Asynchronous Timed Multimedia Environments Based on the Coordination Paradigm

George A. Papadopoulos

Department of Computer Science
University of Cyprus
75 Kallipoleos Str.
Nicosia, CY-1678, P.O. Box 537, CYPRUS
george@cs.ucy.ac.cy

Abstract. This paper combines work done in the areas of Artificial Intelligence, Multimedia Systems and Coordination Programming to derive a framework for Distributed Multimedia Systems based on asynchronous timed computations expressed in a certain coordination formalism. More to the point, we propose the development of multimedia programming frameworks based on the declarative logic programming setting and in particular the framework of object-oriented timed concurrent constraint programming (OO-TCCP). The real-time extensions that have been proposed for the concurrent constraint programming framework are coupled with the object-oriented and inheritance mechanisms that have been developed for logic programs yielding an integrated declarative environment for multimedia objects modelling, composition and synchronisation. Furthermore, we show how the framework can be implemented in the general purpose coordination language Manifold, without the need for using special architectures or real-time languages.

Keywords: Multimedia; Timed Concurrent Constraint Programming; Timed Asynchronous Languages; Coordination; Distributed Computing.

1 Introduction

The development of distributed multimedia frameworks is a quite common phenomenon in our days. Furthermore, any distributed programming environment can be viewed as being comprised by two separate components: a computational part consisting of a number of concurrently executing processes and responsible for performing the actual work, and a communication/coordination part which is responsible for inter-process communication and overall coordination of the executing activities. This has led to the development of the so called *family of coordination models and languages* ([3, 12]) which can be used to support the coordinated distributed execution of a number of concurrently executing agents.

The purpose of this paper is to present a framework for coordinating the distributed execution of multimedia applications exhibiting real-time behaviour. However, unlike most of the other approaches that are primarily based on using special purpose real-time languages and platforms ([2, 6, 7, 8, 13]), our model is based on declarative programming and, in particular, that of concurrent constraint programming. More to the point, we show how the *timed* version of concurrent constraint programming ([15]), combined with already existing techniques supporting object-oriented programming ([5]), can be used to produce a framework for multimedia programming which we name OO-TCCP. We then show how a general purpose coordination formalism, namely Manifold ([1]), can be used to support the run-time environment that satisfies the real-time execution requirements of OO-TCCP agents, thus effectively presenting an implementation of OO-TCCP in Manifold, or in other words, a coordination formalism for distributed multimedia applications.

The rest of the paper is organised as follows: The next section presents OO-TCCP and shows how it can be used as the basis for multimedia programming. The following section describes briefly the coordination language Manifold and shows how OO-TCCP can be implemented in it. The last section concludes the paper with a discussion of current and future work.

2 A Declarative Object-Oriented Real-Time Multimedia Programming Framework

Timed concurrent constraint programming (TCCP), developed by Saraswat *et al.* ([15]), is an extension of concurrent constraint programming, itself being a combination of constraint logic programming and concurrent logic programming, with temporal capabilities along the lines of state-of-the-art real-time languages such as ESTEREL, LUSTRE and SIGNAL ([2, 8]), offering temporal constructs and interrupts, and suitable for modelling real-time systems. In TCCP variables play the role of *signals* whose values from one time instance to another can be different. At any given instance in time the system is able to detect the presence of any signals; however, the absence of some signal can be detected only at the end of the time interval and any reaction of the system will take place at the *next* time interval. Thus, the behaviour of a process is influenced by the set of positive information input up to *and including* some time interval t and the set of negative information input up to but not including t. This has been called the *timed asynchrony hypothesis* ([15]) and contrasts the perfect synchrony hypothesis, usually advocated by teal-time languages ([6]). These time intervals t at the end of which no more positive information can be detected are termed the *quiescent* points of the computation.

Thus, the fundamental differences between the timed and the untimed version of concurrent constraint programming are that in the timed version: (i) recursion (and iteration for that matter) are eliminated, and (ii) no information is carried over (by means of variables) from one time instance to the next one. These restrictions guarantee bounded time response and hence a real-time behaviour. Note that the basic ideas characterising TCCP are not unique to concurrent constraint programming and in fact could be introduced into any asynchronous model of computation. It is precisely this property that we exploit in the next section to derive an implementation of the model in terms of a general purpose coordination formalism.

