Event-Driven Coordination of Real-Time Components

Theophilos Limniotes, Costas Mourlas and George A. Papadopoulos

Department of Computer Science
University of Cyprus
75 Kallipoleos Street, POB 20537, CY-1678
Nicosia, CYPRUS

{theo,mourlas,george}@cs.ucy.ac.cy

Abstract

The coordination paradigm has been used extensively as a mechanism for software composition and integration. However, relatively little work has been done for the cases where the software components involved have real-time requirements. The paper presents an extension to a state-of-the-art control- or event-driven coordination language with real-time capabilities. It then illustrates the expressiveness of the proposed extensions by means of modeling a distributed multimedia application. Finally, it discusses how these extensions can be supported by the underlying architecture.

Keywords: Coordination Paradigm; Distributed Multimedia; CBSE; Real-Time Systems.

1. Introduction

The concept of coordinating a number of activities, possibly created independently from each other, such that they can run concurrently in a parallel and/or distributed fashion has received wide attention and a number of *coordination* models and associated languages ([7]) have been developed for many application areas such as high-performance computing or distributed systems. Nevertheless, most of the proposed coordination frameworks are suited for environments where the sub-components comprising an application are conventional ones in the sense that they do not adhere to any real-time constraints. Those few that are addressing this issue of real-time coordination either rely on the ability of the underlying architecture apparatus to provide real-time support ([12]) and/or are confined to using a specific real-time language ([9]). However, all the issues pertaining to conventional coordination frameworks are still valid in those exhibiting real-time properties. Even more, the use of a separate coordination formalism with real-time capabilities "gluing together" real-time (software and/or hardware) components helps in separating issues related to specifying the computational part of the components from those concerned specifically with their real-time behaviour ([13]). This approach has a number of advantages, ranging from the ability to design, implement and test real-time algorithms independent from the intended environment in which they will be used, to offering flexible dynamic re-configuration by hiding away real-time programming and analysis details, to allowing the re-use of common real-time synchronization patterns that may be applicable in a number of different scenaria.

In this paper we address the issue of real-time coordination within the context of the so-called control- or event-driven coordination languages ([7]) which we feel are particularly suited for this purpose, and specifically the language Manifold ([1, 8]). We show that it is quite natural to extend such a language with primitives enforcing real-time coordination and we apply the proposed model to the area of distributed multimedia systems.

The rest of this paper is organised as follows. The next section is a brief introduction to the coordination language Manifold; some emphasis is put on those features of the language that are particularly suited to express real-time behaviour, notably its event-driven state transition model. The following section uses the extended framework to model a particular case from the area of distributed multimedia systems. Then, we focus on "real" environments where a set of multimedia applications share a number of non-preemptable resources or access shared data (e.g. storage servers, live media sources, etc.) which are part of a high-speed local area network. We concentrate on the effect of this synchronization in the schedulability and the predictability of the set of parallel running continuous media applications and we propose a real-time scheduler for Manifold that makes at run-time the timing properties of each application predictable. The paper ends with some conclusions and reference to related work.

2. The Coordination Language Manifold

Manifold is a control- or event-driven coordination language, and is a realisation of the Ideal Worker Ideal Manager (IWIM) model ([1]). 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 group of worker processes it controls (non-exclusively). A worker on the other hand is completely unaware of who (if anyone) needs the results it computes or from where it itself receives the data to process. Manager processes are written in Manifod whereas worker processes may be written also in Manifold or in some computational language (typically C, Fortran). In this latest case, these worker processes are called *atomics*. In particular, 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.
- *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* or *channels*. These are the means by which interconnections between the ports of processes are realised. A stream connects a producer process to 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 channels, 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 though, usually only a subset of the potential receivers is interested in an event occurrence. 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 channels. 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.

3. Extending Manifold with a Real-Time Event Manager

The IWIM coordination model and its associated language Manifold have some inherent characteristics, which are particularly suited to the modelling of real-time (software or otherwise) systems. Probably the most important of these is the fact that the coordination formalism has no concern about the *nature* of the data being transmitted between input and output ports since this data plays no role at all in setting up coordination patterns. In particular, a stream connection between a pair of input-output ports, simply passes anything that flows within it from the output to the input port. Furthermore, the processes involved in some coordination or cooperation scenario are treated by the coordination formalism (and in return treat each other) as black boxes without any concern being raised as to their very nature or what exactly they do. Thus, for all practical purposes, some of those black boxes may well be devices (rather than software modules) and the information being sent or received by their output and input ports respectively may well be signals, or continuous data (as opposed to ordinary discrete data). Note also that the notion of stream connections as a communication metaphor, captures both the case of transmitting discrete signals (from some device) but also continuous signals (from, say, a media player). Thus, IWIM and Manifold are ideal starting points for developing a real-time coordination framework.

