Synchronization of Multimedia Objects Using Concurrent Constraint Programming Techniques

George A. Papadopoulos
Multimedia Research & Development Laboratory,
Department of Computer Science,
University of Cyprus,
75 Kallipoleos Str.,
Nicosia, T.T. 134, P.O. Box 537,
CYPRUS
e-mail: george@jupiter.cca.ucy.cy

Abstract

The problem of synchronization of multimedia objects is addressed within the framework of concurrent constraint programming by presenting an analysis on how the timed version of concurrent constraint programming can be used to model the temporal behaviour and relationships of multimedia objects. The implementation of a non-trivial multimedia application is presented, using the techniques discussed in the paper. Finally, some of the advantages and repercussions of introducing the model of concurrent constraint programming into the research field of multimedia object modeling and synchronization are addressed along with further current and future work.

Keywords: Synchronization of Multimedia Objects, Constraint Concurrent Real-Time Programming.

1 Introduction

The development of multimedia applications involves the managing of, often, complex issues such as programming the behaviour of a variety of media objects (still and motion video frames, audio samples, text), expressing the spatial and temporal relationships between and within multimedia objects, and using special hardware. This leads to a number of problems, among others the lack of knowledge regarding novel programming concepts required in the handling of, say, audio recording or video production, portability issues, etc. A way of handling these problems is the development of high-level user-friendly multimedia programming frameworks that hide away hardware dependencies and media characteristics and provide abstractions suitable for expressing easily the, often, complex modeling and synchronization programming paradigms.

In this paper we address one of the major issues involved in the development of high-level multimedia programming frameworks, that of synchronization. Unlike most of the other approaches that are based on imperative programming techniques, our model is based on declarative programming and, in particular, that of *concurrent constraint programming* ([1]). More to the point, we show how the *timed* version of concurrent constraint programming ([2]) can be used to model and support the synchronization requirements of multimedia objects. However, our work should also be seen as the first step towards the design and implementation of a high-level multimedia programming framework based on declarative programming. To the best of our knowledge this is the first time that an attempt is made to use declarative programming in the development of multimedia programming frameworks.

The rest of the paper is then organised as follows: The next section discusses some of the issues related to multimedia object synchronization followed by a brief introduction to the timed version of concurrent constraint programming ([2]) in order to make the paper self-contained. The third section presents the development of a complete non-trivial programming example using timed concurrent constraint programming, thus showing the capabilities of the model in handling the requirements imposed by multimedia object synchronization. The fifth section discusses some of the advantages and repercussions of introducing the model of concurrent constraint programming into the research field of multimedia object modeling and synchronization and presents some of the research directions we are currently pursuing.

2 Multimedia Object Synchronization

By synchronization in the context of programming multimedia applications we usually refer to the need for expressing the temporal

© *Tenth International Conference on Advanced Science and Technology (ICAST'94)*, Illinois, USA, 25-26 March, 1994, pp. 75-82.

behaviour of a media object both with reference to the environment (say, signals from the outside environment) but also with reference to the state of other media objects playing simultaneously with the object in question. In particular, a high-level multimedia programming framework should be able, among other things, to express real-time behaviour such as the starting and stopping of some media object, establish or remove connections between various media objects and ensure global synchronization between concurrently executing media objects ([3,4]).

The synchronization constraints that must be expressed fall into the following categories ([3,4]): i) *intramedium* (*intrastream*), that refer to the constraints related to the temporal behaviour (rate of presentation of a single stream of data) of some particular media object (say, a MIDI device) but also to more refined notions such as playout deadlines of an object; ii) *intermedia* (*interstream*), that refer to the constraints (joint presentation of multiple data streams) occurring between different media objects or instances of the same object.

In addition to providing the necessary abstractions required to model the temporal behaviour of multimedia objects, a multimedia programming environment should also be able to: i) be based on formal semantics allowing formal reasoning about the written programs, and ii) offer the implementation support required to guarantee the temporal behaviour of the media objects as expressed by a program. Note that the second issue is particularly important in the case of distributed multimedia applications such as teleconferencing, remote visualization or groupware, where there is a need to satisfy the so called Quality of Service (QOS) requirements of the application.

