

A Multimedia Programming Model Based on Timed Concurrent Constraint Programming

George A. Papadopoulos

Multimedia Research & Development Laboratory

Department of Computer Science

University of Cyprus

75 Kallipoleos Str.

Nicosia, CY-1678, P.O. Box 537

CYPRUS

e-mail: george@turing.cs.ucy.ac.cy

Abstract

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. 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. We then define a “universal” set of multimedia programming primitives and we show how they can be implemented in OO-TCCP. Finally, we argue about the benefits of our approach wrt both easiness of programming and implementation of the proposed framework. To the best of our knowledge this is the first time logic programming (in the form of concurrent constraint

programming or otherwise) is used as a basis for designing and implementing multimedia programming frameworks.

Keywords: Multimedia Programming; Declarative Programming; Modelling and Synchronisation of Multimedia Objects; (Applications of) Concurrent Constraint 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 modelling and synchronisation programming paradigms. In addition, the model used must preferably be amenable to formal analysis whose purpose is to reason about the “correctness” of a program; in this case by correctness we refer to the ability of the model to prove that the interaction between the media components of an application (such as in the case of synchronisation) is as expected. Such a formal reasoning capability is very useful in, say, deriving the quality of service (QOS) requirements of some application framework.

The purpose of this paper is to address the above issues; however, unlike most of the other approaches that are primarily based on imperative programming techniques and the use of languages such as C++ ([21,22]) or real-time ones ([2]) such as ESTEREL ([8,15]), 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 ([18]) of concurrent constraint programming ([17]) 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 ([4,6,12]). This work should be seen as a 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 and argue about the benefits of our approach. This is followed by a brief introduction to OO-TCCP, a combination of timed concurrent constraint programming ([18]) with the object-oriented ([4,12]) and inheritance facilities ([6]) that have been developed for logic programs. The fifth section presents a “universal” set of primitives for multimedia programming and discusses their implementation in OO-TCCP, thus showing the capabilities of the model in handling the requirements imposed by multimedia object modelling and synchronisation. The following section discusses briefly implementation techniques for the proposed framework. 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 all programmers are not necessarily familiar. 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 ([5]) 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 media objects are usually composed to form composite multimedia objects. Our object-oriented framework is a variant of Gibb's "active objects" metaphor ([22]) and is 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.

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) 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 ideally be based on rigorous operational semantics allowing formal reasoning of the written programs. A number of proposals have been put forward ranging from process calculi such as CSP ([20]) to extending Petri Nets with temporal properties ([13,16]). 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 [8] or [15]) advocating principles from real-time systems and in particular the so called *perfect synchrony*

hypothesis ([2]) 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.

There are two points worth mentioning about the above mentioned approaches: (i) the perfect synchrony hypothesis, although very convenient to depend upon when real-time constraints must be imposed, requires the use of special languages as well as sufficient support by the underlying architecture; (ii) some of the formal models proposed ([13,16,20]), although are expressive enough to cover most of the temporal relationships between media objects, are however not always very intuitive and easy to handle by programmers who are not sufficiently experienced with formal specification techniques. The present work tries, among other things, to address these issues by proposing the use of a framework for multimedia development which is both intuitive in use but also easier to implement in that it does not rely on the use of a specific real-time language or special features of some underlying architecture.

4. Declarative Object-Oriented Real-Time Programming

4.1 Timed concurrent constraint programming

Timed concurrent constraint programming (TCCP), developed by Saraswat *et al.* ([18]), is an extension of concurrent constraint programming ([17]), itself being a combination of constraint logic programming ([10]) and concurrent logic programming ([19]), with temporal capabilities along the lines of state-of-the-art real-time languages such as ESTEREL, LUSTRE and SIGNAL ([2]), 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* ([18]) 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.

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. We will come back to this issue in the sixth section where we discuss the implementation of the proposed framework.

In TCCP an agent A takes one of the following forms.

