Towards a High-Level Multimedia Modelling & Synchronisation Environment Based on Constraint Programming

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 modelling and synchronisation of multimedia objects is addressed in the declarative logic programming setting and in particular within 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. The expressiveness and ability of OO-TCCP to act as a basis for building high-level, user-friendly authoring environments is illustrated by presenting a simple object model for multimedia composition and synchronisation followed by an analysis on how it can be used to model the temporal behaviour and relationships of multimedia objects. The implementation in OO-TCCP of a realistic multimedia application is presented, using the techniques discussed in the paper. To the best of our knowledge this is the first time logic programming (in the form of concurrent constraint programming or otherwise) is used for multimedia modelling and synchronisation.

© *Sixth International Conference on Computing and Information (ICCI'94)*, Peterborough, Canada, 26-28 May, 1994, Journal of Computing and Information (special issue), pp. 1295-1316.

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 modelling and synchronisation programming paradigms.

In this paper we address two of the major issues involved in the development of high-level multimedia programming frameworks, those of modelling and synchronisation. Unlike most of the other approaches that are primarily based on imperative programming techniques and the use of languages such as C++ ([9,10,11]) or real-time ones ([1]) such as ESTEREL ([5]), 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 ([7]) can be used to model and support the synchronisation requirements of multimedia objects; this is then combined with already existing techniques supporting object-oriented programming (such as [2,3]). This work should 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 (and, in particular, concurrent constraint programming) in the development of multimedia programming frameworks.

The rest of the paper is organised as follows: The next two sections discuss some of the issues related to multimedia object modelling and synchronisation. This is followed by a brief introduction to OO-TCCP, a combination of timed concurrent constraint programming ([7]) with the object-oriented ([2]) and inheritance facilities ([3]) that have been developed for logic programs. The fourth section presents the development of a complete non-trivial

programming example using OO-TCCP, thus showing the capabilities of the model in handling the requirements imposed by multimedia object modelling and synchronisation. The last section concludes the paper with a discussion of current and future work.

2 Multimedia object modelling and composition

By modelling in the context of programming multimedia applications we usually refer to the need for developing suitable abstractions, able to hide away concepts particular to the behaviour and characteristics of a media object, with which are not necessarily familiar all programmers. Such complex issues include, among others, data encoding, compression techniques, quality factors, timing, etc. In addition, these abstractions attempt to produce models which are platform-independent. In our research project we are particularly concerned with the so called *time-based* media ([10]) that comprise digital audio and video, music and animation, and whose functionality includes such aspects as dataflow, timing, and temporal composition and synchronisation.

A complex multimedia application contains a number of multimedia objects, each encapsulating the behaviour and synchronisation of some medium. The behaviours of simple multimedia objects are usually composed to form composite multimedia objects. Our object-oriented framework is a variant of Gibb's "active objects" metaphor ([9,10]) based on the actor model and object-oriented concurrent logic programming techniques.

3 Multimedia object synchronisation

By synchronisation in the context of programming multimedia applications we usually refer to the need for expressing the temporal 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 synchronisation between concurrently executing media objects ([9,10]).

The synchronisation constraints that must be expressed fall into the following categories: 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 and ii) *intermedia* (*interstream*), that refer to the constraints (joint presentation of multiple data streams) occuring 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 ideally be based on formal semantics allowing formal reasoning of the written programs. A number of proposals have been put forward ranging from process calculi such as CSP ([8]) to extending Petri Nets with temporal properties ([6]). Furthermore, the implementation of the model should guarantee the temporal behaviour of the media objects as expressed by a program. To this end a number of proposals have been put forward (such as [5]) advocating principles from real-time systems and in particular the so called *perfect synchrony hypothesis* ([1]) 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.

4 Declarative object-oriented real-time programming

4.1 Timed concurrent constraint programming

Timed concurrent constraint programming (TCCP), developed by Saraswat *et al.* ([7]), is an extension of concurrent constraint programming with temporal capabilities along the lines of state-of-the-art real-time languages such as ESTEREL, LUSTRE and SIGNAL ([1]), 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* ([7]) and

contrasts the perfect synchrony hypothesis mentioned in the previous section. These time intervals t at the end of which no more positive information can be detected are termed the *quiescent* points of the computation.

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 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. Either of the `then` or `else` parts can be omitted. By "guarding" recursion within an `else` part it can be guaranteed that computation within a time interval is bounded; in fact, a TCCP program can be compiled to a finite state automaton. 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 be kept as arguments to the relative predicate.