To this end a number of proposals have been put forward ([4,5]) advocating principles from real-time systems and in particular the so called *perfect synchrony hypothesis* ([6]) which states that a reactive object is expected to react instantaneously to the presence of input signals. Consequently, the system must be able to detect at any instance both the presence and the absence of signals.

3 Timed Concurrent Constraint Programming

Concurrent Constraint Programming (CCP) was proposed ([1]) as a natural generalisation and extension of Constraint Logic Programming ([7]) with the concurrency features found in Concurrent Logic Programming ([8]). Whereas in traditional (imperative) programming the store is viewed as a set of variables which are eventually instantiated to fully formed values, in CCP the store is itself a constraint (or rather a set of constraints) providing at every instance (perhaps partial) information regarding the values of a program's variables. A program then consists of a number of concurrently executing processes (agents) that interact with the (shared) store in order to either post there more information or check whether some information holds. Instead of the traditional *read* and *write* operations, CCP introduces, respectively, the *ask* operation (check whether the store *entails* a given constraint) and the *tell* operation (post a new constraint to the store). An *ask* operation succeeds if the required information can be found in the store, fails if the asked constraint cannot be entailed and *suspends* if the store does not have enough information to either entail or disentail an asked constraint, thus providing the required synchronization mechanism. A *tell* operation succeeds if the constraint to be told is consistent with the information that is already present there and fails otherwise. Thus the variables in the store are incrementally and monotonically refined, never being in an inconsistent state. Note that the domain of discourse over which the values of variables range is orthogonal to the rest of the framework and can be one of several kinds (real numbers, finite domains, etc.).

Recently ([2]), the CCP framework has been extended to a *timed* version (TCCP), along the lines of state-of-the-art real-time languages such as ESTEREL, LUSTRE and SIGNAL ([6,9]), offering temporal constructs and interrupts, and suitable for modeling real-time systems. In TCCP declarative variables play the role of *signals* whose values from one 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 instance. Thus, the behaviour of a process is influenced by the set of positive information input up to *and including* some time instance t and the set of negative information input up to but not including t. This has been called the *timed asynchrony hypothesis* ([2]) and contrasts the perfect synchrony hypothesis mentioned in the previous section. Note that, since it can be ensured that the state of the system is bounded, the time taken at each stage of the computation will be bounded.

In TCCP, a program is a collection of procedures that take the following form:

Agents	**A ::=**	**c**	**telling a constraint**
		| now c then A	**timed positive ask**
		| now c else A	**timed negative ask**
		| A, A	**parallel conjunction**
		| X ^ A	**defining a local signal X**
		| p(t1,...,tn)	**procedure call**
Declaration			
	D ::=	**p(t1,...,tn)::A**	**procedure definition**

The sole temporal construct in TCCP is the following:

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 to **A**; otherwise, if at the end of the current time instance 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 instance. Note that either the **then** or the **else** part can be omitted as shown in the syntax diagram above. Note also that the "temporal" constraints **c** should be viewed as *signals* (non-rigid variables) which hold only for the current time instant and not necessarily for any subsequent ones. That is why only the current instance of a variable **c** is allowed to be accessed and any need to store "persistent" data should be satisfied by means of extra parameters in the relevant recursive procedures.

As shown in [2], the above construct can be used to implement a number of temporal constructs that are usually found in real-time languages. In the sequel we show only those that are used in the rest of this paper. 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 modeling a temporal wait construct. On the other hand, the following construct

```
now
    case c1 do A1          now c1 then A1,
    case c2 do A2          now c2 then now c1 else A2,
    ...
    case cn do An          now cn then now (c1 AND ...
                           AND cn-1) else An,

    default B              now (c1 AND ... AND cn) else B
end
```

models a prioritised wait: the process reduces to **Ai** if **ci** is now true but **c1** to **ci-1** *were* false at the previous time instances.

