COORDINATING COMPONENTS IN THE MULTIMEDIA SYSTEM SERVICES ARCHITECTURE

Theophilos A. Limniotes, George A. Papadopoulos[†]

Department of Computer Science
University of Cyprus
75 Kallipoleos St, Nicosia, POB 20537, CY-1678, CYPRUS
Tel: +357-2-892242, Fax: +357-2-339062
{theo,george}@cs.ucy.ac.cy

ABSTRACT

The purpose of this work is to examine and exploit the potential of the coordination paradigm to act as the main communication and synchronization mechanism between components forming a distributed multimedia environment and exhibiting real-time properties. Towards this purpose, we have developed a mechanism for coordinating the distributed execution of components, as these are defined by the Multimedia System Services Architecture (MSSA). Our coordination environment uses the control-driven approach to coordination, namely the model IWIM and the associated language Manifold. In the process we show how Manifold can be used to realize object communication and synchronization of MSSA components and we present a methodology of combining a software architecture such as MSSA with a coordination language such as Manifold. We illustrate our approach by means of a suitable example.

Keywords: *Coordination Paradigm; Distributed Multimedia Systems; Component-Based Systems; Real-Time Systems.*

1. INTRODUCTION

One of the most important developments in contemporary Software Engineering for Distributed Systems is that of component-based systems. Towards that end, we have seen a proliferation of models supporting the development of component-based software, such as middleware platforms, software architectures, coordination models and languages, etc.

The purpose of this work is to explore and exploit the potential of the coordination paradigm to act as the communication and synchronization mechanism between components forming a distributed multimedia environment and exhibiting real-time properties. We are particularly interested in two specific environments: the Multimedia System Services Architecture (Hewlett Packard, 1994), a software architecture framework for Distributed Multimedia environments proposed by some major companies in the field; and the coordination language Manifold (Arbab, 1996; Arbab *et al.*, 1998) which belongs to the control- or event-driven category of coordination models and associated languages (Papadopoulos and Arbab, 1998).

[†] Please refer to this author for all correspondence regarding this paper.

More to the point, we have developed a framework for executing components developed using the MSSA paradigm where object communication and synchronization is realized by means of a coordination infrastructure. The advantages of this approach is that we can develop reusable coordination patterns for distributed multimedia applications (Blair and Stefani, 1998), but also we make easier the distributed execution of components developed in the MSSA framework. Furthermore, and in a more general setting, we illustrate how software architectures (at the modeling level) can be combined with coordination languages (at the implementation level) to form a coherent methodology for developing component-based systems.

The rest of the paper is organized as follows. The next two sections provide a brief introduction to MSSA and Manifold. The next section presents the general philosophy of combining the software architecture MSSA with the coordination language Manifold. An example illustrating the relevant ideas is presented in the following section. The paper ends with some conclusions and reference to future work.

2. THE MULTIMEDIA SYSTEM SERVICES ARCHITECTURE

The primary goal of Multimedia System Services Architecture (Hewlett Packard, 1994), developed by the combined efforts of HP, IBM and SunSoft, is to provide an infrastructure for building interactive multimedia applications, dealing with synchronized and time-based media, and consisting of components running in heterogeneous distributed environments. In that respect, MSSA is a *software architecture* that specifies a methodology for building distributed multimedia frameworks. In particular, MSSA addresses issues such as the provision of abstract interfaces for media objects, grouping of objects into single composite ones, separation of media format abstractions from dataflow ones, etc. However, the actual communication and synchronization between MSSA media objects is a responsibility left to other formalisms that can be used in conjunction with the MSSA, typically a middleware platform for registering objects and providing inter-object communication mechanisms.