As shown in [7], 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. 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.

Finally, TCCP supports the powerful construct `clock B on A` which executes A only on those time instances which are quiescent points of B. This allows objects to be sampled at different rates but it also allows the modelling of different notions of time produced by, say, the existence of operations such as pause and resume. Such a behaviour can be part of some special object referred to sometimes as a synchronisation manager ([8]).

The following example illustrates some of TCCP's features.

```
counter(Counter,Value)
   :- whenever Counter:access_counter
         do (now Counter:ask_value
             then ({value_is:Value}, next counter(Counter,Value)),
             now Counter:incr_value
             then (Value1=Value+1, next counter(Counter,Value1)),
             now Counter:decr_value
             then (Value1=Value-1, next counter(Counter,Value1))).
```

The predicate `counter` suspends until it receives a signal `Counter:access_counter` where `Counter` identifies the particular `counter` object and then it examines the parameter of an expected accompanying signal. Depending on what it is, it either posts the current

value or increases/decreases it by 1. In all cases it calls itself recursively at the next time instance, thus achieving bounded time behaviour.

4.2 Object-oriented and inheritance mechanisms for TCCP programs

The ability of concurrent logic programming to support object-oriented programming is well known. In particular, an object is represented as a process which calls itself recursively. The state of the object is represented as a number of unshared (local to that process) arguments, whereas the clauses defining the process represent the object's methods. Communication between objects is achieved by means of sending and/or receiving messages via shared variables. The model is inherently concurrent with elementary media objects forming collectively composite multimedia objects by means of delegating messages to the appropriate object. This framework is enhanced with real-time primitives allowing an object to be subjected to real-time constraints and be activated even if it has not received any messages from other objects.

In addition, a number of techniques ([2,3]) have been developed in order to solve two major problems: explicit reference to all the arguments of a predicate in the recursive call and lack of class-based inheritance mechanisms. Here we combine the TCCP framework with the techniques proposed in [3], adapted to satisfy the constraints imposed by TCCP, to form what will be referred to in this paper as object-oriented timed concurrent constraint programming (OO-TCCP). Due to lack of space a couple of simple examples must suffice to illustrate the characteristics of this combined framework. Consider the case of a simple media object that suspends until a condition `C` is satisfied and then emits the signal `S`. In OO-TCCP this object could be coded up as follows.

```
emit(C,S) :- whenever C do {S}.
```

Now a similar object `emit2` that emits signal `S1` if a condition `C1` is satisfied and alternatively emits signal `S2` if a condition `C2` is satisfied can be coded up in terms of the object `emit` as follows.

```
emit2(C1,C2,S1,S2) :- +emit(C1,S1);
                      +emit(C2,S2).
```

The above code fragment is equivalent to the following expanded one.

```
emit2(C1,C2,S1,S2) :- whenever C1 do {S1},
                      whenever C2 do {S2}.
```

Using the techniques proposed in [3] the previous `counter` predicate can be written as follows.

```
counter(Counter)+(Value=0)
   :- whenever Counter:access_counter
         do (now Counter:ask_value then ({value_is:Value}, next self),
             now Counter:incr_value then (Value1=Value+1, next self),
             now Counter:decr_value then (Value1=Value-1, next self)).
```

Note the use of the operator '+' to denote implicit arguments, possibly with default values (as in this case). Note also the use of the keyword '`self`' to denote (guarded) recursion. Finally, note that primed or indexed variables denote new incarnations of these variables with updated values.

5 Multimedia modelling and synchronisation in OO-TCCP

5.1 OO-TCCP as a multimedia modelling and synchronisation language

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

- ability to express delays relative to absolute time or the behaviour of other media objects,
- object-oriented facilities for encapsulating object behaviour and forming composite objects,
- a formal treatment of time,
- time-constrained synchronisation of processes,
- sequential, parallel, repetitive behaviour of processes,
- exception handling,

- ability to provide generic solutions to modelling and synchronisation problems that can be used in a wide variety of scenaria.

OO-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. 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 of 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 (eg. by using nested `now..then` constructs). Repetition is achieved by means of recursive procedures. Furthermore, the model can exploit the object-oriented capabilities of logic programming and, more to the point, concurrent logic programming ([2,3]).

We believe that the timed asynchrony hypothesis advocated by TCCP is not an obstacle in using the model for multimedia object synchronisation, 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` ([9,10]):

```
| C.current_world_time - Ci.current_world_time | < Δi
```

where `Δi` is the synchronisation tolerance for every component `Ci`. A certain synchronisation 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 synchronisation in distributed multimedia applications.