Timeouts can be modelled by the construct **c before T then A else B** which reduces to **A** if **c** is entailed within **T** time units; otherwise it reduces to **B** at the end of the specified period **T**. This construct is defined as follows:

```
c before 1 then A else B = now c then A else B
c before (T+1) then A else B =
    now c then A else (c before T then A else B)
```

Finally, 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. The most important rules defining the above construct are the following:

```
do d watching c timeout A = d
do (now d then A) watching c timeout B =
    now d then (do A watching c timeout B)
do (now d else A) watching c timeout B =
    now (d AND c) else then (do A watching c timeout B),
    now c then next B
```

Note that **next A** is an abbreviation for **now false else A** where the intended interpretation is to behave like **A** from the next instance onwards.

4 Multimedia Synchronization in TCCP

4.1 TCCP as a Multimedia Synchronization Sublanguage

What sort of synchronization requirements must be satisfied by some programming formalism in order to be able to act as a basis for designing a multimedia scripting (sub)language? The following comprises a non-exhaustive list:

- ability to express delays relative to absolute time or the behaviour of other media objects,
- a formal treatment of time,
- time-constrained synchronization of processes,
- sequential, parallel, repetitive behaviour of processes,
- exception handling,
- ability to provide generic solutions to synchronization problems that can be used in a wide variety of scenaria.

TCCP supports all the above points, and more. It offers a programming environment based on declarative programming with well defined and fully abstract operational and denotational semantics, inheriting all the programming techniques that over the years have been developed in the field of concurrent logic languages ([8]). A rich variety of temporal constraints can be defined, able to express the temporal behaviour of multimedia objects. In addition, other constraint systems can be added, offering powerful techniques for expressing the spatial constraints of multimedia objects. The model is naturally parallel, supporting a high degree of concurrency although, if desired, sequential behaviour can be enforced (e.g. by using nested **now...then** constructs). Repetition is achieved by means of recursive procedures.

Although not discussed extensively in this paper, the model can exploit the object-oriented capabilities of logic programming and, more to the point, concurrent logic programming ([10,11,12]).

We believe that the timed asynchrony hypothesis advocated by TCCP is not an obstacle in using the model for multimedia object synchronization, especially due to the fact that a certain time delay can be tolerated (thus, placing multimedia systems in the category of *soft* real-time systems). In particular, for any (composite) multimedia object **C** the following formula should hold for all its components **Ci** ([3]):

| C.current_world_time - Ci.current_world_time | < Δi

where **Δi** is the synchronization tolerance for every component **Ci**. A certain synchronization tolerance is also expected between two objects **M1** and **M2** expressed by a similar formula:

| ObjToWorld(M1.object_time) -
ObjToWorld(M2.object_time) | < Δ

where the function **ObjToWorld** translates an object's relative (internal) time to world (the application's) time. Note also, that the timed asynchrony hypothesis may allow more effective synchronization in distributed multimedia applications ([13]).

In the next section we illustrate the ability of TCCP to act as a basis for developing a multimedia scripting environment by coding up a complete non-trivial application using TCCP techniques and primitives. To set the stage we show here the implementation of some simple programming examples related to multimedia object synchronization. Parameters in all capital letters denote signals coming from or going to external devices, whereas the rest denote local signals (within an object or between two or more objects) and ordinary data.

We start with an implementation of the primitive **Wait_For(N,SIG)**, found in ESTEREL, which suspends a process until the signal **SIG** has appeared **N** times (lines starting with % denote comments).

```
% wait for signal SIG to appear N times

Wait_For(N,SIG) ::
    whenever SIG
      do ( now N=1 then true else Wait_For(N-1,SIG) ).
```

Note that the call **Wait_For(0,SIG)** waits indefinitely (by entering an infinite recursion). The above primitive can be used to define the agent **Play_For(MM,TU,SIG)** which plays the media object **MM** for **TU** time units defined by the signal **SIG** (seconds, minutes, or otherwise).

```
%   Play multimedia object MM for TU time units
%   (defined by SIG)

Play_For(MM,TU,SIG) :: START(MM),
                       whenever Wait_For(TU,SIG)
                         do STOP(MM).
```

Note that we assume the existence of the signals **START(MM)** and **STOP(MM)** that start and stop respectively the device for the object **MM**. So, the call **Play_For(MM,5,SECOND)** will play the media object **MM** for 5 seconds (assuming the constant emission of the signal **SECOND** from a timing device). The next primitive, **Delay_By(MM,Delay)**, suspends an agent executing it until the media object **MM** has played for **Delay** time units.

```
%   suspends until the multimedia object MM
%   has been playing for Delay time units

Delay_By(MM,Delay) ::
    now STARTED(MM)
      then now Delay=0
             then true
             else Delay_By(MM,Delay-1).
```