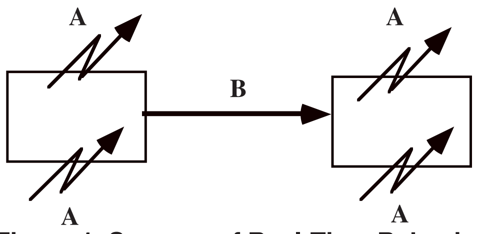

Figure 1. Sources of Real-Time Behaviour

Figure 1 illustrates where precisely real-time behaviour must be introduced and supported; in particular, two such areas are indicated. Since Manifold is event-driven, event raising and detection must now be done within bounded time intervals, as opposed to the asynchronous nature of the original model. Introducing such a temporal behaviour, allows the expression of temporal relationships between interacting components. This is the first area (A). Additionally, we must ensure that the transfer of timed media through the streams adheres to any QoS requirements. This is the second area (B) and is addressed in section 5.

In order to deal with the first kind of real-time behaviour we want our model to support, we enhance 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. Thus, while in the ordinary Manifold system the raising of some event `e` by a process `p` and its subsequent observation by some other process `q` are done completely asynchronously, in our extended framework timing constraints can be imposed regarding when `p` will raise `e` but also when `q` should react to observing it. Effectively, an event is not any more a pair `<e,p>`, but a triple `<e,p,t>` where `t` denotes the moment in time at which the event occurs. 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. Thus, our real-time Manifold system goes beyond ordinary coordination to providing temporal synchronization.

This extension couples conviniently with the rest of Manifold's event model which already supports the rest of the functionality usually found in event-based systems ([5]), such as anonymous broadcasting, filtering, priorities, etc.

Below we describe the most important primitives that our extended event manager supports. These primitives can be classified into two categories: the first category is responsible for expressing the notion of time while the second one expresses temporal relationships among events.

3.1 Recording Time

The extended event manager maintains a table of records of the form `<e,p,t>`, as was defined above. The use of anyone of the primitives whose role is to express temporal relationships, effectively updates the entries in the table of the events involved in the expressed relationships. In particular, the time `t` is modified, according to the temporal relationships expressed. A linked list is used to build an association between the events and the time points that triggers them. The `AP_Event` field stores the identification for the event. The `time_t` type stores the relevant delay of execution to another event's timepoint, and the lock field is useful for deferring the execution of the event for a period of time, when its execution is permitted by the timepoint period.

Type	`AP_Event`	`time_t`	`u_short`
Description	Event ID	Delay	Lock Period

A number of primitives exist for capturing the notion of time, either relative to world time, the occurrence of some event, etc. These primitives have been implemented as atomic (i.e. not Manifold) processes in C and Unix. In particular:

- **`AP_CurrTime(int timemode)`**

returns the current time according to the parameter **`timemode`**, the latter being world time or relative.

- **`AP_OccTime(AP_Event anevent, int timemode)`**

returns the time point (in world or relative mode) of an event. Time points represent single instances in time; two time points form a basic interval of time.

- **`AP_PutEventTimeAssociation(AP_Event anevent)`**

creates a record for every event that is introduced and inserts it in the events table mentioned above.

- **`AP_PutEventTimeAssociation_W(AP_Event anevent)`**

is a similar primitive which additionally marks the world time when a scenario starts, so that the rest of the events can relate their time points to it.

We stress again the fact that, contrary to what is happening in the ordinary Manifold system, the events are not raised explicitly by means of the `raise` primitive, but only implicitly via the time points of the occurrence of some other event. In that way, the system builds a sequence of events to be raised and observed according to the specified timing constraints and imposes the necessary temporal ordering required.

3.2 Expressing Temporal Relationships

There are two primitives for expressing temporal constraints among events raised and/or observed. The first is used to specify when an event must be triggered while the second is used to specify when the triggering of an event must be delayed for some time period.

