the corresponding signals declared in the synchronous program. ii) Using the interface described above, triggers the automaton after setting the appropriate set of signals. iii) Collects the signals produced in an automaton reaction, maps them to external events and presents these events to the environment.

The mechanisms for getting notified and signalling external events may vary depending on the host environment.

3.5 Correctness of a Virtual Synchronous Machine

3.5.1 Properties of a Synchronous Machine

In order to implement a virtual synchronous machine, it is necessary to address the following questions:

- How are asynchronous events mapped to instants in the execution of the synchronous program?
- How are instants in the execution of the synchronous program mapped to real-time (e.g. when the automaton runs)?
- How much time is needed for executing an automaton transition?
- Is the synchronous hypothesis approximated in interactions with asynchronous objects in the distributed application to sustain the synchronous virtual machine abstraction?

In the next section we further discuss these questions in the context of the example applications presented in section 3.2. The discussion illustrates the particular requirements of virtual machines and the need for a flexible engineering infrastructure for meeting these requirements.

3.5.2 The Examples Revisited

The synchronization requirements of example 1 are to impose bounds derived from tolerances of the human visual perception: i) on the latency from a change in the user's position to the instant the corresponding images are displayed, and ii) on the jitter in the display of the left and right eye images. The latency requirement can be expressed in the model discussed in section 2 by the appropriate QoS constraints on the bindings between the interfaces of the participating object, imposing a bound on time taken for rendering, and by a bound on the reaction time of the reactive object so that the synchrony hypothesis is approximated in a satisfactory way. The jitter requirement can be expressed by the appropriate QoS constraints on the bindings to the display objects, and by requiring that the implementations of the reactive and display objects correctly approximate the synchrony hypothesis.

In a similar way the requirement of a fixed frame rate in example 2 imposes bounds on the reaction time of the reactive object. What is important in this example is that the automaton reacts fast enough to the deadline signal and that the emission of display signals have acceptable latency and jitter. The constraints on the reaction with respect to signals from the decompressor is more relaxed as the application can tolerate that some frames are not being decompressed in time.

The main requirement imposed on the reactive objects in the first two examples is that they react fast enough to the occurrence of some external events. In the programs discussed in example 3, the time it takes for the reactive object to react to events is not as critical. What is more important is to be able to accurately detect the time of occurrence of external events and to then map them to instants in the execution of the synchronous program. These examples also show the need to explicitly specify the real-time granularity characterizing simultaneous events. This should be used by the infrastructure to correctly approximate the notion of simultaneous in the synchronous program. If this granularity was left unspecified it would lead to programs with infrastructure-dependent behaviour. For instance, in the Quiz example the timing should be chosen so that it matches the way users perceive the occurrence of events such as pushing a button.

The precise way such requirements are expressed and are associated with the components of a distributed application is outside the scope of this paper; more on this issue can be found in [22]. The next section concentrates on an infrastructure that can be used to implement virtual synchronous machines that satisfy such requirements.

4. Implementing the Computational Language

4.1 Engineering Support for Reactive Objects

4.1.1 The Use of the Chorus Micro-Kernel

Our computational language requires careful engineering to provide the predictable levels of performance required by multimedia applications. It is now becoming clear that this level of support cannot be achieved using conventional operating systems such as Unix[15][21]. We are therefore investigating the use of micro-kernels, in particular the Chorus micro-kernel[6], to support multimedia.

Chorus is a useful starting point for the research because it is lightweight and exhibits a number of real-time features. However, in common with other micro-kernels, it fails to address some key requirements of multimedia. Firstly, Chorus is message-oriented whereas, as discussed above, continuous media types require stream-oriented communications. Secondly, Chorus offers no quality of service control over communications and only coarse grained priority-based control over thread scheduling. In order to overcome these deficiencies, we have introduced a number of new facilities to the Chorus micro-kernel. These facilities are presented in detail in the literature[9]. In this paper, we concentrate on the features required to support reactive objects.