We illustrate some of OO-TCCP's capabilities in modelling and synchronising (composite) multimedia objects and we show how various temporal behaviours between two or more media objects can be implemented in OO-TCCP. We associate with each object a number of data values related to the spatio-temporal behaviour of that object. The following piece of code declares a class `media_object` which is used to define a media object with some implicit arguments (some of which may have default values) and handle a number of messages to that object. For reasons of brevity we focus our attention on those attributes that are directly relevant to the object's temporal behaviour such as the length and the scaling factor; in addition, we show the handling of only a token number of messages (or combinations of them). Throughout the paper emphasis is on clarity rather than efficiency of coding.

```
media_object(Object)+(Length,ScFactor=1.0,Data) :-
   whenever Object:access
      do (now Object:setScFactor:X then (ScFactor'=X, next self),
          now Object:askLength then ({Object:length:Length}, next self),
          now Object:start
          then (dev_PLAY(Object,…), ack_mess(Object), next self),
          now Object:stop
          then (dev_STOP(Object), ack_mess(Object), next self).
```

The parameter `Data` denotes other relevant to the media object data such as its type (eg. text, video, etc.), description (eg. colour, sampling rate, etc.), size (number of video frames or audio samples), spatial constraints, direction of play, etc. The agent `media_object` remains suspended until it receives the message `Object:access`. We recall here that there are no rigid variables in TCCP and any bindings to be retained must be posted at every time instance. The interaction of an object with its environment is achieved via its name which is effectively used as a communication channel. Upon receiving the message `Object:access`, `media_object` expects the presence of an accompanying message which can belong to either of three categories: i) it can be an updating type of message 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 in which case it posts a signal with the value(s) of the requested parameter(s), or iii) it can be a control type of message in which case it accesses the relevant device for the appropriate action. In the second case we

assume that the requesting agent can detect the posted message in the same time instance; if that is not possible the message with the requested information can be kept posted at every time instance until an appropriate acknowledgment message has been received (here we assume that the synchronisation tolerance of the media involved allows the materialisation of such a scenario). We expect the underlying implementation to be able to support the generation of *boolean* signals describing the state of some object (such as `started`, `stopped`, `playing`, etc.). The agent `ack_mess` is responsible for handling these messages, which incidentally may be generated in various formats by the different devices involved, and communicate them to the rest of the program using a standard format.

```
ack_mess(Object) :- whenever dev_ACK(Object,Message)
                        do ({Object:boolean:Message}, next self).
```

Again here we assume that the message posted by the agent `ack_mess` can be received by an interested agent in the same time instance.

The class `media_object` can now be used to define simple media objects but also *composite* ones. In the latter case a composite object comprises a number of media objects executing either in sequence or concurrently depending on the kind of temporal (and spatial) interrelationships that have been specified. The following piece of code defines a composite multimedia object comprising a video object, an audio object and a text object with lengths of 10, 5 and 3 seconds respectively.

```
composite_object_A :- media_object(video1,10,…),
                      media_object(audio1,5,…),
                      media_object(text1,3,…),
                      scenario_A(30,video1,audio1,text1).
```

The way the above composite object is played is specified by the agent `scenario_A`. In general, there are 13 possible ways to associate two or more objects in time ([6,8]) depending on whether their playing overlaps each other, precedes one another, etc. In addition, the playing of a single object of length L with respect to some specific interval T (where T≠L) can belong to one of several styles depending on whether the object should be played at the beginning, middle or end of the interval, stretched over the interval or repeated

and chopped. Due to space limitations we do not show the implementation of all these cases in OO-TCCP. However, the following non trivial scenario demonstrates how the above temporal intra- and interrelationships can be expressed in OO-TCCP.

```
scenario_A(Delay,Video,Audio,Text)
:- delay_by(Delay,sec),
   whenever continue
      do ({Video:access, Video:askLength,
           Audio:access, Audio:askLength},
          whenever (Video:length:VidLen, Audio:length:AudLen)
             do (NewVidScFact=VidLen/AudLen,
                 {Video:access, Video:setScFactor:NewVidScFact},
                 next scenario_A_cont(Video,Audio,Text))).

scenario_A_cont(Video,Audio,Text)
   :- {Video:access, Video:start, Audio:access, Audio:start},
      whenever (Video:boolean:stopped, Audio:boolean:stopped)
         do ({Text:access, Text:start}).
```

The above piece of code implements the following scenario: After a delay of 30 seconds play together `Video1` and `Audio1` which should finish together by changing, if necessary, the speed of the first object. Hence, the scaling factor of `Video1` is changed accordingly (in this case it is doubled). After the completion of these two objects, the third one (a piece of text) is displayed for the predefined period. Note the way posted signals access and change the attributes of the media objects. Note also that any such changes are in effect from the *next* time instance.