- **`AP_Cause(AP_Event anevent, AP_Event another, AP_Port delay, AP_Port timemode)`**

enables the triggering of the event **`another`** in relation to the timepoint of **`anevent`**. The timepoint for this second event is found using the **`OccTime()`** function. Its difference to its current time is given by (**`OccTime()-CurrTime()`**). Another basic time unit is the **`delay`** which is entered as a parameter in the event management predicates **`AP_Cause`**, **`AP_Defer`** and **`AP_Synch`** (described below), and implies that a **`cause`**, a **`defer`**, or a **`synch`**ronisation predicate will take place only some time after the triggering of the event's timepoint. It represents seconds added to this timepoint and it is an optional parameter; likewise **`timemode`**, which allows for the occurrence of the timepoint and the current time to be calculated in either time mode (world or relative).

- **`AP_Defer(AP_Event eventa, AP_Event eventb, AP_Event eventc, AP_Port delay)`**

inhibits the triggering of the event **`eventc`** for the time interval specified by the events **`eventa`** and **`eventb`**. **`delay`** here represents how much time will elapse after the start of the interval during which inhibition will take place. During the duration of the inhibition the event's record remains locked, so that every attempt to update its timepoint (i.e. cause it) will cause the cancellation of the update. As before, the primitive **`AP_OccTime`** is used to retrieve the time points **`eventa`** and **`eventb`**, and the primitive **`AP_CurrTime`** to get the current time. Note that in the case that a **`cause`** and a **`defer`** coincide for the same event, **`defer`** prevails.

In addition to the above two primitives, two more auxiliary ones are used at a lower level:

- **`AP_UpdateEventTimeAssociation(AP_Event anevent, int delay, int timemode)`**

updates an event's record with a new timepoint, which is derived from the sum of the delay and the time point of the triggering event, and makes sure that two updates are not performed on the same record at the same time. Moreover, when the time comes for preemption to take place, a lookup process checks if the occurrence of the event is deferred or not and proceeds accordingly with raising of the event or the abortion of the update.

Finally, the primitive

- **`AP_Synch(AP_Event eventa, AP_Event eventb, AP_Event eventc, AP_Event eventd, AP_Port delay)`**

expresses synchronization of intervals that are executed simultaneously. Here an interval defined by **`eventa`** and **`eventb`** is related to an interval defined by **`eventc`** and **`eventd`**. The events **`eventc`** and **`eventd`** are caused according to the timepoints of **`eventa`** and **`eventb`**.

4. Coordination of Real-Time Components in a Multimedia Presentation

We show the applicability of our proposed model by modelling an interactive multimedia example with video, sound, and music ([10]). A video accompanied by some music is played at the beginning. Then, three successive slides appear with a question. For every slide, if the answer given by the user is correct the next slide appears; otherwise the part of the presentation that contains the correct answer is re-played before the next question is asked. There are two sound streams, one for English and another one for German. A graphical presentation of the example is given below.

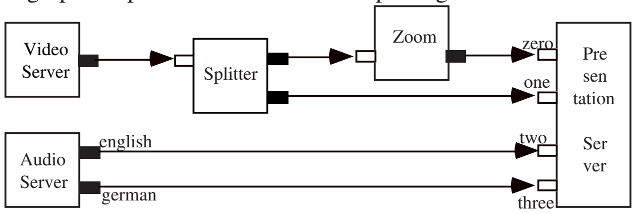

Figure 2. A Distributed Multimedia Application

For each medium, as appeared in the diagram above, there exists a separate manifold process. Each such manifold process is a, potentially reusable, “building block”. For instance, the following manifold coordinates the execution of atomics that take a video from the media object server and transfer it to a presentation server.

```
manifold tv1()
{
 begin:(activate(cause1,cause2,mosvideo,
        splitter,zoom),cause1,WAIT).
 start_tv1:(cause2,mosvideo ->
                         ( -> splitter),
           splitter.zoom ->zoom,
zoom-> (->ps.zero),ps.out1->stdout,WAIT).
 end_tv1:post(end).
 end:(activate(ts1),ts1).
}
```

In addition to the `begin` and `end` states which apply at the beginning and the end of the manifold’s execution respectively, two more states are invoked by the `AP_Cause` commands, namely `start_tv1` and `end_tv1`. At the `begin` state the instances of the atomics `cause1`, `cause2`, `mosvideo`, `splitter`, and `zoom` are activated. These activations introduce them as observable sources of events. This state is *synchronized* to preempt to `start_tv1` with the execution of `cause1`. The declaration of the instance `cause1`:

```
process cause1 is
AP_Cause(eventPS,start_tv1,3,CLOCK_P_REL)
```

indicates that the preemption to `start_tv1` should occur 3 seconds (relative time) after the raise of the presentation start event `eventPS`.