If `c` is a constraint and `A` and `B` are agents, the fundamental temporal construct in TCCP is the following combination:

```
now c then A else B
```

whose interpretation is as follows: if there is enough positive information to entail the constraint `c` then the process reduces immediately (in the current time interval) to `A` and the operations further performed by `A` are also observable immediately; otherwise, if at the end of the current time interval the store cannot entail `c` (i.e. negative information, or in other words, the absence of some signal has been detected), the process reduces to `B` *at the next* time interval (the work performed by `B` will not be observable in the current time instance). As implied by the syntax for the agents above, either of the `then` or `else` parts can be omitted. By „guarding" recursion within an `else` (or `next`) part it can be guaranteed that computation within a time interval is bounded. In fact, reachable states of the computation in a TCCP program can be identified at compile time leading to the generation of a finite state automaton in the same way that this is possible for state-of-the-art real-time languages ([2]). Note that when moving from one time interval to another all the positive information accumulated within the current time interval are discarded. Thus, the value of a program's „variable" varies at different time intervals and any data must either be kept as arguments to the relative predicate or be posted as signals at every time interval.

To recapitulate, at any moment in time a number of agents are executed concurrently exchanging information by means of posting signals to a, possibly only notionally, common store. Each agent is allowed to either suspend waiting for some signal to be posted from some other concurrently running agent, or post itself signal(s) and/or spawn other agents. Any (mutually) recursive call will have to wait until the next time instance. Thus, each (loop-free) agent performs only a bounded amount of work and eventually the whole system quiescences. The store is discarded and computation moves on to the next time instance where only those agents present in the else and next constructs are executed (any agent still remaining suspended in the current time instance is also discarded).

As shown in [15], the above construct can be used to implement a number of temporal constructs that are usually found in real-time languages such as ESTEREL, LUSTRE and SIGNAL . In the sequel we show only the basic ones. The construct

```
whenever c do A = now c then A else (whenever c do A)
```

suspends until the constraint c can be entailed and then reduces the executing process to `A`, thus modelling a temporal *wait* construct. Alternatively, the construct

```
always A = A, next (always A)
```

defines a process that behaves like `A` at every time instance.

Timeouts and interrupts in TCCP can be handled by a `do…watching` construct similar to that found in languages like ESTEREL but with a slightly different semantics. In particular,

```
do A watching c timeout B
```

executes `A` and if `c` becomes true before `A` completes execution, the process will reduce to `B` at the *next* time instance. Since (agent) `A` can be a number of things, the above construct is actually defined by a set of rules rather than a single one, the most important of which are the following ones.

The OO-TCCP framework can be used as the basis for developing multimedia programming environments based on the timed asynchronous paradigm, i.e. frameworks that essentially exhibit soft real-time behaviour; such a framework is reported in [9]. Here, we show how we can model time-based media in OO-TCCP.

```
time_media_object(Object)+(QualityFactor,Duration,Encod
ing,Rate,ScFactor=1,…) :-
          whenever Object:access
            do ( now Object:setScFactor:X then
(ScFactor'=X, next self),
                now Object:setDuration:Length then
(Duration'=Length, next self),
                etc. for the rest of the set function
primitives
                now Object:getRate then
({Object:rate:Rate}, next self),
                now Object:getEncoding then
({Object:encoding:Encoding}, next self),
                etc. for the rest of the get function
primitives
               ).
```

The above code defines a time-based media object comprising a name, which plays effectively the role of a communication channel, and a set of attributes such as quality factor (eg. VHS or CD depending on whether it is video, sound, etc.), duration (in, say, seconds) and rate of presentation (in frames for video or samples for audio). Note that all the attributes are defined as implicit arguments; note also that the scaling factor has a default value of 1. The main part of the code defines its interface where we note that the object remains suspended until it receives an initial message `Object:access`; upon receiving such a message `time_media_object` expects the presence of an accompanying message which can belong to either of two categories: i) it can be an updating type of message (implementing effectively the set type of primitive functions) in which case it updates the relevant parameter (and calls itself recursively at the *next* time instance), or ii) it can be a request type of message (implementing effectively the get type of primitive functions) in which case it posts a signal with the value(s) of the requested parameter(s). Note that the accompanying message may be a parameterised one carrying complicated information that should be passed on by the object to some other agent (e.g. some device driver) for processing. We do not explore this scenario any further here.