`scenario_A` makes use of the agent `delay_by(Delay,Time_Signal)` which waits for a certain amount of time as specified by the parameter `Delay` before posting the signal `continue`. The time interval can be measured in terms of world time (by instantiating the second parameter to values such as `sec`, `min`, `hour` or suitable combinations) or some other internal to the application notion of time. It is assumed here that the underlying implementation supports the posting at every time instance of appropriate time signals with

the current (world) time. The following is a simplified implementation of `delay_by`. Note that the agent `delay_by(0,...)` never terminates and can be stopped only by enclosing it in some other suitable construct (such as a `do...watching` one).

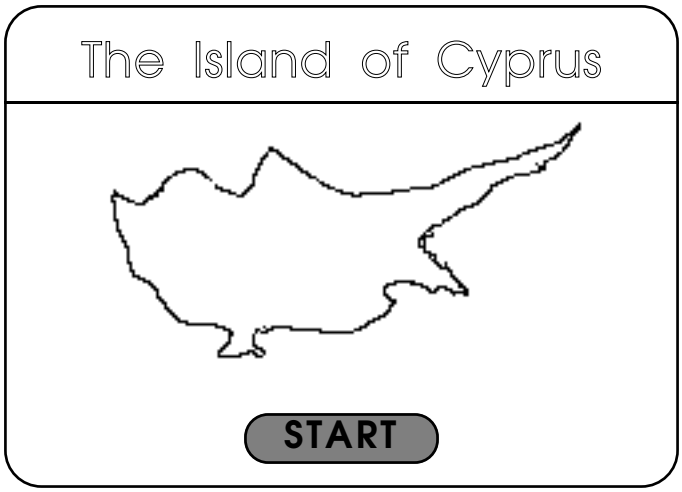
```
delay_by(Ticks,Time_Signal) :-
   whenever Time_Signal
      do (now Duration==1 then {continue} else (Ticks'=Ticks-1, self).
```

What is important to realise in the above example is that we have adopted a *temporal relation* rather than a *time-line* approach which preserves the references and links between the media objects involved if the values of any temporal (or spatial) parameters are changed. In addition, we have demonstrated that complex relationships between media objects can be expressed based not only on (world) time itself but also on the state of the media (whether they are playing or not, etc.) as well as the (non-deterministic) occurrence of some external event (eg. the pushing of some button). In the next section we show how a complete non-trivial multimedia application can be implemented in OO-TCCP.

5.2 The "information kiosk for Cyprus" example

We now show the implementation of a realistic multimedia application using OO-TCCP techniques. The example we have in mind is a variant of the *Virtual Museum* example ([9,10]) similar to ones described in [5,11]; 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 comprises 3 phases, each one of them being effectively a composite multimedia object.

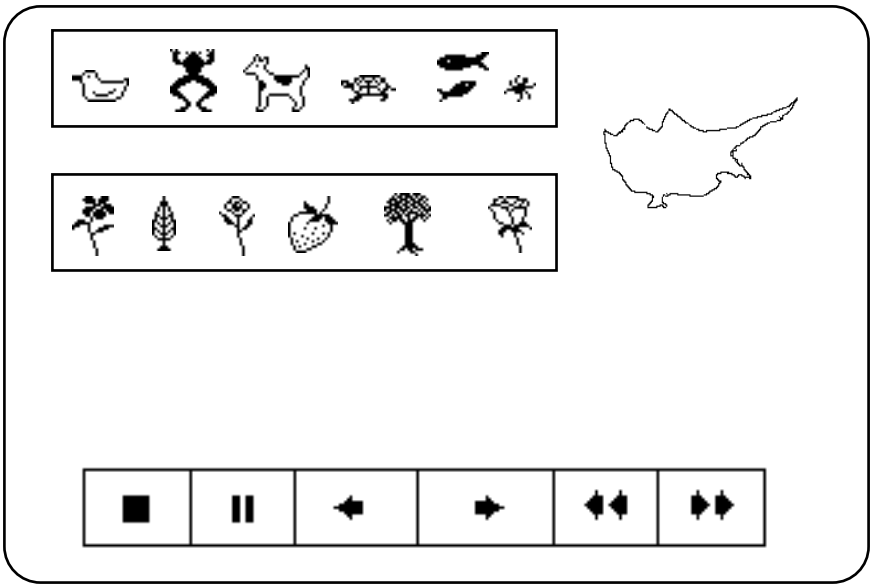
During the first phase the application displays the map of Cyprus along with an explanatory title and plays continuously an audio commentary inviting people to use it. Upon pressing the START button, the second phase begins (similar in functionality with the first one) where a menu is displayed informing the user of the different categories of information available, along with the playing of an explanatory audio.