Within `start_tv1` the other three instances, `cause2`, `mosvideo`, and `splitter`, are executed in parallel. `cause2` synchronizes the preemption to `end_tv1` and its declaration

```
process cause2 is
AP_Cause(eventPS,end_tv1,13,CLOCK_P_REL)
```

indicates that the currently running state must execute the other two atomic instances within 13 seconds. So, the process for the media object `mosvideo` keeps sending its data to `splitter` until the state is preempted to `end_tv1`. The `mosvideo` coordinating instance supplies the video frames to the `splitter` manifold. The role of `splitter` here is to process the video frames in two ways. One with the intention to be magnified (by the `zoom` manifold) and the other at normal size directly to a presentation port. `zoom` is an instance of an atomic which takes care of the video magnification and supplies its output to another port of the presentation server. The presentation server instance `ps` filters out the input from the supplying instances, i.e. it arranges the audio language (English or German) and the video magnification selection. At the `end_tv1` state the presentation ceases and control is passed to the `end` state. Finally at the `end` state, the `tv1` manifold is activated and performs the first question slide manifold `ts1`. This prompts a question, which if answered correctly prompts in return the next question slide. A wrong answer leads to the replaying of the presentation that relates to the correct answer, before going on with the next question slide. The code for a slide manifold is given below.

```
manifold tslide1()
{
 begin:(activate(cause7),cause7,WAIT).
 start_tslide1:(activate(testslide),
             testslide,WAIT).
 tslide1_correct:
              "correct answer"->stdout;
        (activate(cause8),cause8,WAIT).
 tslide1_wrong: "wrong answer"->stdout;
        (activate(cause9),cause9,WAIT).
 end_tslide1:(post(end),WAIT).
 start_replay1: (activate(replay1,
        cause10),replay1,cause10,WAIT).
 end_replay1: (activate(cause11),
            cause11,WAIT).
 end:(activate(ts2),ts2).
}
```

The above example highlights two main issues in distributed multimedia systems. The first one is separation of the coordination from the computation components. As a result, either of the two categories of functionality become parametric to the whole application and can be substituted or re-used in other similar applications. The second one is realizing real-time synchronization between the system’s components.

These tasks are achieved with the atomic primitives and predicates (like `AP_PutEventTimeAssociation()`, `AP_CurrTime`, `AP_OccTime`, `AP_Cause` and `AP_Defer`), as well as the atomics for the transmission and the filtering of media information (like the `mosvideo` and `ps` as described above). The coordination of these is done by the media and the slide coordinating manifolds, namely `tv1`, `mosvideo`, `splitter`, `zoom` for the media, and `tslide1`, `tslide2` and `tslide3` for the slides. Each of the above atomics and coordinators, being manifolds, are reusable and the connectivity of streams remains reliable even if some components are substituted by others. In particular, we may want to change some computational components (say, the ones related to the slides or the media content). This can be achieved without modifying the coordination and synchronization part of the application. In fact, we can take `tv1` (say) as it is and reuse it in another similar scenario. Note also that we can keep the coordination (i.e. input-output relationships) intact and "play around" with different timing constraints, since the related primitives are also quite independent from the rest of the apparatus. As a final point, we note that all the involved components can (and, in fact, do) run in a distributed fashion.

5. Extending Manifold with a Real-Time Scheduler

In this section we deal with the second aspect of real-time behaviour, as was illustrated in figure 2 earlier on. In particular, our goal is the extension of the Manifold run-time environment based on a well-defined theory which will be able to supply the continuous media streams of every multimedia application with enough data to ensure that the playback processes do not starve (intra-media synchronization). We have developed a new theory for the Manifold execution environment that makes the timing properties of the system and the quality of the presentation of each multimedia application *predictable.* This means that one is able to determine analytically whether the timing requirements of the different continuous media will be met, and if not, which timing requirements will fail. The proposed real-time scheduler for Manifold is based on our *Set Based Synchronization Protocol* ([6]) where resources are assigned only when actually required so that the system never wastes an assignment that will not be used. The penalty paid for this, found also in all the on-demand approaches, is blocking. If the blocking time is non-deterministic then the whole system becomes unpredictable and difficult to analyze. In predictable multimedia environments, the blocking has to be deterministic and for this reason our approach imposes a specific structure on blocking to bound the blocking time. It is an extension of our previous work on distributed real-time and multimedia systems and comes from our experience in the interesting field of real-time programming ([4]) where similar problems for resource management and timing correctness have been extensively studied.