We can use the above object class to define a video and an audio object subclass as follows.

```
video_object(Video)+(QualityFactor="VHS",Duration,Encod
ing,Rate,ScFactor,Colour,…) :-
          + time_media_object;
          whenever Video:access
            do ( now Video:setColour:C then (Colour'=C,
next self),
               etc. for the rest of the set function
primitives particular to this object
               now Video:getColour then
({Video:colour:Colour}, next self),
               etc. for the rest of the get function
primitives particular to this object
               now Video:play
               then (video_device_PLAY(Video,…), next
self),
               now Video:stop
               then (video_device_STOP(Video), next
self)
                ).


audio_object(Audio)+(QualityFactor=„CD‟,Duration,Encodi
ng,Rate,ScFactor,Volume,…) :-
          + time_media_object;
          whenever Audio:access
            do (now Audio:setVolume:V then (Volume'=V,
next self),
                etc. for the rest of the set function
primitives particular to this object
                now Audio:getVolume then
({Audio:volume:Volume}, next self),
                etc. for the rest of the get function
primitives particular to this object
                now Audio:play then
(audio_device_PLAY(Audio,…), next self),
                now Audio:stop then
(audio_device_STOP(Audio), next self)
                possibly other control signals
particular to this object
             ).
```

Note that both objects inherit the methods handling the common signals of their superclass. Note also that there is a third category of messages, that of control messages (such as START or STOP) in which case the appropriate device is accessed.

3 The Coordination Language Manifold

Manifold ([1]) is a control-driven coordination language. In Manifold there are 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. A Manifold process features *ports*, *states* and *events*. All these notions are directly derived from the equivalent constructs of Manifold's underlying coordination model, namely IWIM, which is briefly described in the relevant section of another paper by the author in this proceedings volume.

Figure 1 below shows diagramatically 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.).

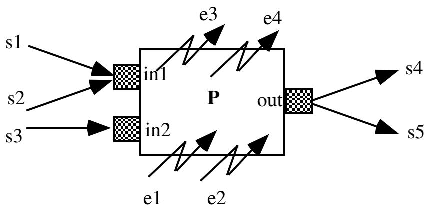

Fig. 1. The basic infrastructure of a Manifold process

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.

Manifold contrasts with the „Linda-like" family of *data-driven* coordination models and languages, where computational components intermix with coordination ones, and coordinator agents see and examine the data involved in some computation. In Manifold all agents are treated as black boxes and there is no concern as to what they actually compute, or indeed whether they are software processes or hardware devices. Thus, this formalism is ideal for coordinating distributed Multimedia frameworks. However, the basic Manifold system does not support real-time behaviour. We show below how the OO-TCCP abstract machine can be implemented in Manifold.

4 Implementing the OO-TCCP Abstract Machine in Manifold

Although Manifold's features were designed with other purposes in mind, we have found them to be suitable in implementing the run-time environment required by OO-TCCP. In particular, a Manifold configuration exhibiting real-time behaviour in the OO-TCCP sense consists of the following components:

- A Manifold coordinator process (the clock) responsible for monitoring the status of the coordinated processes, detecting the end of the current time instance, and triggering the next one. The coordinator process is also responsible for detecting the end of the computation.

- A set of Manifold coordinated processes, each one monitoring the execution of some group of atomic processes. Each such coordinated process performs a bounded amount of work between the ticks as dictated by the coordinator process (thus any loops in such a process „spread over" the next one or more ticks).

- A set of groups of atomic processes (i.e., processes written in some language other than Manifold), each group being monitored by a coordinated process. In order for the whole configuration to exhibit asynchronous real-time behaviour, these atomic processes must also produce results in bounded time. There are two approaches possible here: (i) enforce the constraint that there are no loops within these processes and instead, put these loops in their respective coordinated processes, or (ii) treat them as asynchronous parallel components that take an unbounded amount of time.

The overall configuration is a hierarchical one with the Manifold coordinator process on the top, monitoring a number of Manifold coordinated processes, themselves possibly monitoring groups of atomic (non Manifold) processes. One can regard Manifold as being the „host language“ for writing the control structures of reactive systems, while most of the actual computation (data handling, interfaces with any embedded systems) are done in other more conventional languages, typically C. This fits nicely into the spirit of real-time coordination models as we perceive them and separates the real-time coordination requirements from the rest of the performed activities.