4.1.2 Key Extensions to Chorus

The main extension to the Chorus micro-kernel is the concept of rtports. Rtports are an extension of standard Chorus ports and serve as end-points for stream bindings. The key difference be-

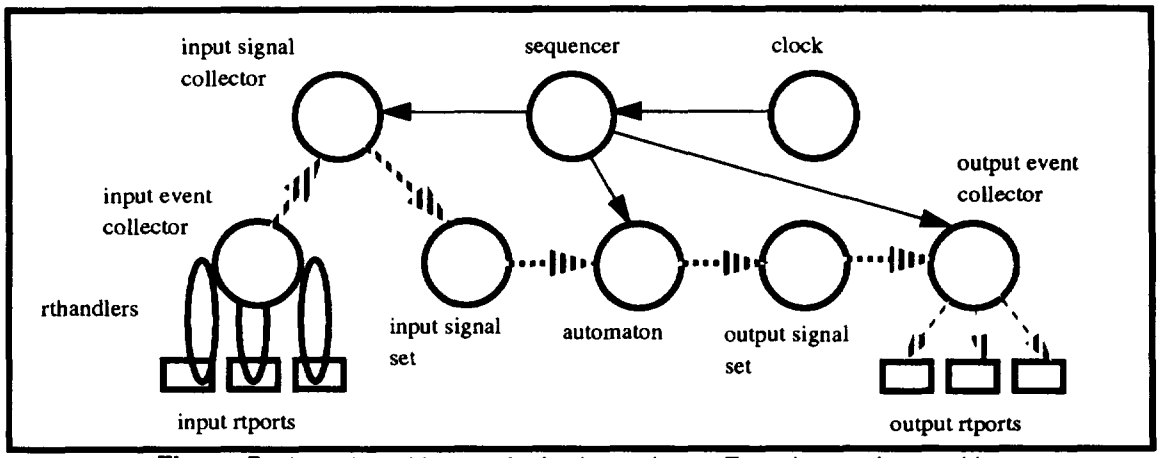

Figure 5 Generic architecture for implementing an Esterel execution machine

tween rtports and standard ports is that rtports have an associated quality of service which is used, for example, to manage buffers within the rtport. Rtports can have associated rthandlers, which are user defined procedures which manipulate real-time data coming from or going to an rtport. Rthandlers are upcalled from their associated rtport whenever data is required at a source rtport or has been delivered, by a binding, to a sink rtport.

Applications can use rthandlers either for the notification of events alone, or for both event notification and data transfer. We feel that this separation of notification and delivery is important for continuous media applications. It permits applications to choose whether they want to actively process continuous media data in user space, or merely to track the passage of continuous media generated and consumed in supervisor space. This latter case arises when the device under consideration is, for example, a kernel managed video device with associated frame buffer which is receiving data directly from the network card. Here, efficiency can be maximized as continuous media data need not cross protection domains.

When data arrives at an rtport, the thread dealing with the incoming data upcalls the rthandler. The same thread will then continue and is given a deadline derived from the quality of service of the associated binding. There is therefore co-ordination between the communications and scheduling subsystems to ensure end-to-end deadlines are met. Note that data is not copied to the application. Rather, the rthandler is given access to the buffer, containing the incoming data, directly.

In our scheme, there is an assumption that rthandlers will be fairly lightweight and can therefore complete processing within that deadline. Should the deadline expire, then the buffer containing the incoming data may be reclaimed by the system. A quality of service violation would also be notified to the application. Note that the rthandler thread can communicate with other threads in the processing of the incoming event. Communication is achieved by the use of semaphores. Signalled threads inherit the deadline of the rthandler.

The scheduling implementation exploits the concept of user-level lightweight threads to minimize the overhead due to context switches. It is now recognized that user level threads packages can be an order of magnitude faster than kernel level threads packages which are, in turn, an order of magnitude faster than conventional heavyweight processes. Threads are scheduled according to an earliest deadline first policy. The deadline (as mentioned above) is derived from the quality of service of the associated binding.

4.2 Implementing a Virtual Esterel Machine

In order to address the varying requirements of Esterel execution machines that we discussed in section 3.3, we have designed an object-oriented framework for supporting the implementation of Esterel machines. The appropriate selection of components in this framework can be used to accommodate different execution machines. Furthermore, object-oriented techniques can be used for integrating new components in the framework or refining existing ones using class inheritance.

In contrast to the previous sections, the terms component and object are used in this section to denote lightweight software entities such as C++ classes and instances.

4.2.1 A Generic Architecture

The Esterel machine is implemented as a single Chorus actor that communicates with the environment using a set of rtports. One sink rtport is used for each input signal and one source rtport is used for each output signal declared in the Esterel program.

Figure 5 shows the components used in the generic implementation of an Esterel machine. Dashed arrows are used to represent dataflow between components whereas continuous ones represent control flow.

External events are notified, asynchronously to the execution of the machine, through input rtports. The arrival of a message on an rtport triggers the execution of its associated rthandler. Each rthardler is in turn associated with an object of class Signal that maintains information about external events and their associated Esterel signals. The set of Signal objects is stored in the Input event collector object. The Signal class is important for scheduling and buffer management and is discussed in detail in section 4.2.2.