In our model, we assume that the QoS for continuous media objects is expressed with temporal and spatial resolutions. The temporal resolution can be expressed by the number of frames per second or sample rate and the spatial resolution can by expressed by data size, number of bits per pixel, compression scheme, etc. For example, in one simple digital video application, the user may choose 22 frames per second for its temporal resolution and a spatial resolution of 160 by 120 pixels wide with an 8-bit color resolution. It is assumed also that every program executes periodic reads of a number of frames from a remote media server into a local buffer first and then plays them due to the fact that continuous media require periodic service activities for transmission and presentation.

We consider a new *task oriented model* that addresses all the real-time requirements of the media streams. More precisely, we view every different multimedia application executed in a distributed environment as a *periodic task* that can require in each period the use of non-preemptable resources or access shared data. For example, one multimedia session can be modeled as a task which in every 50 ms needs to deliver 3 video frames from the *storage_ server*$_1$ and 6 audio frames from *storage_ server*$_2$. Since these storage servers are shared and exclusively used (i.e. guaranteed exclusive access), there is a possibility for one such task to block waiting for the use of these servers. The period of each multimedia task is determined by the desired quality of service (i.e. the temporal and spatial resolutions of the continuous media), the number of continuous media used in the application, the processor speed and the buffer size used by the node that executes the application. This means that high quality applications using many continuous media are represented by our model as tasks having short periods, i.e. high frequency tasks.

The proposed strategy is based on the rate monotonic algorithm ([4]) in the way the priorities are assigned to tasks. The assumptions and basic notation follow:

1. any continuous media application executed in a node of a distributed environment is represented by a multimedia *task* $_i$.
2. every multimedia task$_i$ is periodic with period T_i and has deadline D_i at the end of its period (i.e. D_i=T_i).
3. multimedia tasks are assigned fixed priorities inversely to their periods. Hence, task$_i$ with period T_i receives higher priority than task$_j$ with period T_j if $T_i < T_j$.
4. every periodic task$_i$ is allocated on a different node p_i of the distributed system and can require the use of non-preemptable resources or access shared data R_i.
5. every task asks for all of its global resources R_i only once in its period and subsequently can release these resources (one by one or all at once) after their use. Two operations are used for this reason, taking as an argument a set of resources:
   - *allocate(ResourceSet)* and
   - *release(ResourceSet)*

   When a task$_i$ issues the *allocate* command asking for its resources it then blocks (i.e. hangs) until all these resources have been allocated to task$_i$ by the resource manager. The duration of this time interval constitutes the blocking time B_i of the task.
6. every multimedia task$_i$ has known, deterministic worst-case execution time C_i. This is the total deterministic computation requirement of task$_i$ during each period, and $C_i = C^i_{cs} + C^i_{non\text{-}cs}$ where:
   $\mathbf{C^i_{cs}}$ is the total time that task$_i$ uses the resources and the network in each period for data retrieval, for example for the delivery of a set of frames from the storage servers to the node where task$_i$ resides,
   $\mathbf{C^i_{non\text{-}cs}}$ is the deterministic computation requirement of task$_i$ that task$_i$ needs to process the received data frames.

Due to the fact that every multimedia task is allocated on a different node of the distributed system, CPU scheduling is not the main problem, but since tasks are inter-dependent the main problem is task synchronization and resource allocation. Hence, blocking due to synchronization has to be deterministic in order to have nice analysis properties and a high degree of system predictability. Given a blocking duration *B* of a $task_i$ with period *T*, then the ratio *B/T* is a measure of schedulability loss due to blocking. In our approach, we try to minimize this ratio as much as possible. A formal definition of the *Set Based Synchronization Protocol* is given in [6] where the exact evaluation of blocking duration B of every $task_i$ is also described.

6. Conclusions

In this paper we have addressed the issue of real-time coordination in parallel and distributed systems. In particular, we have extended a control- or event-driven coordination language with a real-time event manager that allows expressing timing constraints in the raising, observing, and reacting to events.