Note that here we make use of a primitive **STARTED(MM)** which polls the device of the media object **MM** and returns **true** if **MM** is still playing and **false** otherwise. The above primitives can be used, for instance, to model the following scenario: Objects **MM1** and **MM2** start playing concurrently for a period of **TU1** and **TU2** time units respectively. Object **MM3** starts playing after object **MM1** has played for **TU1** time units (depending on what **SIG** is) and stops when object **MM2** has played for **TU2** time units).

```
scenario(MM1,TU1,MM2,TU2,MM3,Delay1,Delay2,SIG) ::
    Play_For(MM1,TU1,SIG),
    Play_For(MM2,TU2,SIG),
    whenever Delay_By(MM1,Delay1),
      do (do Play_For(MM3,0,SIG) % play forever
          watching Delay_By(MM2,Delay2) ).
```

Note that in the above example we have adopted a *temporal relation* rather than a *time-line* approach (see also related discussion in the next section on the benefits of using this approach).

As a final simple example we show how the sequential and parallel elementary synchronization operators proposed by MHEG — the Multimedia and Hypermedia information coding Group ([14]) — can be implemented in TCCP.

```
%   play M1 after a delay of T1 followed by M2 after a
%   delay of T2 from the time instance M1 has ended

seq(M1,T1,M2,T2) ::
    now Wait_For(T1,SIG)
```

```
then ( START(M1),
       whenever STOP(M1)
          do now Wait_For(T2,SIG)
             then START(M2) ).
```

The implementation of the parallel operator is slightly simpler.

```
%  play in parallel M1 after a delay of T1 and M2 after a
%  delay of T2 from the time instance the parent object
%  has invoked the operation

par(M1,T1,M2,T2) ::
     now Wait_For(T1,SIG) then START(M1),
     now Wait_For(T2,SIG) then START(M2).
```

4.2 The 'Information Kiosk for Cyprus' Example

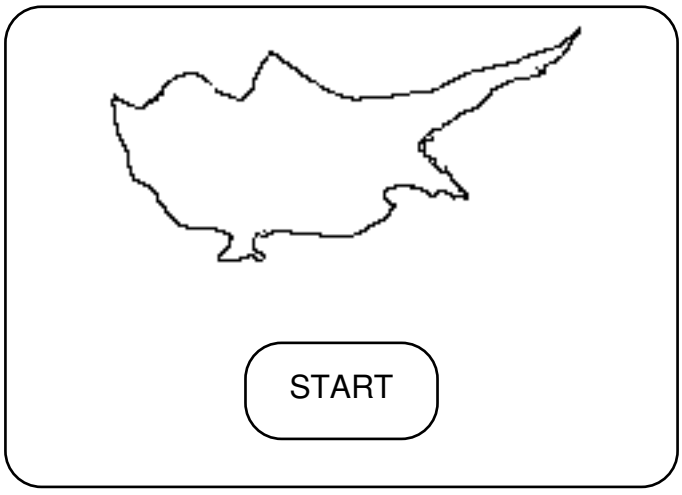
In the previous section we showed the capability of TCCP to model various simple synchronization scenaria and implement proposed synchronization operators and primitives. In this section we show the implementation of a complete non-trivial example using TCCP techniques as a means to illustrate the usefulness of timed concurrent constraint programming as a high-level tool for building multimedia applications. The example we have in mind is a variant of the *Virtual Museum* example ([3,4]) similar to the one described in [5]; more to the point, the application is about an information kiosk for the Mediterranean island of Cyprus providing various sorts of information (tourist, archeological, geographical, etc.) in an interactive way. Due to lack of space we show only a simplified version of the application with minimal functionality and focused only onto one area of information, that of flora and fauna. The application consists of three phases as explained below.

During the first phase the application shows the map of Cyprus and plays continuously an audio commentary inviting people to use it. The second phase begins when the user presses a START key. A menu is displayed informing the user of the various sorts of information available and a cyclic audio commentary is played explaining to the user how to select a particular information category.