$A ::= c$	(post constraint c to the current store)
now c then A	(if the current store entails c then behave like A)
now c else A	(if c is entailed then behave like A in the <i>next</i> time instance)
next A	(behave like A in the next time instance - unit delay)
abort	(abort computation)
skip	(skip)
A, A	(parallel composition)
$p(t_1, \dots, t_n)$	(procedure call)

Thus, 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 [18], 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,

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 (note here that **{signal}** means emit **signal**). 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 ([4,12]). 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 objects forming collectively composite ones by means of delegating messages to the appropriate object.

In addition, a number of techniques ([6,12]) 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 [6], adapted to satisfy the constraints imposed by TCCP, to form what is referred to in this paper as *object-oriented timed concurrent constraint programming* (OO-TCCP). The main reason why we chose the framework proposed in [6] to enhance TCCP with object-orientation is that it is essentially a textual sugaring for TCCP programs (inheritance, for instance, is based on copy semantics). Other, admittedly more powerful proposals, usually involve some, even mild, semantic extensions to the underlying language formalism (for instance, the way concurrent method invocations are handled and how they interfere within stateful objects, etc.). We did not want to introduce such powerful formalisms before understanding fully any influence these may have on the temporal constructs of the framework. Therefore, OO-TCCP is only a *notational extension* to TCCP, but one we feel yields a more convenient as well as powerful programming language.