An application featuring timed asynchronous behaviour takes the general form:

```
<application> ::= <coordinator>
                  <coordinated>+ <atomic>+
```

The general behaviour of a coordinator (clock) process is shown, as a first approximation, below (note that the construct (`A1,…,An`) denotes a block where all `n` activities will be executed concurrently; there is also a '`;`' separator imposing sequentiality).

```
manifold Clock()
port in term_in, next_in. port out term_out, next_out.
{
 event tick, next_phase, end_comp.
 begin: (<set up network of initial processes>;
         terminated(void)).
 next_phase: (raise(tick), terminated(void)).
 end_comp: (<perform clean up>; post(end)).
}
```

`Clock` first sets up the initial network of `coordinated` processes. It then suspends waiting for either of the following two cases to become true (one way to achieve suspension in Manifold is by waiting for the termination of the special process `void` which actually never terminates):

- The coordinated processes have completed execution within the current time instance and are waiting for the next clock tick. `Clock` posts the appropriate event (or signal) and suspends again.
- The computation has terminated in which case `Clock` terminates, possibly after performing some clean up.

Detecting the completion of both the current phase and the end of the computation is done in a distributed fashion, provided some constraints regarding the organisation and communication protocols between the participating coordinated processes are imposed. We elaborate further on the exact nature of the work done by `Clock` once we describe the activities performed by a coordinated process.

The general behaviour of a coordinated process is as follows:

```
manifold Process()
port in term_in, next_in.
port out term_out, next_out.
{
 begin:   (<raise event>;
           <wait until input event received>)
```

```
  input_event: (<perform data transfer p1->p2>,
                <generate new processes>;
                terminated(void)).


  tick.Clock: (<perform further actions>;
               post(end)).
}
```

A typical behaviour of a timed asynchronous coordinated process, as understood in the Manifold world, is to post some events, possibly wait until the presence of some event in the current time instance is detected and then react by producing some data transfer between a group of atomic processes that it itself coordinates (say from `p1` to `p2`), post more events and/or generate further processes. Upon termination of its activities within the current time instance, the process suspends waiting for the next `tick` event from the coordinator (`Clock`) process, in which case it performs more activities of similar nature or simply terminates within the current time instance.

We now present in more detail the way detection of the end of the current phase, as well as the whole computation, is achieved. Due to space limitations only the most essential parts of the Manifold code are shown below. The techniques we are using are reminiscent of the ones usually encountered within the concurrent constraint programming community based on short circuits. We recall that the coordinator and each one of the coordinated processes have (among others) two pairs of ports: the `term_in`/`term_out` pair is used to detect termination of the whole computation whereas, the `next_in`/`next_out` pair is used to detect termination of the current clock phase. Upon commencing the computation, the `Clock` process sets up a configuration like the one shown in figure 2 below. This is achieved by means of the following Manifold constructs:

(C.next_out->P1.next_in,…,P3.next_out->C.next_in)

(C.term_out->P1.term_in,…,P3.term_out->C.term_in)