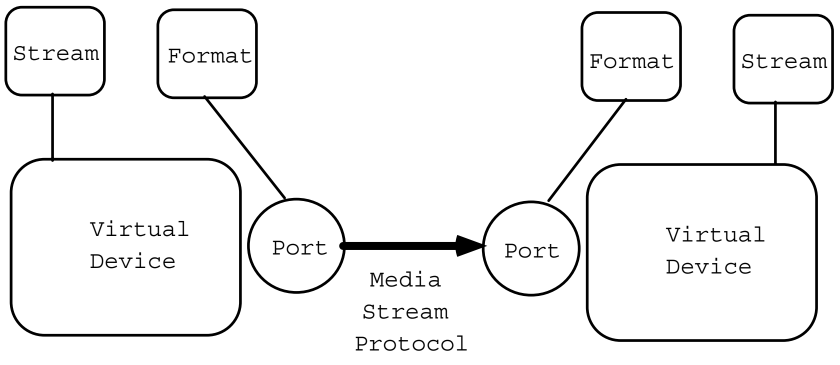

Fig. 1. Basic Component Functionality in MSSA

Figure 1 above shows a simplified illustration of MSSA, featuring two of its three main entities, namely virtual devices and virtual connections (the third, groups, is not relevant to the contents of this paper and is omitted for reasons of brevity). A *virtual device* is an abstraction over a physical device (e.g., CD player, microphone, etc.). Virtual devices offer a *stream interface* for communication with the environment. They also feature *format interfaces* providing an abstract representation of the details of media formatting. Finally, virtual devices feature one or more *ports* as input or output mechanisms.

MSSA adheres to an object-oriented approach. Thus, there exists an `MSSObject` class that produces a `VirtualResource`, `Format` or `Stream`. A client in order to access an `MSSObject` has to declare and initialise the MSSA client-side library. Then it requests a reference to a *factory* that would be able to satisfy a constraint list, from the Registration and Retrieval Service (RR). The reply is used with a constraint list to request an object from the factory. The factory instantiates the object and returns a reference of the newly created object. Objects can register with the factory their interest to be informed of any evolution in the system's architecture, such as failing of streams, non-adherence to QoS requirements, etc.; this is achieved by means of registering to and monitoring of *events*.

MSSA objects are associated with *characteristics* which define their behaviour and are specified by *capabilities*, which are key/value pairs. Furthermore, *constraints* are used to select objects that satisfy certain characteristics with a key/value/operator triple. These can be used in searching, creating and setting the requirements of an object. The *state* of an object reflects the constraints enforced on its capabilities. An *event* is a message between objects defined (as a key/value pair) by the sender, while the receivers must register for it. An event handler generates, registers, and processes the events. *Exceptions* are generated by an object or the distributed object infrastructure in case of encountering errors. There exists also a *narrowing* function that returns a pointer to the class of an object reference. Finally, an object can determine its class inheritance with a class *relationship* function.

Although a number of other formalisms have appeared lately addressing issues related to developing distributed multimedia information systems, MSSA remains one of the few which is both platform and language independent. Note however that MSSA does not concern itself with providing a methodology for coordinating the concurrent and distributed activities of media objects, nor the establishment of reusable collaboration patterns between such components (Blair and Stefani, 1998). It also relies on the underlying infrastructure for the enforcement of any real-time constraints and the satisfaction of any Quality-of-Service requirements. It is in these areas that we propose to use the coordination paradigm for.

3. THE COORDINATION LANGUAGE MANIFOLD

Manifold (Arbab, 1996; Arbab *et al.*, 1998) is a control-driven coordination language which is a realisation of the so-called Ideal Worker Ideal Manager (IWIM) coordination model (Arbab, 1996). In Manifold there exist two different types of processes: *managers* (or *coordinators*) and *workers*. A manager is responsible for setting up and taking care of the communication needs of the worker processes it controls. A worker is completely unaware of who (if anyone) needs the results it computes or from where it itself receives the data to process. Manifold possesses the following characteristics:

- *Processes*. A process is a *black box* with well defined *ports* of connection through which it exchanges *units* of information with the rest of the world. A process can be either a manager (coordinator) process or a worker. A manager process is responsible for setting up and managing the computation performed by a group of workers. Note that worker processes can themselves be managers of subgroups of other processes. The bottom line in this hierarchy is *atomic* processes which may in fact be written in any programming language.
- *Ports*. These are named openings in the boundary walls of a process through which units of information are exchanged using standard I/O type primitives analogous to read and write. Without loss of generality, we assume that each port is used for the exchange of information in only one direction: either into (*input* port) or out of (*output* port) a process. We use the notation `p.i` to refer to the port `i` of a process instance `p`.
- *Streams*. These are the means by which interconnections between the ports of processes are realized. A stream connects a (port of a) producer (process) to a (port of a) consumer (process). We write `p.o -> q.i` to denote a stream connecting the port `o` of a producer process `p` to the port `i` of a consumer process `q`.
- *Events*. Independent of streams, there is also an event mechanism for information exchange. Events are broadcast by their sources in the environment, yielding *event occurrences*. In principle, any process in the environment can pick up a broadcast event; in practice, only a subset of the potential receivers is interested in an event occurrence. We say that these processes are *tuned in* to the sources of the events they receive. We write `e.p` to refer to the event `e` raised by a source `p`.