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).
```

where ‘;’ is the (committed) OR-parallel operator and ‘+’ is the inheritance operator (based on copy semantics). The above code fragment is equivalent to the following expanded one.

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

Furthermore, 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; it is a syntactic convention that these variables are the new parameters passed to **self**. In other words **self** expands to **counter(Counter, Value1)** as in the previous (original) TCCP version. Thus OO-TCCP programs become less verbose than TCCP ones without creating any complications arising from the introduced object-oriented functionality.

Along the above lines, it is further possible to introduce additional notations, especially for operations that are being performed all the time. For instance, we could have

c ? A ! B as a shorthand for **now c then A else B**

in which case our **counter** example would take the form:

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

and so on. But we refrain from introducing here a complete set of such shortcut notations so that we do not hide the way the barebone model operates. Note, of course, that by a preprocessing step OO-TCCP compiles down to TCCP.

4.3 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 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 (eg. by using nested **now...then** constructs). It is also inherently non-deterministic, thus being able to capture unpredictable functionalities imposed by a user such as the pushing of some button and browsing or posing queries. 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.

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 **C_i** ([21]):

$$| \mathbf{C}.\mathbf{current_world_time} - \mathbf{C}_i.\mathbf{current_world_time} | < \Delta_i$$

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

$$| \mathbf{ObjToWorld}(\mathbf{M1}.\mathbf{object_time}) - \mathbf{ObjToWorld}(\mathbf{M2}.\mathbf{object_time}) | < \Delta$$

where the function **ObjToWorld** translates an object's relative (internal) time to world (the application's) time. Thus the synchronisation tolerance Δ_i can be thought of as the very short amount of time that a framework like OO-TCCP would take to reply to some signal.

Furthermore, the timed asynchrony hypothesis offers more choices regarding implementation of the proposed framework. We further explore this issue in the sixth section. The next section however is devoted to establishing the basis for building a multimedia programming environment using OO-TCCP.

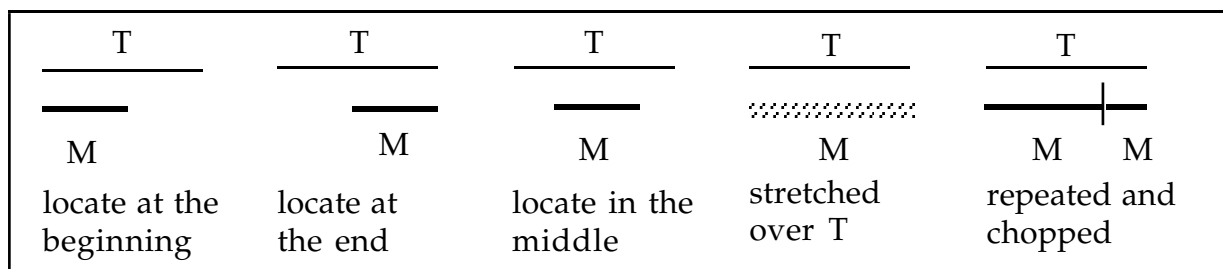
5. Multimedia Programming in OO-TCCP

5.1 A “Universal” Set of Multimedia Primitives

A (composite) multimedia object comprises a set of independent and more elementary media objects such as text, video, sound, etc. Programming in a multimedia environment requires dealing with the spatio-temporal requirements of the multimedia objects comprising a certain application: temporal relations or constraints between the media objects involved must be expressed so that the required intra and inter media synchronisation will be achieved; also displayable media information must also be composed in space.

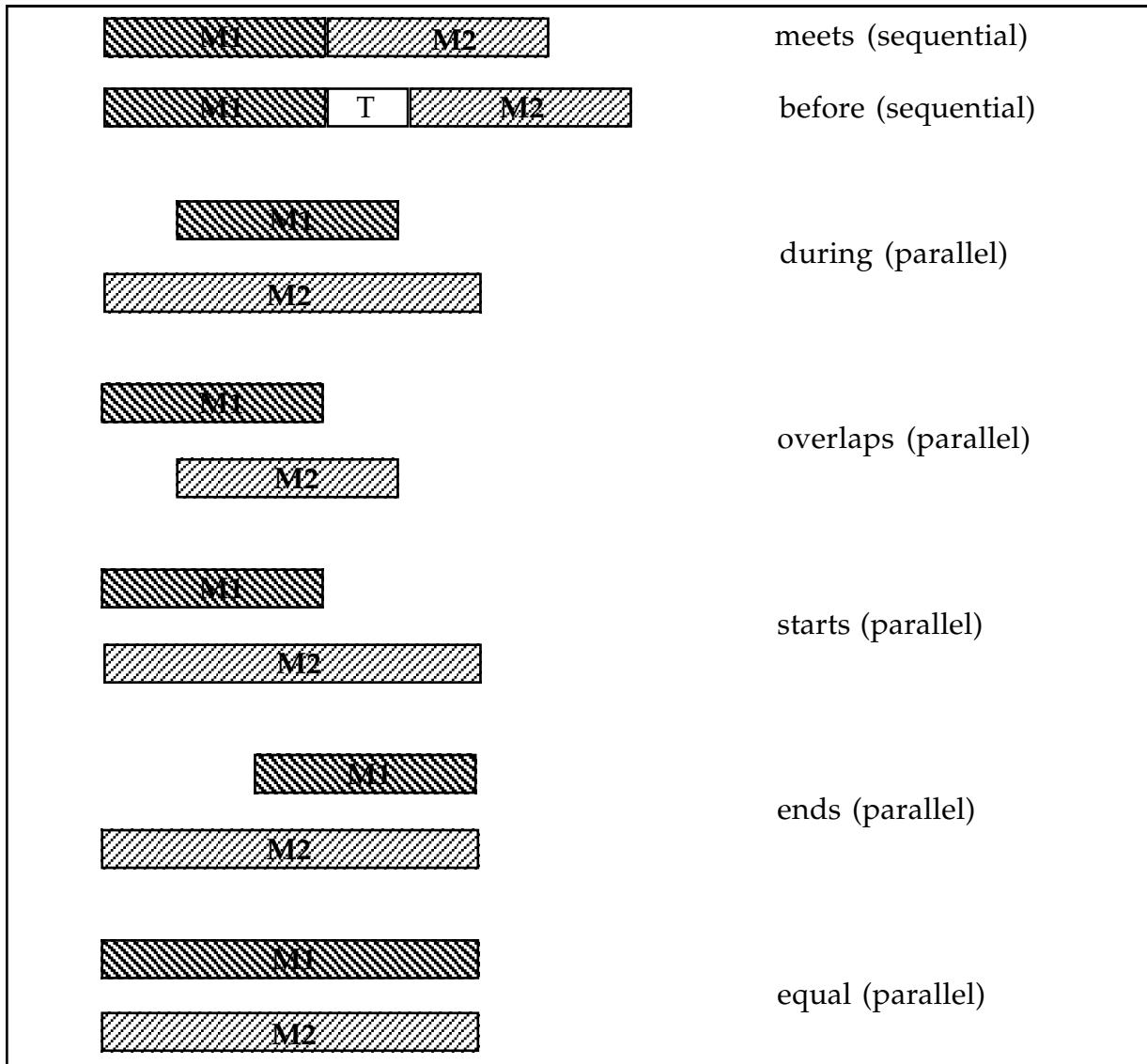
We are particularly interested in the time-based media objects ([5]) requiring the concept of time. We note that many authoring environments are offering a *timeline* metaphor where actions on the media involved are associated with points in time. The main problem with this approach is that once a media object is moved in time (but also in space) all logical references and links with the other objects are lost. In this paper we advocate the alternative approach where the media involved in an application are linked by means of (bidirectional) constraints. Thus when some temporal attribute of an object changes all associated references are automatically updated. Note that the constraint based representation of media relationships does not preclude it being mapped internally to a timeline framework.