The main execution loop of the machine is encapsulated in the sequencer. The sequencer is activated by the clock object, then, it activates in turn the other machine components. First it activates the input signal collector. This object interacts with the input event collector and, according to the encapsulated signal collection policy, it constructs at set of input signals. This set is represented by the "input signal set" and is used as input to the automaton. The automaton object is activated next. This object encapsulates the automaton module produced by the Esterel compiler plus the code needed to feed the signals in the input set to the automaton module, trigger its reaction and construct an output signal set. After the automaton, the sequencer activates the output event collector which distributes the external events that correspond to the Esterel output signals through the output rtports. Finally, the sequencer activates the input signal set to release the resources associated with the signals that where used as input to the current automaton transition

(more on this in the next section). Having completed the machine execution cycle, the sequencer waits for the next clock tick before continuing with the next cycle.

4.2.2 Scheduling and Buffer Management

The real-time scheduling and buffer management integrated into rtports have been incorporated in our framework in the design of a signal class hierarchy. The classes in this hierarchy have been designed in such a way so that:

- the deadlines of rthandlers associated to selected rtports are inherited by the execution machine.
- the buffers associated with rtports are released as soon as they are not needed by the Esterel execution machine.
- these features are encapsulated in the design of the signal classes in such a way that they can be used through data abstractions that hide the details of their implementation.

A signal object is associated with each input rtport. When the handler associated with the rtport is invoked, it calls the method notify defined for all objects of the class hierarchy. There is a set of classes in the signal class hierarchy. Depending on the class of the signal object, it is possible to obtain different behaviours with respect to scheduling and buffer management.

Figure 6 (a) shows the signal classes and their inheritance relationships (if need be, more classes may be added in the future).

The Signal class is used when the Esterel signal that is associated to the rtport is a pure signal and the deadline of the rthandler does not affect the automaton (This can be determined by the analysis of the requirements of the execution machine as discussed in section 3.5). When the rthandler calls the notify method of a signal object of this class, the information needed by the framework, such as the timestamp of the event, is saved in the object's instance variables; the rthandler then returns and the associated buffer is released. The deadline associated with the rthandler then has no effect on the execution of the Esterel machine.

The ValuedSignal class is used when the Esterel signal associated to the rtport is a valued one. In this case, to avoid copying the value, the buffer should not be released. The implementation of the notify method for this class achieves this by blocking the rthandler that called it on a semaphore. When the buffer is no longer needed the semaphore is signalled, causing the rthandler to complete its execution and the buffer is then released. The semaphore may be signalled only indirectly by calling other methods of the ValuedSignal class, in particular **release** and **copyValue**. The former is called at the end of an execution cycle of the Esterel machine on signals that have been used as input to the current automaton transition and they are no longer needed. The latter is called when, during the execution of the Esterel machine, it is determined that the value is no longer needed. A **releaseValue** that simply signals the semaphore is also supported for increased flexibility.

The purpose of the QoSSignal class is to cause the execution machine to inherit the deadline of the rthandler of its associated rtport. The choice to associate this signal class with a particular rtport is driven by considerations discussed in section 3.5. The inheritance of deadlines takes place by having the rthandler wait on a semaphore that is signalled by the main machine thread at the end of the machine execution cycle. The approach taken for achieving this is very similar to the one used for the ValuedSignal class expect that the rthandler's deadline is inherited by the automaton.

Finally, the QoSValuedSignal class provides a combination of the functionality of the classes discussed above and is implemented in a much the same way.

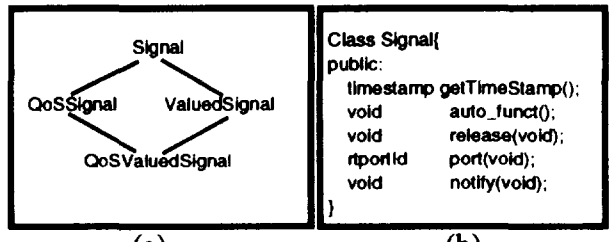

Figure 6 (a) Signal class inheritance dag
(b) Some methods of the signal class

4.2.3 Accommodating the Requirements of Different Machines

To illustrate the flexibility offered by the object-oriented design, we discuss some ways of tailoring the generic machine to meet the needs of the example applications.