Activity in a Manifold configuration is *event driven*. A coordinator process waits to observe an occurrence of some specific event (usually raised by a worker process it coordinates), which triggers it to enter a certain *state* and perform some actions. These actions typically consist of setting up or breaking off connections of ports and streams. It then remains in that state until it observes the occurrence of some other event, which causes the *preemption* of the current state in favour of a new one corresponding to that event. Once an event has been raised, its source generally continues with its activities, while the event occurrence propagates through the environment independently and is observed (if at all) by the other processes according to each observer's own sense of priorities.

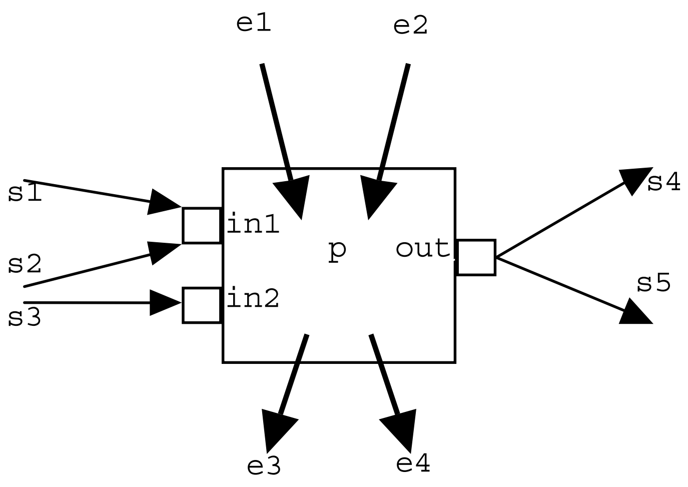

Fig. 2. An Illustration of Manifold's Coordination Structures

Figure 2 above shows diagrammatically the infrastructure of a Manifold process. The process `p` has two input ports (`in1`, `in2`) and an output one (`out`). Two input streams (`s1`, `s2`) are connected to `in1` and another one (`s3`) to `in2` delivering input data to `p`. Furthermore, `p` itself produces data, which via the `out` port are replicated to all outgoing streams (`s4`, `s5`). Finally, `p` observes the occurrence of the events `e1` and `e2` while it can itself raise the events `e3` and `e4`. Note that `p` need not know anything else about the environment within which it functions (i.e. who is sending it data, to whom it itself sends data, etc.).

The following is a Manifold program computing the Fibonacci series.

```
manifold PrintUnits() import.
manifold variable(port in) import.
manifold sum(event)
  port in x.
  port in y.
  import.
event overflow.

auto process v0 is variable(0).
auto process v1 is variable(1).
auto process print is PrintUnits.
auto process sigma is sum(overflow).

manifold Main()
{
 begin:(v0->sigma.x, v1->sigma.y,v1->v0, sigma->v1,sigma->print).
 overflow.sigma:halt.
}
```

The above code defines `sigma` as an instance of some predefined process `sum` with two input ports (`x`,`y`) and a default output one. The main part of the program sets up the network where the initial values (`0`,`1`) are fed into the network by means of two "variables" (`v0`,`v1`). The continuous generation of the series is realised by feeding the output of `sigma` back to itself via `v0` and `v1`. Note that in Manifold there are no variables (or constants for that matter) as such. A Manifold variable is a rather simple process that forwards whatever input it receives via its input port to all streams connected to its output port. A variable "assignment" is realised by feeding the contents of an output port into its input. Note also that computation will end when the event `overflow` is raised by `sigma`. `Main` will then get preempted from its `begin` state and make a transition to the `overflow` state and subsequently terminate by executing `halt`. Preemption of `Main` from its `begin` state causes the breaking of the stream connections; the processes involved in the network will then detect the breaking of their incoming streams and will also terminate.