In order to derive a useful set of multimedia primitives able to model the temporal behaviour of a media object we must first note that it is possible to define the following temporal relationships between a media object M and some time period T.



We can also note that a number of temporal relationships are possible between two media objects M1 and M2 ([20]) as shown in the figure below. Note that the first two relations are

sequential whereas the rest are parallel (both media objects are active at the same time). In the latter case, the duration of the composite (multi) media object is that of the longest one of its constituents.



We can thus conclude that the time instance when a media object starts or stops playing can depend upon:

- actions on other media (“play M2 when M1 finishes”),
- a time value (“play M1 at 5 secs after the user starts the application”),
- an external event (the pressing of some button).

We can thus derive a set of “universal” primitives which can be used as the basis for building a multimedia programming environment. By *universal* we mean that the actual set of primitives

offered by some particular programming environment may be slightly different from the one presented here but the overall functionality offered should be more or less the same. Also, if OO-TCCP can model the set of primitives described in this paper it should be able to model any other similar set.

We could organise the primitives into the following categories:

- **get** functions that ask for the value of some object's primitives (eg. **get_video_length**, **get_audio_volume**);
- an associated set of **set** functions that change the value of some object's primitives (eg. **set_scaling_factor**);
- a set of functions to program the synchronisation between the, possibly concurrently, executing objects.

Regarding this last set of primitive functions and based on the aforementioned analysis on the inter and intra relationships involving media objects and time we can derive the following set of primitives ([3]) where **M** denotes a media object, **MM** a composite object, **C** a condition, **T** a time period, and **E** an external event.

MM=Sequential(MM1, M2)	start M2 only when MM1 has stopped playing
MM=Parallel(MM1, M2)	play MM1 and M2 in parallel
Play(MM, C)	start playing MM when C is satisfied
Stop(MM, C)	stop playing MM when C is satisfied
C=Started(MM, T)	MM has been playing for T time (maybe 0)
C=Stopped(MM, T)	MM has stopped playing for T time (maybe 0)
C=Playing(MM)	MM is still playing
C=NotPlaying(MM)	MM is not playing
C=Time(hh:mm:ss)	world time is now hh:mm:ss
C=Event(E)	event E has taken place
C=OR(C1, ..., Cn)	either of C1 to Cn has been satisfied
C=AND(C1, ..., Cn)	all C1 to Cn have been satisfied

In order to show how the above sets of primitives are modelled in OO-TCCP we define in the next section an object-oriented multimedia model which is also formulated in OO-TCCP itself.

5.2 A multimedia object model

We show here the basics of defining media objects in OO-TCCP. To keep the length of the paper within reasonable limits we refrain from giving a complete breakdown of all the features associated with each one of the types of media objects available; instead, we concentrate on the principles of building such a framework which can be classified as follows: i) define an object with a set of associated attributes, ii) define a communication interface between the object and the outside world by means of signals, iii) exploit the programming features available in OO-TCCP. Note that what is noteworthy about the code below is not so much its complexity but rather the fact that within a single declarative framework we can express modelling, composition and synchronisation requirements for time-based multimedia objects.

Adherence to all the principles just described is illustrated in the modelling of perhaps the two most important types of media, video and sound. We identify a common basic media object with some characteristic attributes ([5]).