Once the user has made a selection by pressing the appropriate button, the third phase commences showing 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 is shown. Finally, the user is able to pause and resume the presentation, play it in inverse direction, jump forward or rewind it, or end it at any moment in time. When the presentation ends the application loops back to the first phase.

In addition, we impose the following synchronisation requirements:

- If the user has not responded within one minute during the second phase, it is assumed that he has left and the application goes back to the first phase.
- If the application is suspended during the third phase, it resumes automatically in 30 seconds unless the RESUME key has been pressed earlier.
- Finally, during the third phase, audio and video are synchronised in the following sense: for every still image a corresponding audio commentary is played that starts 2 seconds after the display of the image.



The first two phases are very similar and can in fact be implemented using a generic composite object comprising three elementary ones: an image (the map of Cyprus or a menu list), an associated title (‘The Island of Cyprus’ or ‘MENU’) and an audio commentary. Again due to space limitations we often only sketch out the actual code.

```
phase_1_2(Image,Data_Image,Audio,Data_Audio,Text,Data_Text):-
   media_object(Image,Data_Image),
   media_object(Audio,Data_Audio),
   media_object(Text,Data_Text),
   scenario_phase_1_2(Image,Audio,Text).
```

For simplicity we assume that a suitable keyboard is used to implement the button functionality rather than a touch screen. In the latter case the implementation of a number of button objects would be needed which of course can also be done in OO-TCCP. Also, by

Data we mean both spatial and temporal ones (where each is applicable). A title's data for instance is just the coordinates where it should be displayed (and possibly the string itself) whereas an image's data can include also a scaling factor. The implementation of scenario_phase_1_2 is as follows.

```
scenario_phase_1_2(Image,Audio,Text) :-
   do ({Image:access, Audio:access, Text:access,
        Image:start, Audio:start, Text:start},
        always (now Audio:boolean:stopped
                then {Audio:access, Audio:start}))
   watching button_signal.
```

The agent scenario_phase_1_2 sends simultaneously a starting signal to all three objects. It then keeps checking continuously for the boolean signal Audio:boolean:stopped which indicates that the audio has finished playing in which case it restarts it. This block of commands is encapsulated within a do...watching construct which terminates the agent when a button signal is detected. To indicate that the Image and Text objects are to be displayed continuously we can use a special length value (say 0).

The next agent implements the third phase. It calls the function get_images_number which will post a signal Image:Number where Number is the number of images that will be presented. We assume here an equal number of audio commentaries.

```
phase_3(Image,Data_Image,Audio,Data_Audio,Delay,Time_Unit):-
   get_images_number(Image),
   whenever Image:Number
      do scenario_phase_3(Image,Data_Image,Audio,Data_Audio,
                          Delay,Time_Unit,1,Number).
```

The following auxilliary agent implements the presentation itself. Note the enhancement of the signals to the nth image with the corresponding index.

```
scenario_phase_3(Img,D_Img,Aud,D_Aud,Delay,TU,M,N) :-
   now M<=N
   then (media_object(Img:M,D_Img), media_object(Aud:M,D_Aud),
```

```
        Img:M:access, Img:M:start, delay_by(Delay,TU),
        whenever continue
           do (Aud:M:access, Aud:M:start,
               whenever Aud:boolean:stopped do (M'=M+1, next self))).
```