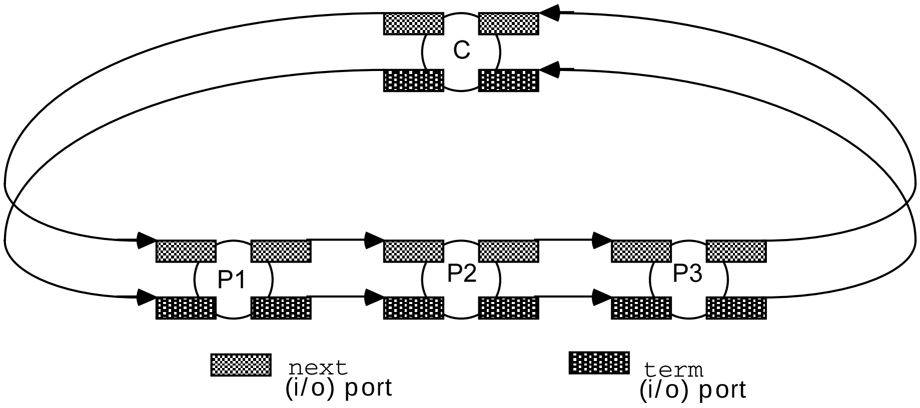

Fig. 2. A short circuit of inter-connecting processes

Any process wishing to further generate other processes is also responsible for setting up the appropriate port connections between these newly created processes. Detecting termination of the whole computation is done as follows: a process P wishing to terminate, first redirects the stream connections of its input and output term ports so that its left process actually bypasses P. It also sends a message down the term.in port of its right process. If P's right process is another coordinated process the message is ignored; however, if it happens to be the Clock controller, the latter sends another message down its term.out port to its left process. It then suspends waiting for either the message to reappear on its term.in port (in which case no other coordinated process is active and computation has terminated) or a notification from its left coordinated process (which signifies that there are still active coordinated processes in the network). The basic Manifold code realising the above scenario for the benefit of the Clock controller is shown below.

```
begin:    guard(term_in,transport,check_term).
check_term: ("token" -> term_out, post(begin)).
got_token: post(begin).
check_term: post(end).
```

A guard process is set up to monitor activity in the term.in port. Upon receiving some input in this port, guard posts the event check_term, thus activating Clock which then sends token down its term.out port waiting to get either a got_token message from some coordinated process or have token reappear again. The related code for a coordinated process is as follows:

```
begin:    guard(term_in,transport,check_term).
check_term: (<term_in->void, if data in port is "token"
               raise(got_token)>).
```

Detecting the end of the current time instance is a bit more complicated. Essentially, quiescence, as opposed to termination, is a state where there are still some processes suspended waiting for events that cannot be generated within the current time instance. We have developed two methods that can detect quiescent points in the computation. In the first scheme, all coordinated processes are connected to a Clock process by means of reconnectable streams between designated ports. A process that has terminated its activities within the current time instance breaks the stream connection with Clock whereas a process wishing to suspend waiting for an event e first raises the complementary event i_want_e. Provided that processes wishing to suspend but also able to raise any events for the benefit of other processes, do so before suspending, quiescence is the point where the set of processes still connected to Clock is the same as the set of processes that have raised i_want_e events. The advantage of this scheme is that processes can raise events arbitrarily without any concern about them being received by some other process. The disadvantage however is that it is essentially a centralised scheme, also needing a good deal of run-time work in order to keep track of the posted events.

An alternative approach requiring less work that is also distributable is a modification of the protocol used to detect termination of the computation: a process wishing to suspend waiting for an event performs the same activities as if it were about to terminate (i.e. have itself bypassed in the port connections chain) but this time using the `next` input/output ports. A process wishing to raise an event before suspending (or terminating for that matter) does so, but waits for a confirmation that the event has been received before proceeding to suspend (or terminate). A process being activated because of the arrival of an event, adds itself back into the next ports chain. Quiescence now is the point where the `Clock` detects, as before, that its `next.out` port is effectively connected to its own `next.in` port, signifying that no event producer processes are active within the current time instance. Note that unlike the case for detecting termination, here the short circuit chain can shrink and expand arbitrarily. Nevertheless, it will eventually shrink completely provided that the following constraints on raising events are imposed:

- Every raised event must be received within the current time instance so that no events remain in transit. An event multicast to more than one process must be acknowledged by all receiver processes whose number must be known to the process raising the event; this latter process will then wait for a confirmation from all the receiver processes before proceeding any further.

- A process must perform its activities (where applicable) in the following order: 1) raise any events, 2) spawn any new processes and set up the next and term port connections appropriately, 3) suspend waiting for confirmation of raised events, 4) repeat the procedure.

The code for the `Clock` controller is very similar to the one managing the `term` ports, with the major difference that upon detecting the end of the current phase `Clock` raises the event `tick`, thus reactivating those coordinated processes waiting to start the activities of the next time instance.

```
begin:    guard(next_in,transport,check_term).
check_term: ("token" -> next_out, post(begin)).
got_token: post(begin).
check_term: (raise(tick), post(begin)).
```

The code for a coordinated process is as follows:

```
some_state: { begin: (raise(e),
              <possibly spawn other processes>;
              terminated(void)).
              i_got_e:
            }
<continue>
```

The framework presented above can be used to implement the OO-TCCP primitives and, thus, provide a Manifold-based implementation for OO-TCCP. We show below the implementation of three very often used such primitives:

```
manifold Whenever_Do(event e, process p)
{
 begin:   terminated(void).
 e:       activate(p).
 tick.Clock: { ignore *.
```

```
                 begin: post(begin).
               }.
  }

  manifold Always(process p)
  {
   begin: (activate(p), terminated(void)).
   tick.Clock:post(begin).
  }
  manifold Do_Watching(process p, event e)
  {
   begin: (activate(p), terminated(void)).
   e: {     begin: terminated(void).
            tick.Clock: raise(abort). }.
            tick.Clock:  terminated(void).
  }
```

Note that `ignore *` clears the event memory of the manifold executing this command. By using `ignore` a „recursive" manifold can go to the next time instance without carrying with it events raised in the previous time instance.