Changing the input signal collection policy

The simplest approach for constructing the input signal set, is to include all the signals associated with events stored in the input event collector. However, in an application like the Quiz game controller or the door control system, an important issue is whether or not the signals associated to input events should be presented simultaneously to the Esterel automaton. A flexible way for controlling this is to have a collection policy that is based on time-stamps and which is parameterised by a threshold used to determine whether events should be considered simultaneous. For this, we define a subclass of the "standard" input signal collector. The interface of this class has an additional method for setting the threshold and accessing the threshold's value. The algorithm implemented in this class for constructing an input signal set selects all events that have occurred close enough in time, according to the threshold.

Changing the clock

The clock component determines how often the Esterel machine should go through its cycle. A **pulse** method is used to advance the clocks time. A tick method is called to signal a new instant for running a machine cycle. Several possibilities for running the machine can be encapsulated within subclasses of the clock class, some of them are briefly discussed below:

- to run periodically. This is supported by a cyclic thread that waits for given time, then calls **pulse**.
- on the occurrence of input events. The occurrence of a particular input event or a set of them is used to call **pulse**.
- on the invocation of an rthandler associated to an output rtport.

5. Related Work

Other work on using synchronous languages in an asynchronous environment is reported in [1][2][3][8]. [8] describes the use of Esterel in robotics, [1][2][3] describe a mixed synchronous/asynchronous approach based on Esterel for the implementation of automatic control systems. In contrast, the work presented in this paper focuses on providing a general way to support the implementation of synchronous programs in a distributed environment. Moreover, our work integrates synchronous programming with real-time QoS constraints, and takes advantage of the integrated scheduling and QoS support provided by our engineering infrastructure.

The use of object-based models for supporting the construction of distributed multimedia applications is reported in [12][23] and[20]. The approach proposed by Nicolaou in [20] has a lot in common with ours. The most important differences are the con-

cept of reactive object and the use of synchronous languages in our model for supporting real-time synchronization. In [20] this is achieved by using lower level synchronization mechanisms. In the approach proposed by the Object Systems Group at the University of Geneva synchronization is specified through temporal composition[13]. In this approach objects that encapsulate multimedia activities have to support temporal operations allowing them to be positioned, oriented and scaled relative to other objects in time, in way analogous to positioning graphics objects in space. A number of ways has been also suggested for maintaining synchronization. This framework provides an elegant way to describe and implement (although this work does not address QoS and OS support) synchronization of temporal activities by positioning them onto the time-line, it is difficult to see how it can be used for synchronizing activities with respect to events that can not be placed on the time line. For instance it is hard to see how the second example in section 3 could be expressed in their framework. In fact, in applications [14][23] where synchronization did not match the timeline model, synchronization between components has been implemented in an ad hoc way using conventional Unix, IPC mechanisms. We believe that our approach provides a more general way for addressing synchronization issues (some of which are further discussed in [19]) in multimedia systems structured as interacting objects. On the other hand, their synchronization framework, which could be implemented using the approach proposed in this paper, would be more convenient to use in some cases. Also, in their work, more support is provided for end user tools for constructing applications. It would be interesting to apply this aspect of their work to support the construction of applications based on the model presented in this paper.

6. Concluding Remarks

This paper has presented an approach to supporting multimedia in an open distributed processing environment. The approach relies on two key concepts, namely QoS-managed bindings (for predictable end-to-end communication) and reactive objects (for real-time control and synchronization). Using this approach, the real-time synchronization aspects can be programmed using a synchronous language, a notation which is highly suitable for expressing such real-time concerns.

We presented an underlying infrastructure which provides the necessary integration of communications, real-time process scheduling and buffer management to support our approach. The features provided by the infrastructure are incorporated into an object-oriented framework that can be tailored to match the needs of particular classes of applications.

We are currently completing the implementation of rtports, rthandlers and threads scheduling as reported in this paper. Details of this implementation are reported in [9].

Acknowledgments

Lancaster University was supported for this work by France Telecom through grant no. 93-5B-067. We would like to thank the Swiss FNRS for its support through grant no. 8220-037225. We would also like to thank Phil Adcock, Andrew Campbell and the reviewers for their suggestions for improving the presentation of this paper.

References

[1] C. Andre and L. Fancelli, "A Mixed Implementation of a Real-Time System," *Microprocessing and micro-programming*, vol. 30, North-Holland, 1990.

[2] C. Andre, J.P. Marmorat and J.P. Paris, *Execution Machines for Esterel*, Proceedings ECC '91, Grenoble,France.

[3] C. Andre and M.A. Peraldi, *Effective implementation of Esterel programs*, 1993, Euromicro Workshop on Real-Time Systems, Oulu, Finland.