Our model is similar to models such as DAMSEL ([10]) which also relies on the use of a real-time event manager. There are, however, a number of differences between that approach and ours in the way interfaces to components are defined (inlets and outlets instead of ports), the nature of stream connections (ours are more flexible and secure) and also on the very nature of events (in DAMSEL they can carry data which makes their broadcasting a rather expensive operation). On the other hand, our model defers from other ones in that it does not rely on the use of special real-time languages ([9]) or the underlying system architecture to support hard real-time activities ([12]). Especially regarding the last point, we note that implementing real-time languages based on the synchronous hypothesis (i.e. instantaneous reaction to events) is a non-trivial issue ([3]). However, we must also note that our model is mostly suitable to model cases of "soft" real-time systems where bounded time (but not instantaneous) reaction to events is sufficient.

We have also proposed a real-time resource allocation mechanism formally defined in [6] to extend the run-time system of Manifold that is analyzable and understandable at a high level. Given a set of multimedia applications encoded in the extended Manifold framework we know in advance if they will meet their deadlines or not. This scheduling strategy has been designed for multimedia applications that operate in a distributed computing environment where every multimedia task is allocated on a different node and can require the use of global resources. We have focused on continuous media that have an implied temporal dimension, i.e. they are presented at a particular rate for a particular length of time and if the required rate of presentation is not met the integrity of these media is destroyed. The proposed synchronization protocol places an upper bound on the task blocking duration and once the blocking durations have been computed we can easily determine the schedulability of a set of tasks. This protocol refers to the maintenance of real-time constraints across continuous media streams and can ensure for example that audio is presented with the required throughput, jitter and latency characteristics, often referred to as quality of service parameters of the media stream. Thus, it is an approach for deterministic guarantees and provides *predictable* distributed multimedia applications. Contrasting with other approaches ([11]) ours pays more emphasis on issues of (dynamic) predictability rather than enforcing fixed priority schemes.

Our work is mainly focused on application-level coordination techniques, specifying language constructs for real-time generation and delivery of events. An efficient and accurate implementation of these constructs requires a minimum set of real-time operating system primitives that support scheduling of processes and resource management according to real-time constraints and a complete real-time clock facility.

References

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

[2] G, Blair, J-B. Stefani, *Open Distributed Processing and Multimedia*, Addison-Wesley, 1998.

[3] N. Halbwachs, *Synchronous Programming of Reactive Systems*, Kluwer Academic Publishers, 1993.

[4] J.P. Lehoczky, L. Sha, J.K. Strosnider and H. Tokuda, "Fixed Priority Scheduling Theory for Hard Real-TimeSystems", *Foundations of Real-Time Computing: Scheduling and Resource Management*, Editor A.M. van Tilborg and G.M. Koob, Kluwer Academic Publishers, 1991, Ch. 1, pp. 1-30.

[5] R. Meier, *State of the Art review of Distributed Event Models*, M.Sc. Thesis, Department of Computer Science, University of Dublin, 2000.

[6] C. Mourlas and C. Halatsis, "Task Synchronization for Distributed Real-Time Applications", *Ninth Euromicro Workshop on Real-Time Systems*, Toledo, Spain, 11-13 June, 1997, IEEE Computer Society Press, pp. 184-190.

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

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

[9] M. Papathomas. G. S. Blair and G. Coulson, "A Model for Active Object Coordination and its Use for Distributed Multimedia Applications", LNCS, Springer Verlag, 1995, pp. 162-175.

[10] P. Pazandak and J. Srivastava, "The Temporal Language Component of DAMSEL: An Embeddable Event-driven Declarative Multimedia Specification Language", *IEEE International Conference on Multimedia Computing and Systems (IEEE MMS'96)*, Hiroshima, Japan, June, 1996, IEEE Computer Society Press.

[11] R. Rajkumar, *Synchronization in Real-Time Systems: A Priority Inheritance Approach*, Kluwer AP, 1991.

[12] S. Ren and G. A. Agha, "RTsynchronizer: Language Support for Real-Time Specifications in Distributed Systems", *ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems*, La Jolla, California, 21-22 June, 1995.

[13] F. Thoen, M. Cornero, G. Goossens and H. de Man, "Software Synthesis for Real-Time Information Processing Systems", *ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems*, La Jolla, California, 21-22 June, 1995.