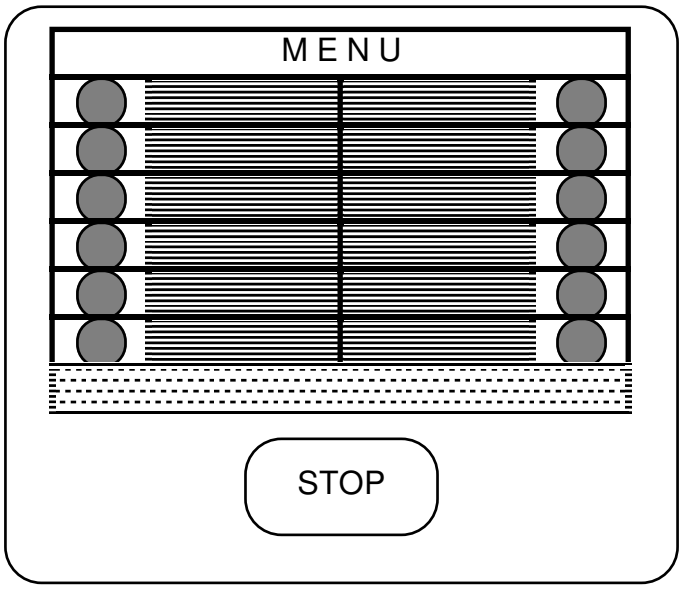
Once the user has made a selection by pressing the appropriate button, the application starts showing a *diaporama*, i.e. a combination of a sequence of still video images accompanied by an audio commentary. In addition, part of the display is occupied by a small map of Cyprus where the location of the various exhibits presented in the diaporama is shown.

When the third phase has completed execution, the application goes back to the first phase. The following synchronization requirements are imposed:

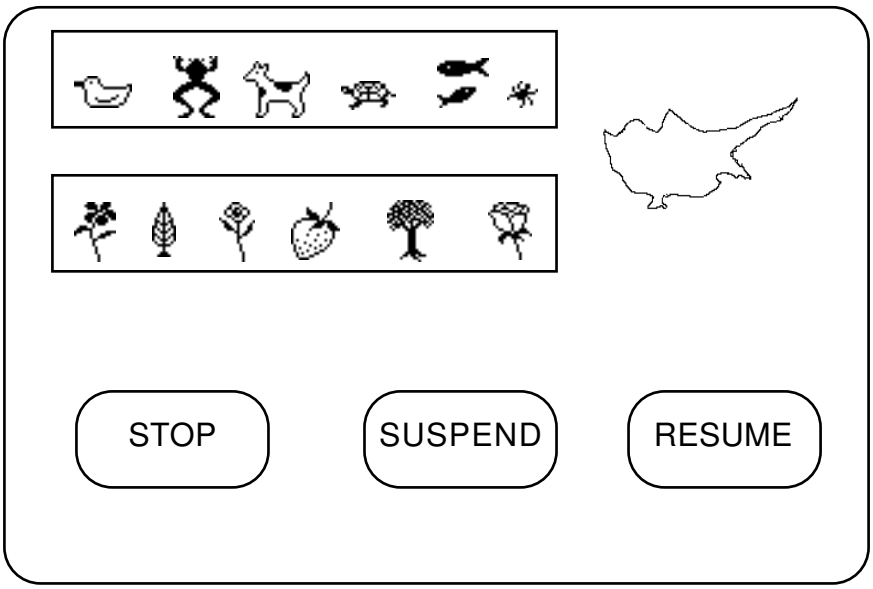
- If the user has not responded within 5 minutes during the second phase, it is assumed that he has left and the application goes back to the first phase.
- The third phase can be suspended and resumed by means of suitable buttons. If suspended, the application resumes automatically in 30 seconds unless the RESUME key has been pressed earlier. In addition, both phases can be killed, in which case the application goes back to the beginning.
- Finally, during the third phase, audio and video are synchronized in the following sense: for every still image a corresponding audio commentary is played that starts 2 seconds after the display of the image.