More information on Manifold can be found in Arbab (1996), Arbab *et al* (1998), Papadopoulos (1998); the language has been implemented on top of PVM and has been successfully ported to a number of platforms including Sun, Silicon Graphics, Linux, and IBM AIX, SP1 and SP2.

A natural way to enhance the model with real-time capabilities is by extending its event manager. More to the point, we have enhanced the event manager with the ability to express real-time constraints associated with the raising of events but also reacting in bound time to observing them. With events that can be raised and detected respecting timing constraints, we essentially have a real-time coordination framework, since we can now guarantee that changes in the configuration of some system's infrastructure will be done in bounded time. A number of predicates that we have introduced in Manifold are useful in coordinating the raising of events when real-time constraints must be observed. Such predicates are `AP_Cause` which causes the raising of an event based on the time point of another event and `AP_Defer` which restricts the raising of an event based on the period of the time points of other two events. The use of these new primitives in coordinating real-time applications and safeguarding QoS constraints is discussed in Limniotes and Papadopoulos (2000).

Unlike the MSSA, which supports rather simple stream-based inter-object interactions, Manifold can handle stream and multi-stream connections of arbitrary complexity. Moreover streams in Manifold are just as good, for the transfer of discrete or continuous data for QoS agreement between objects or invocation and termination of operations. Explicit binding, which is a must in multimedia support in order to achieve QoS management, can be created between operational, stream and signal interfaces, for negotiation and agreement. However the stream bindings can only be passive with respect to initiating interactions. Note that the QoS provided by an object still depends on the QoS of the objects from which it inherits. The Manifold platform finally, has the additional advantage of the event management, which acts as an immediate reactive system, and provides the means to build a real-time response system.

4. USING MANIFOLD TO COORDINATE MSSA OBJECTS

In this section we discuss the general principles and philosophy in using the coordination paradigm, as this is expressed in the language Manifold, to synchronize the distributed execution of MSSA components. In the Manifold world we have two types of processes: *coordinator* programs written in Manifold itself and *atomic* processes performing computational work. The latter are viewed by the system as black boxes, communicating with their environment by means of well-defined interfaces, whose internals are completely hidden and play no role in the apparatus. Furthermore, some of the properties of this coordination model, such as ports, streams and events, are directly related to similar features of the MSSA paradigm.

Thus, we are going to view the MSSA objects as Manifold atomic processes, whose internal structure (as dictated by the MSS software architecture) is immaterial to our coordination framework. In particular, we are going to use the interface inheritance proposed by MSSA with the Manifold platform. For every type of an instance in the Manifold code, there should be a Manifold to activate and run it (something which makes the Factory Services of MSSA redundant). In the implementation every `VirtualDevice` type should correspond to every `PhysicalDevice` registered in the network and should be instantiated in the Manifold code. Each type may have different security, bandwidth, delay bound, resolution, transport mechanism and location.

The overall MSSA structure is not abandoned but some elements are replaced by corresponding elements in the Manifold model. In particular, the event, port and stream management of MSSA can be directly mapped to the event, port and stream management of Manifold. These mappings are done at the coordination level. The rest of the mapping is done at a lower level. In particular, the following `MSSObject` characteristics have the associated corresponding attributes in a Manifold atomic process: Capabilities associate machine and encoding, constraints associate location and machine. Also, exceptions are still generated by the object. The narrowing function remains intact into a `RegistrationRetrieval` Manifold atomic and so does the class relationship function.

For example, if we are going to have a particular `VirtualDevice` class (inherited from a `VirtualResource`) registered in the system, it should contain all the port management and event management functions that a Manifold atomic process should have. If that `VirtualDevice` is a `MicrophoneDevice` this should have its own in/out ports and events (such as one which is triggered if the signals sent per second exceed a certain maximum) defined in its C++ (say) code. The format of transferring the data is arranged from MIDI to tuples. The opposite transformation is done with the object that manages the speaker. QoS factors like *timeliness* could be important for allocating the process to a task that runs on a particular machine. Similarly the *throughput* of data required (sound signals per second) could weight (through event voting) to the decision for the specific `MicrophoneDevice` class that will run on a specific site. *Reliability* of the interactions is considered as secured throughout the system. Moreover, negotiations for QoS management can easily be carried out on the Manifold platform, with the coordination control transfer to a preliminary state where discrete information can be exchanged between two objects before being transferred to a final interaction state. The stream connection management relieves the MSSA from the need of a `VirtualConnection` class, although stream positions cannot be determined as with the MSSA. The Registration and Retrieval Service which allows clients to locate and retrieve a service can be preserved as is with the MSSA structure that keeps a name-to-object binding list which is then associated to a key list. These methods can be called by a `RegistrationRetrieval` atomic process in Manifold and every chosen device can invoke its own events that would cause its activation.