```
time_media_object(Object)+(QualityFactor,Duration,Encoding,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 (eg. 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,Encoding,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,Encoding,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.

We recall that there are no rigid variables in TCCP and any bindings to be retained must be posted at every time instance. Therefore in all cases where signal communication is performed we assume that the requesting agent can detect the posted signal in the same time instance; if that is not possible the signal 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).

5.3 Implementing the Primitives in OO-TCCP

Having defined an object model and an associated interface between concurrently executing objects we can now show the formulation of the rest of the primitives, namely the programming primitives, in OO-TCCP. For those primitives taking a variable number of media objects as parameters we show the implementation with two such arguments.

However we should clear a point at this stage. We expect the underlying implementation to be able to support the generation of *boolean* signals describing the current state of some object (such as **started**, **stopped**, **playing**, etc.). We can then have two objects: **ack_mess** is responsible for handling these messages, which incidentally may be generated in various formats by the different devices involved, and **post_object_status** communicates these messages to the rest of the program using a standard format. A possible, simplified, implementation of these two objects is the following one.

```
ack_mess :- whenever Device:Object:BoolSig do ({ change_state:Object:BoolSig}, next self).
```

```
post_object_status (Object, BoolValue) :-
```

```
  do always { Object:BoolValue} watching change_state:Object:BoolSig,
  whenever change_state:Object:BoolSig do (BoolValue'=Object:BoolSig, next self).
```

Note that **post_object_status** effectively implements the conditions **Playing** and **NotPlaying**. In the same way, we can define the conditions **Time** and **Event** as being signals generated by corresponding devices (such as the clock), and we do not show their implementation here.

Now we turn our attention to the primitives described in section 5.1. In general, the interface of a primitive is represented in OO-TCCP as follows.

MM=Prim(T1,...,Tn)	translates in OO-TCCP to	Prim(MM)+(T1,...,Tn)
C=Prim(T1,...,Tn)	translates in OO-TCCP to	Prim(C,T1,...,Tn)
Prim(T1,...,Tn)	remains the same, i.e.	Prim(T1,...,Tn)

where **MM** represents a composite media object, **C** a condition, **Prim** the name of a primitive and **T1,...,Tn** its parameters. For the first case, note that **MM** is effectively the composite object's communication channel with the environment. So, starting with the sequential composite object we have:

```
sequential(MM)+(M1,M2) :-
  whenever MM:access
    do ( now MM:play then ({ M1:access, M1:play}, next self),
      now MM:stop then ({ M1:access, M1:stop, M2:access, M2:stop}, next self)),
  whenever M1:stopped do ({ M2:access, M2:play}, next self),
  whenever M2:started do ({ M1:access, M1:stop}, next self),
  whenever M2:stopped do ({ MM:stopped}).
```