The agent CLOCK is interfacing the world-time with the application's time in order to freeze operations such as timeouts or delays when the the application has been suspended by the user. The constant values could be replaced with parameters to CLOCK if desired. CLOCK comprises four agents running concurrently: emit posts at every time instance the state of the application (running or suspended) which is changed by change_state depending on whether the RESUME button was pressed or the agent monitor_delay has indicated that the maximum allowed period of inactivity has been reached. Finally, tempo is responsible for keeping the application's tempo provided it is running.

```
clock(Delay,World_Time) :-
   emit(running), change_state, tempo(World_Time), monitor_delay(Delay).

emit(State) :- do always {State} watching state_changed.

change_state
   :- whenever (button_signal OR delay_out)
         do ( (now button:suspend
               then ({state_changed}, next emit(suspended))
               now (button:resume OR delay_out)
               then ({state_changed}, next emit(running))), next self).

tempo(World_Time) :- whenever World_Time
                        do ((now running then {appl_tempo}), next self).

monitor_delay(Delay) :- whenever suspended
                           do (do (delay_by(Delay,sec),
                                   whenever continue do {delay_out})
```

```
                watching running),
            whenever running do next self.
```

The top level agent `kiosk` is responsible for running all the agents and satisfying the 1 minute timeout requirement and the detection of other control signals (for reasons of brevity only the stop signal is shown here and the option signal is assumed to be the one for the flora and fauna).

```
kiosk :- do phase_1_2(map,…,welc_aud,…,appl_t,…) watching button_signal,
         whenever button:start
            do (do (phase_1_2(menu,…,aud_m,…,tit_m,…), delay_by(1,min))
                   watching (continue OR button_signal),
                   whenever continue do next self,
                   whenever button:option
                    do (do (phase_1_3(fl_faun,…,ff_aud,…,2,appl_tempo),
                            clock(30,sec))
                        watching button_signal,
                        whenever button:stop do next self)).
```

6 Conclusions and further work

We have presented an alternative approach to the issue of modelling and synchronising multimedia objects, that of using object-oriented 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 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 it is possible to combine the model with a hierarchical module system ([3]) where each object (simple or composite) communicates with its environment by means of a well defined signal interface.

We must stress here the fact that in this paper we focused our attention on showing the capabilities of OO-TCCP in modelling, composing and synchronising multimedia objects rather than presenting a specific multimedia development environment. In that sense a lot of work lies ahead. First of all OO-TCCP itself must be implemented and we are currently looking into this issue by examining how the temporal constructs of TCCP can be supported by the implementation of a specific concurrent logic language. In parallel, we are designing a specific object-oriented model and associated specification language drawing expertise from our Multimedia Laboratory equipment which includes state-of-the-art authoring tools such as AVC (Audio Visual Connection) and exploits Intel's DVI (Digital Video Interactive) technology. Further on, we are planning to adapt the model for distributed environments On a different front, we are investigating the enhancement of the model with *non-monotonic reasoning capabilities*, such as abduction with reactive capabilities ([12]), in order to develop intelligent and adaptable multimedia user interfaces. Finally, we are exploring the possibility of using constraint solvers, and in particular the ones for finite domains ([4]), to assist in specifying valid spatiotemporal constraint relations between media objects.

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 and partial financial support from the University of Cyprus, the Cyprus Popular Bank Ltd. and IBM SEMEA S.p.A, Cyprus Branch.

References

[1] 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.

[2] A. Davison, *"POLKA: A Parlog Object-oriented Language"*, Ph.D. Thesis, Department of Computing, Imperial College, London, May 1989.

[3] Y. Goldberg, W. Silverman and E. Y. Shapiro, "Logic Programs with Inheritance", *FGCS'92*, Tokyo, Japan, June 1-5, Vol. 2, pp. 951-960.

[4] P. V. Hentenryck, V. A. Saraswat and Y. Deville, "Constraint Processing in cc(FD)", Technical Report, Computer Science Department, Brown University, 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] T. D. C. Little, A. Ghafoor, C. Y. R. Chen, C. S. Chang and P. B. Berra, "Multimedia Synchronization", *IEEE Data Engineering Bulletin*, Vol. 14, No. 3, 1991, pp. 26-35.

[7] V. A. Saraswat, R. Jagadeesan and 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, 1994.

[8] R. Steinmetz, "Synchronization Properties in Multimedia Systems", *IEEE Journal on Selected Areas in Communications*, Vol. 8, No. 3, 1990, pp. 401-412.

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

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

[11] M. Vazirgiannis and C. Mourlas, "An Object-Oriented Model for Interactive Multimedia Presentations", *The Computer Journal*, Vol. 36, No 1., 1993, pp. 78-86.

[12] A. Wærn, "Weighted Abduction for Reactive Diagnosis", Research Report R92:11, SICS, Sweden, 1992.