The methods for selecting the proper type by an RR Service of a virtual device are:

- `ConstraintList(constraints)` which defines a constraint builder and adds a constraint list for the type of object that is going to be selected.
- `DeviceType=VirtualDeviceNarrow`() which defines an object reference and gets the constraint list in order to narrow to the proper `DeviceType`. Note that it merely provides the type of an instance and not the instance itself.
- `VirtualDeviceRaise()` which raises an event for a particular `DeviceType`. This will cause preemption to a state (of a manifold) that contains the activation of a `DeviceCoordinator()` which contains the execution of this `DeviceType`.

All of the above classes in their implementation can either behave as atomic processes or be called by the `RegistrationRetrieval` atomic process.

5. A SIMPLE EXAMPLE

This paradigm deals with a remote audio and video capture with synchronized local play as might be used for half of a LAN based video conferencing system. The weaving is done at the remote machine by a process that sends its information at an unweaving device to the local machine which in turn supplies the proper outputs to a monitor and a speaker. For an overall view see figure 3.

5.1. The Manifold Atomic Processes

The atomic processes correspond to the `DeviceTypes` of our model. The implementation of the paradigm requires two device atomics for capturing the video and sound information, namely `CameraDevice` and `MicrophoneDevice` respectively, two atomics for weaving and unweaving this information, namely `WeaverDevice` and `UnweaverDevice`, and two atomic devices for producing this information at the output devices namely `XwindowDevice` and `SpeakerDevice`.

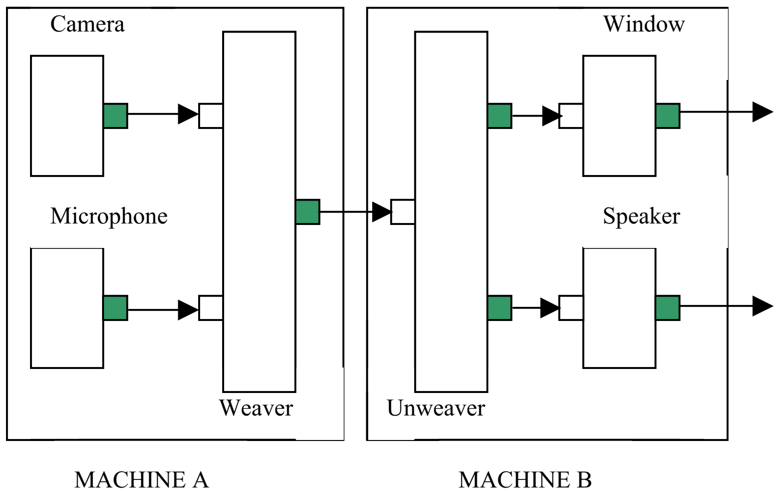

Fig 3. Audio/Video Remote Capture Example

5.2. The Manifold Coordinator Processes

These play the role of the `DeviceCoordinator` in the MMSA prototype, i.e. they activate the device drivers and set up the pattern of the pipelines for the transition of data from source to sink. So the atomics above are coordinated by the following processes:

`Weaver()` which takes care of processing in parallel the outputs from `CameraDevice` and `MicrophoneDevice` instances to the `WeaverDevice` instance. Simultaneously it passes the woven information to the output of the `WeaverDevice` output which in turn supplies the `Weaver`'s output. This coordination is processed in the `start_weaver` state invoked by the `cause1` instance. This is an instance of `AP_Cause` that raises the `start_weaver` event 5 seconds after the start of the application. The weaving is ceased with a preemption to the `finish_weaver` state, invoked by the `cause2` instance. This raises the `finish_weaver` event 30 (say) seconds after the `eventStart` of the application.

```
manifold Weaver()
{
begin:(activate(cause1,cause2,camera,microphone,weaver),
       cause1,WAIT).
start_weaver:(cause2,camera->weaver.x,
             microphone->weaver.y,weaver->output,WAIT).
finish_weaver:("weaver done"->stdout,WAIT).
}
```

In the code above `camera` is the instance of `CameraDevice`, `microphone` is the instance of `MicrophoneDevice`, and `weaver` the instance for `WeaverDevice`. `x` and `y` are the receiving ports for `weaver`.