PHASE 1



PHASE 2



PHASE 3

The implementation of the above scenario which is shown below exploits both the object-oriented and the temporal capabilities of the timed concurrent constraint paradigm. The program consists of a number of agents running in parallel and communicating by means of signals. Along the lines of the application described in [5], each agent models in a generic way some media behaviour. In addition, temporal behaviour of objects is expressed by means of temporal constraints rather than following a time-line approach. This has the advantage of preserving references and links between media objects when temporal parameters are changed. In addition, the "tempo" of the application is kept by a separate object (the **CLOCK**) whose code is not shown here due to lack of space; the idea here is that when the application is suspended timeouts and delays should be freezed.

The following procedure, **FI_CA**, models the behaviour of a composite multimedia object comprising the display of a still image and the cyclic playing of an audio commentary. Note that the procedure, being generic, can be used in both phases 1 and 2 by simply calling it with different parameters.

```
% Fixed Image - Cyclic Audio

FI_CA(VIDEO_START,AUDIO_START,AUDIO_END,STOP) ::
        do ( VIDEO_START, AUDIO_START,
            whenever AUDIO_END do AUDIO_START )
        watching STOP.
```

The next procedure, **FI_MIA**, models the display of a single image (the map of Cyprus) in connection with a sequence of video images accompanied by audio commentary. The procedure makes use of an auxiliary one, **Cycle_Image_Audio**, that is responsible for modeling the presentation of the diaporama (thus, implementing implicitly a repetitive operator). Here we assume the existence of a function **GET_UNITS(N)**, which returns in **N** the number of video images comprising the user's choice.

```
% Fixed Image - Multiple Image with Audio

FI_MIA(Delay,STOP,Second,FIXED_VIDEO,MOTION_VIDEO,CYCLIC_AUDIO_START,CYCLIC_AUDIO_END)
  :: Units^
        do ( GET_UNITS(Units), FIXED_VIDEO,
            Cycle_Image_Audio(1,Units,Delay,Second,MOTION_VIDEO,CYCLIC_AUDIO_START,
                              CYCLIC_AUDIO_END) )
        watching STOP.

Cycle_Image_Audio(Index,Limit,Delay,Second,MOTION_VIDEO,CYCLIC_AUDIO_START,CYCLIC_AUDIO_END)
  ::
        now Index#Limit+1
        then ( MOTION_VIDEO(Index),
                whenever Wait_For(Delay,Second),
                  do ( CYCLIC_AUDIO_START(Index),
                        whenever CYCLIC_AUDIO_END
                            do next Cycle_Image_Audio(Index+1,Limit,Delay,Second,MOTION_VIDEO,
                                                      CYCLIC_AUDIO_START,CYCLIC_AUDIO_END)) ).
```

Finally, the top agent **kiosk** invokes all the other agents and synchronises their execution. Any variables not defined within the agent are assumed to have been defined in the surrounding context.

```
% Information kiosk - it is called with a list of signals SIGNALS

kiosk(SIGNALS) :: Tempo^
        FI_CA(INTRO_VIDEO,INTRO_AUDIO_START,INTRO_AUDIO_END,PRESENT_START),
        whenever PRESENT_START
          do ( do FI_CA(MENU_VIDEO,MENU_AUDIO_START,MENU_AUDIO_END,MENU_CHOICE)
                watching Wait_For(5,MINUTE) timeout PRESENT_TERM ),
        whenever MENU_CHOICE or PRESENT_TERM
          do ( now
                case PRESENT_TERM
                    do kiosk(SIGNALS)
                case MENU_CHOICE
                    do ( FI_MIA(2,PRESENT_TERM,Tempo,LOCATION_MAP_VIDEO,
                                TOUR_VIDEO,TOUR_AUDIO_START,TOUR_AUDIO_END),
                         State=running^CLOCK(30,State,Tempo,SECOND,SUSPEND,RESUME,
                                             SUSPEND_TOUR,RESUME_TOUR)) ).
```

5 Conclusions and Further Work

We have presented an alternative approach to the issue of modeling and synchronising multimedia objects, that of using (timed) concurrent constraint programming. To the best of our knowledge this is the first time declarative programming (in the form of concurrent constraint programming or otherwise) is proposed as the basis for developing high-level multimedia development frameworks.