We should note here the bidirectionality of the timed constraints. For instance, if for some reason, external to the **sequential** object, **M2** starts playing even before **M1** has stopped then according to the intended semantics of the sequential primitive **M1** should stop playing. This is achieved in the one but last line of the code. The parallel composite object is defined as follows.

```
parallel(MM)+(M1,M2,Counter=2) :-
  whenever MM:access
    do ( now MM:play then ({ M1:access, M1:play, M2:access, M2:play}, self),
      now MM:stop then ({ M1:access, M1:stop, M2:access, M2:stop}, next self)),
  whenever M1:stopped do (Counter'=Counter-1, next self),
  whenever M2:stopped do (Counter'=Counter-1, next self),
  whenever Counter==0 do { MM:stopped}.
```

Note that the sequential and parallel primitives subsume those proposed by the MHEG group as standards ([9]). The play and stop primitives are defined as follows.

play(MM,C) :- whenever C:true do {MM:access, MM:play}.

stop(MM,C) :- whenever C:true do {MM:access, MM:stop}.

The conditions that have not been tackled so far are implemented as follows.

**started(C,MM,Delay) :- now MM:playing then delay_by(NC,Delay),
whenever NC:continue do {C:true}.**

**stopped(C,MM,Delay) :- now MM:stopped then delay_by(NC,Delay),
whenever NC:continue do {C:true}.**

Note the use of the “private” signal ids **C** and **NC** to distinguish between similar signals posted on behalf of different objects. Both primitives make use of the agent **delay_by** defined as follows.

delay_by(C,Ticks) :- do (now Ticks==0 then {C:continue} else (Ticks'=Ticks-1, self)).

Actually this is a simplified version; in reality we would have additional code to distinguish between the different notions of **Ticks** (seconds, minutes, etc.).

We end this section with the implementation of the AND and OR conditions.

**and(C,C1,C2) :- whenever (C1:true,C2:true) do {C:true},
whenever (C1:false;C2:false) do {C:false}.**

**or(C,C1,C2) :- whenever (C1:true;C2:true) do {C:true},
whenever (C1:false,C2:false) do {C:false}.**

where we make use of the ‘;’ (conjunctive) and ‘;’ (disjunctive) constraint operators available in OO-TCCP.

5.4 Examples

We end this part of the paper with a few examples. The first one describes the following behaviour of a composite object comprising two text objects, a video object and an audio

object: play in parallel **Text1** and **Video** and when event **Event** occurs or time **Time** has been reached commence the (sequential) play of **Text2** followed by **Audio**.

```
script1(Scr)+(Text1, Text2, Video, Audio) :-
    text_object(Text1,...), text_object(Text2,...),
    video_object(Video,...), audio_object(Audio,...),
    scenario1(Scr)+(Text1, Text2, Video, Audio).
```

```
scenario1(Scr)+(Text1, Text2, Video, Audio) :-
    whenever Scr:access
    do ( now Scr:start
        then (parallel(P, Text1, Video), {P:access, P:play}, and(C, Event, Time),
            whenever C do (sequential(S, Text2, Audio), {S:access, S:play}),
            whenever S:stopped do {Scr:stopped}),
        now Scr:stop then {P:stop, S:stop}
    ).
```

The next example implements the following scenario: Show **Text** for 10 seconds and **Picture** for 20 seconds. Provided **Text** is still displayed, a user's event **E** may start the play of **Audio1** having 5 seconds duration followed by the sequential playing of **Video** and **Audio2**.

```
script2(Scr)+(Text, Picture, Video, Audio1, Audio2) :-
    text_object(Text,...,10,...), graphics_object(Picture,...,20,...),
    audio_object(Audio1,...,5,...), audio_object(Audio2,...), video_object(Video,...),
    scenario2(Scr)+(Text, Picture, Video, Audio1, Audio2).
```

```
scenario2(Scr)+(Text, Picture, Video, Audio1, Audio2) :-
    whenever Scr:access
    do ( now Scr:start
        then (parallel(P, Text, Picture), {P:access, P:play},
            and(C1, playing(Text), E), play(Audio1, C1),
            whenever Audio1:stopped
            do sequential(S, Video, Audio2), {S:access, S:play}),
        now Scr:stop then {P:stop, S:stop}
    ).
```