`Unweaver()` processes the woven information from `Weaver()` to the `UnweaverDevice` which simultaneously produces the unwoven results from the previous input, and passes them to instances of `XwindowDevice` and `SpeakerDevice`.

5.3. The Main Coordinator

The instances of coordinators `Weaver` and `Unweaver` are activated and run in the `core` state of the `Main` procedure. In this pipeline `Weaver` is the producer and `Unweaver` the consumer. These two are said to be the building blocks of the higher coordination level.

```
manifold Main()
{
 process w is Weaver.
 process u is Unweaver.
 begin: AP_PutEventTimeAssociation_W(eventStart);
        AP_PutEventTimeAssociation(start_weaver);
        AP_PutEventTimeAssociation(finish_weaver);
        AP_PutEventTimeAssociation(start_unweaver);
        AP_PutEventTimeAssociation(finish_unweaver);
        (post(core),WAIT).
 core: activate(w,u);
       (w->u->stdout).
}
```

In the code above the two coordinators instances `w` and `u` form a pipeline. The `AP_PutEventTimeAssociation_W(eventStart)` function creates and initialises a record for the starting event of the paradigm. The rest of `AP_PutEventTimeAssociation` simply create empty records for the rest of the events.

5.4. Distribution of Tasks

The computation manifolds (atomics) that are to be run as separate tasks (usually on separate machines) have to be declared as `elsewhere` in the main object file.

```
manifold CameraDevice elsewhere.
manifold MicrophoneDevice elsewhere.
manifold XwindowDevice elsewhere.
manifold SpeakerDevice elsewhere.
```

So the two source devices are exported from the source Manifold file `source.m`:

```
// pragma include "tm.ato.h"
export manifold CameraDevice() atomic {internal.}.
export manifold MicrophoneDevice() atomic {internal.}.
```

Likewise, the two sink devices are exported from the sink Manifold file `sink.m`.

The rest of the computation manifolds can be chosen to run on the local host by declaring them as internal:

```
manifold AP_PutEventTimeAssociation(event) atomic {internal.}.
manifold AP_PutEventTimeAssociation_W(event)
         atomic {internal.}.
manifold AP_Cause(event,event,port in,port in)
         atomic {internal.}.
manifold AP_CTime() atomic {internal.}.
manifold WeaverDevice()
  port in x.
  port in y.
  atomic {internal.}.
manifold UnweaverDevice()
```

```
    port out x.
    port out y.
    atomic {internal.}.
```

The next step in task allocation is to include each generated object file to a task, in a task mapfile. This will help produce the link information for the tasks accordingly. The merits of running tasks in a heterogeneous environment are that: (i) some of the tasks may include atomics written in different languages for different hardware/software architectures; (ii) some given tasks may have different versions, each suitable for a different hardware/software architecture.

The program `mlink` that composes the various tasks in a Manifold application, produces nothing but text files, suitable for all heterogeneous environments. These can be used to link their corresponding versions of the tasks in the Manifold application. The restriction with `mlink` is that there should be one compilation on an appointed host, which will then hold the *same* version of Manifold object files and Manifold application libraries. These will provide the required link information to `mlink` in order to make up the application tasks. Regarding the library and executables, it follows that these do not have to be recompiled on the remote hosts on which they are intended to run, as `mlink` requires only the native binary format. In case that the appointed hosts cannot make available a meaningful to `mlink` binary version, the `decoy` utility decompiles to a C source that provides the required link information, and then recompiles to create a perceivable object file. So, it is important to take into consideration the following conditions: (i) If the executable code for the new instance is not contained in an *existing task instance*; (ii) If the number of instances exceed a maximum weightload for an *existing task instance.* If the answer is no to either of the above conditions, a new task instance has to be created in order to house the new process instance at the designated site.