The advantages for using 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.

In this paper we have concentrated, almost exclusively, on the synchronization aspects of multimedia applications development. However, we have only touched on the tip of the iceberg. A number of interesting issues that must be resolved lie ahead. First of all, we are currently extending the present work to define a full set of multimedia synchronization primitives using TCCP. Part of this work will involve the examination of more complicated multimedia synchronization problems like the so called "lip Sync" (live synchronization). In addition, we will be using the object-oriented capabilities of the model ([10,11,12]) to extend the present work towards a complete composition and modeling environment ([3,4]). Finally, regarding initial prototype implementations, we are designing a specification language based on TCCP techniques by modifying existing interpreters for concurrent logic languages which will compile the code down to IBM's AVA (Audio Visual Authoring) Language, part of the AVC (Audio Visual Connection) authoring tool, and used in connection with the ActionMedia II card and exploiting Intel's DVI (Digital Video Interactive) technology. Note here that although the user will present the synchronization requirements as temporal relations, these will be represented internally as time-line relations.

Acknowledgments

This work was done as part of the project 'Introducing A.I. Techniques in the Design and Development of Multimedia Information Systems' that has been set up at and pursued by the Multimedia Research and Development Laboratory of the Department of Computer Science in collaboration with the University of Cyprus, the Cyprus Popular Bank Ltd., and IBM SEMEA S.p.A., Cyprus Branch.

References

[1] V. A. Saraswat, *Concurrent Constraint Programming*, Logic Programming and Doctoral Dissertation Award Series, MIT Press, March 1993.

[2] V. A. Saraswat, R. Jagadeesan and V. Gupta, "Timed Concurrent Constraint Programming", TR, Xerox Palo Alto Research Centre, August 1993.

[3] D. Tsichritzis (ed.), "Object Frameworks", Internal Report (collected papers), Centre Universitaire d'Informatique, Université de Genève, Switzerland, 1992.

[4] D. Tsichritzis (ed.), "Visual Objects", Internal Report (collected papers), Centre Universitaire d'Informatique, Université de Genève, Switzerland, 1992.

[5] F. Horn and J. B. Stefani, "On Programming and Supporting Multimedia Object Synchronisation", *The Computer Journal*, Vol. 36, No 1., 1993, pp. 4-18.

[6] G. Berry, "Real-Time Programming: General Purpose or Special Purpose Languages", *Information Processing '89*, G. Ritter (ed.), Elsevier Science Publishers, North Holland, 1989, pp. 11-17.

[7] J. Jaffar and J-L. Lassez, "Constraint Logic Programming", *Proc. 14th ACM Symposium on Principles of Programming Languages*, Munich, Germany, January 1987, pp. 111-119.

[8] E. Y. Shapiro, "The Family of Concurrent Logic Programming Languages", *Computing Surveys*, Vol. 21 (3), 1989, pp. 412-510.

[9] IEEE Special Issue on Another Look at Real-Time Systems (collected papers), September 1991.

[10] A. Davison, *POLKA: A Parlog Object Oriented Language*, Ph.D. Thesis, Department of Computing, Imperial College, London, May 1989.

[11] Y. Goldberg and E. Y. Shapiro, "Logic Programs with Inheritance", *Proc. International Conference of Fifth Generation Computer Systems*, ICOT, Tokyo, Japan, 1992, pp. 951-960.

[12] K. M. Kahn., E. D. Tribble, M. S. Miller and D. G. Bobrow, "Vulcan: Logical Concurrent Objects", *Research Directions in Object Oriented Programming*, P. Shriver and P. Wegner (eds.), MIT Press, 1987.

[13] K. Lakshman and R. Yavatkar, 'PolySchmues: An Integrated Framework for Distributed Multimedia Applications', *IEEE International Workshop on Principles and Applications of Distributed Systems*, Princeton, New Jersey, October 1993, pp. 64-69.

[14] ISO-IEC, "Coded Representation of Multimedia and Hypermedia Information Objects", Multimedia and Hypermedia Information Coding Experts Group (MHEG) Working Document S.5, JTC1/SC29/WG12, 1991, pp. 70-72.