[4] G. Berry and G. Gonthier, "The ESTEREL Synchronous Programming Language:Design, Semantics and Implementation," 842, INRIA, 1988.

[5] G. Berry and I. Cosserat, *The ESTEREL Synchronous Programming Language and its Mathematical Semantics*, LNCS, vol. 197, Springer Verlag, 1985.

[6] A. Bricker, M. Gien, M. Guillemont, M., Lipkis, J., Orr, D., and M. Rozier, "Architectural Issues in Microkernel-based Operating Systems: the CHORUS Experience", Computer Communications, Vol. 14, No 6, pp 347-357, July 1991.

[7] P. Caspi, D. Pilaud, N. Halbwachs and J. Plaice, "LUSTRE, a Declarative Language for Real-Time Programming," *Proceedings POPL'87*, ACM, Munich.

[8] E. Coste-Maniere, "Utilisation d'Esterel dans un contexte asynchrone: une application robotique," Research report 1139, INRIA, Dec. 1989, (in French).

[9] G. Coulson, G.S. Blair, P. Robin and D. Shepherd, "Extending the Chorus Micro-Kernel to Support Continuous Media Applications," *Proc. of the 4th International workshop on Network and Operating Systems Support for Digital Audio and Video*, Nov. 1993, Lancaster, UK.

[10] G. Coulson, G.S. Blair, F. Horn, L. Hazard, J.B. Stefani,"Supporting the Real-Time Requirements of Continuous Media in Open Distributed Processing," to appear in Computer Networks and ISDN Systems, (Special Issue on Open Distributed Processing), 1994.

[11] *Esterel V3 Language Reference Manual, Version 3.1 release 4*, CISI INGENIERIE, Agence Provence Est, Les Cardoulines B1, 06560 Valbonne, France.

[12] S. Gibbs, "Composite Multimedia and Active Objects," *Proc. OOPSLA'91 SIGPLAN Notices*, vol. 26, no. 11, ACM, 1991, pp. 97-112.

[13] S. Gibbs, L. Dami and D. Tsichritzis,"An Object-Oriented Framework for Multimedia Composition and Synchronization," proc. of Eurographics Multimedia Workshop, Stockholm 1991,

[14] S. Gibbs, V. de Mey, C. Breiteneder and M. Papathomas, "Video Widgets and Video Actors," *UIST'93 Conference proceedings*, ACM.

[15] Hanko, J.G., Keurner, E.M., Northcutt, J.D. and G.A. Wall, "Workstation Support for Time Critical Applications", *Proc. Second International Workshop on Network and Operating System Support for Digital Audio and Video*, Heidelberg, Springer Verlag, November 18-19 1991.

[16] D. Harel and A. Pnueli, "On the Development of Reactive Systems," *Logics and Models of Concurrent Systems* , K. Apt (Ed.), NATO ASI series, Springer, 1985.

[17] P. Le Guernic, A. Benveniste, P. Bournai and T. Gauthier, "SIGNAL: a Data Flow Oriented Language for Signal Processing," Report 246, IRISA, Irisa, Rennes, France, 1985.

[18] ODP, *Basic Reference Model for Open Distributed Processing Part 3: Prescriptive Model*, ANSI, 1430 Broadway, New York, NY 10018, USA, June 1993, ISO /IEC JTC1/SC21/WG7 Draft Recommendation X.903.

[19] M. Papathomas, C. Breiteneder, S. Gibbs and V. de Mey, "Synchronization in Virtual Worlds," *Multimedia and Virtual Worlds*, M.N Thalmann and D. Thalmann (Ed.), John Wiley, 1993.

[20] C. Nicolaou,"An architecture for real-time communication system," IEEE, *Journal on Selected Areas in Communications*, vol. 8, no.3, pp. 391-400, April 1990.

[21] Jason Nieh, J.N. Northcutt, J. G. Hanko, "SVR4 UNIX Scheduler Unacceptable For Multimedia Applications", *Proc. Fourth International Workshop on Network and Operating System Support for Digital Audio and Video*, Lancaster University, Lancaster LA1 4YR, UK, October 93.

[22] J.B. Stefani, " Computational Aspects of QoS in an Object-Based Distributed Architecture," *3rd International Workshop on Responsive Computer Systems*, Lincoln, NH, USA, September 1993.

[23] Vicki de Mey and Simon Gibbs, "A Multimedia Component Kit," *Proceedings ACM Multimedia '93*, Annaheim, CA, Aug 4-6, 1993.