6. CONCLUSIONS AND FURTHER WORK

In this paper we have addressed a rather general issue, namely the use of a coordination paradigm as the gluing mechanism and communication medium for the synchronized execution of distributed components, as the latter are defined by a software architecture framework. We have done the above by concentrating on the specific case of distributed multimedia components, as these are defined by the MSSA software architecture (Hewlett Packard, 1994), and we have developed for them an execution framework based on a real-time extension of the coordination model IWIM and its associated language Manifold.

The IWIM model enforces isolation of computational aspects from the matters of connectionism, coordination and reusability (after recompilation) while our Manifold real-time primitives were able to improve further the real-time behaviour of the system. Thus, the same coordination code can be applicable to any other similar behavioural pattern of media modules. With respect to the continuous data transfer issue, Manifold's streams guarantee, by virtue of the model, a flow without loss, error or duplication, and with causal order preserved. Moreover the stream and event services of Manifold offer the means to manage discrete message passing and state invocation for QoS support with minimum programming effort (Blair and Stefani, 1998).

So far we have built language constructs to define temporal interdependencies in a multimedia presentation. The next step is to synchronise distributed multimedia systems with blocking times for every task (Mourlas, 1999). More to the point, every multimedia task should have a determined execution time which is made up of the time that a task requires to retrieve information from certain resources plus the computation period for processing. To this, a blocking time must be added for the possible lower priority tasks that are already using the same resources, and all the higher priority resources that are waiting to use these resources (Halbwachs, 1993).

An important factor in the above scheme is the determination of priority for each task. The calculation of the blocking time depends on the priority of the task. In Manifold priorities can be allocated to states that can contain many tasks in a pipeline. So it is only possible to give priority to a set of tasks that are included in the same pipeline.

ACKNOWLEDGMENTS

This work has been partially supported by the INCO-DC KIT (Keep-in-Touch) program 962144 "Developing Software Engineering Environments for Distributed Information Systems" financed by the Commission of the European Union.

REFERENCES

F. ARBAB (1996), "The IWIM Model for Coordination of Concurrent Activities", *1st International Conference on Coordination Models, Languages and Applications (Coordination'96)*, Cesena, Italy, 15-17 April, 1996, LNCS 1061, Springer Verlag, pp. 34-56.

F. ARBAB, C. L. BLOM, F. J. BURGER and C. T. H. EVERAARS (1998), "Reusable Coordinator Modules for Massively Concurrent Applications", *Software: Practice and Experience*, Vol. 28 (7), 1998, pp. 703-735.

G. BLAIR and J-B. STEFANI (1998), *Open Distributed Processing and Multimedia*, Addison-Wesley, 1998.

N. HALBWACHS (1993), *Synchronous Programming of Reactive Systems*, Kluwer Academic Publishers, 1993.

HEWLETT PACKARD Company (1994), International Business Machines Corporation, SunSoft, Inc., *Multimedia System Services Version 1.0*, 1994.

T. A. LEMNIOTES and G. A. PAPADOPOULOS (2000), "Real-Time Coordination in Distributed Multimedia Systems", *14th International Parallel and Distributed Processing Symposium (IPDPS 2000), 8th International Workshop on Parallel and Distributed Real-Time Systems (WPDRTS 2000)*, Cancun, Mexico, 1-2 May, 2000, LNCS 1800, Springer Verlag, pp. 685-691.

C. MOURLAS (1999), "Multiprocessor Scheduling of Real-Time Tasks with Resource Requirements", *5th International Euro-Par Conference (Euro-Par'99)*, Toulouse, France, 31 Aug. – 3 Sept., 1999, LNCS 1685, Springer Verlag, pp. 497-504.

G. A. PAPADOPOULOS (1998), "Distributed and Parallel Systems Engineering in Manifold", *Parallel Computing*, Elsevier Science, special issue on Coordination, 1998, Vol. 24 (7), pp. 1107-1135.

G. A. PAPADOPOULOS and F. ARBAB (1998), "Coordination Models and Languages", *Advances in Computers*, Marvin V. Zelkowitz (ed.), Academic Press, Vol. 46, August, 1998, 329-400.