5 Conclusions; Related and Further Work

We have presented an alternative (declarative) approach to the issue of developing multimedia programming frameworks, that of using object-oriented timed concurrent constraint programming. The advantages for using OO-TCCP in the field of multimedia development are, among others, the use of a declarative style of programming, exploitation of programming and implementation techniques that have developed over the years, and possible use of suitable constraint solvers that will assist the programmer in defining inter and intra spatio-temporal object relations. Furthermore, we have shown how this framework can be implemented in a general purpose coordination language such as Manifold in ways that do not require the use of specialised architectures or real-time languages.

Our approach contrasts with the cases where specialised software and/or hardware platforms are used for developing multimedia frameworks ([2, 7, 13]), and it is similar in nature to the philosophy of real-time coordination as it is presented, for instance, in [4, 14]. We believe our model is sufficient for *soft* real-time Multimedia systems where the *Quality of Service requirements* impose only soft real-time deadlines.

References

1. F. Arbab, I Herman and P. Spilling: An Overview of Manifold and its Implementation, Concurrency: Practice and Experience, Vol. 5, No. 1 (1993), 23–70
2. G. Berry: Real-Time Programming: General Purpose or Special Purpose Languages, Information Processing '89, G. Ritter (ed.), Elsevier Science Publishers, North Holland (1989), 11–17

3. N. Carriero and D. Gelernter: Coordination Languages and their Significance, Communications of the ACM **35(2)** (Feb. 1992), 97–107
4. S. Frolund and G. A. Agha: A Language Framework for Multi-Object Coordination, ECOOP'93, Kaiserslautern, Germany, LNCS 707, Springer Verlag, (July 1993), 346–360
5. Y. Goldberg, W. Silverman and E. Y. Shapiro: Logic Programs with Inheritance, FGCS'92, Tokyo, Japan, Vol. 2 (June 1-5 1992), 951–960
6. N. Halbwachs: Synchronous Programming of Reactive Systems, Kluwer (1993)
7. F. Horn, J. B. Stefani: On Programming and Supporting Multimedia Object Synchronisation, The Computer Journal, Vol. 36, No 1. (1993), 4–18
8. IEEE Inc. Another Look at Real-Time Programming, Special Section of the *Proceedings of the IEEE* **79(9)** (September 1991)
9. G. A. Papadopoulos: A Multimedia Programming Model Based On Timed Concurrent Constraint Programming, International Journal of Computer Systems Science and Engineering, CRL Publs., Vol. 13 (4) (1998), 125–133
10. G. A. Papadopoulos: Distributed and Parallel Systems Engineering in Manifold, Parallel Computing, Elsevier Science, special issue on Coordination, Vol. 24 (7) (1998), 1107–1135
11. G. A. Papadopoulos, F. Arbab: Coordination of Systems With Real-Time Properties in Manifold, Twentieth Annual International Computer Software and Applications Conference (COMPSAC'96), Seoul, Korea, 19–23 August, IEEE Press (1996), 50–55
12. G. A. Papadopoulos, F. Arbab: Coordination Models and Languages, Advances in Computers, Academic Press, Vol. 46 (August 1998), 329–400.
13. M. Papathomas, G. S. Blair, G. Coulson: A Model for Active Object Coordination and its Use for Distributed Multimedia Applications, Object-Based Models and Languages for Concurrent Systems, Bologna, Italy, LNCS 924, Springer Verlag (July 5, 1994), 162–175
14. S. Ren, 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 (June 21–22 1995)
15. V. A. Saraswat, R. Jagadeesan, V. Gupta: Programming in Timed Concurrent Constraint Languages, Constraint Programming, B. Mayoh, E. Tyugu and J. Penjam (eds.), NATO Advanced Science Institute Series, Series F: Computer and System Sciences, LNCS, Springer Verlag (1994)