The final example 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 and adhering to the

following scenario: After a delay of 30 seconds play together **Video** and **Audio** which should finish together by changing, if necessary, the speed of the first object. Hence, the scaling factor of **Video** 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.

script3(Scr)+(Text,Video,Audio) :-

```
text_object(Text,...,3,...), video_object(Video,...,10,...), audio_object(Audio,...,5,...),
scenario3(Scr)+(Text,Video,Audio).
```

scenario3(Scr)+(Text,Video,Audio) :-

```
whenever Scr:access
do ( now Scr:start
then ( delay_by(C,30),
whenever C: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 scenario3_cont(Scr,Text,Video,Audio))),
now Scr:stop then { Scr:stopped}
).
```

scenario3_cont(Scr)+(Text,Video,Audio) :-

```
parallel(P,Video,Audio), { P:access, P:play}, play(Text,P:stopped),
whenever Scr:access do now Scr:stop then ({ P:access, P:stop, Text:access, Text:stop}).
```

What is important to realise in the above examples is that we have adopted a *temporal relation* rather than a *timeline* approach which preserves the references and links between the media objects involved even 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).

6. Implementing OO-TCCP

In this section we briefly touch upon the issue of implementing OO-TCCP. We recall that the timed asynchrony hypothesis allows programs to express real-time behaviour in the sense that response to some signal can be guaranteed to be done in a very short period of time (even if not instantaneously). The additional advantage of this framework is that its basic principles can be used to enhance any asynchronous computational model without the need to resort to special hardware features or specific programming languages. We are currently building an implementation of OO-TCCP ([14]) on top of the distributed coordination model MANIFOLD ([1]), an event-based coordination language. OO-TCCP primitives, like the ones described in this paper are compiled down to equivalent MANIFOLD programs. Signal sending and receiving is implemented directly by means of the event mechanism available in MANIFOLD. The processes constituting a MANIFOLD program are monitored by a “clock” process which is responsible for both detecting the end of the current time instance and triggering the next one. Actually, this is done in a distributed fashion. The actual media devices that an OO-TCCP program handles are viewed in MANIFOLD as atomic processes whose internal behaviour is not (and should not be) known to the rest of the program and whose operations are “coordinated” by the MANIFOLD program ([15]). Right now programs are manually translated from OO-TCCP to MANIFOLD although eventually the process will be done automatically.

7. Conclusions 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. 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 programming frameworks (however, this has already been done in the related field of animation ([11])). 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 ([6]) where each object (simple or composite) communicates with its environment by means of a well defined signal interface. Finally, OO-TCCP offers a formal framework which is not only more intuitive than others proposed for developing time-based multimedia environments but also it is easier to implement requiring no special hardware support or the use of specific (real-time) languages.

Our approach is somewhere between those using an existing imperative real-time language ([8,15]) and those advocating a more formal and theoretical model ([13,16,20]) in the sense that OO-TCCP has sound formal semantics but it is also by itself a programming formalism; thus, we manage to combine the advantages of both these approaches. We believe that OO-TCCP offers an elegant and attractive alternative approach to designing conceptual models able to capture the spatial and temporal functionality of multimedia environments and offer suitable abstractions where manipulation and control of a presentation is achieved by means of reference modifications. In addition, the high-level nature of the model renders it independent of any particular application environment. Also, OO-TCCP can be used as a synchronisation model for both the *presentation* and the *computation* part of a multimedia application ([13]).

A number of issues are currently under investigation. The implementation of OO-TCCP itself is under way; as we mentioned in the previous section this is a distributed one based on a specific asynchronous coordination model. On a different front, we are investigating the enhancement of the model with *non-monotonic reasoning capabilities*, such as reactive abduction ([23]), 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 ([7]), to assist in specifying valid spatio-temporal 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.

We would also like to thank the referees for many constructive comments which improved the presentation of the paper.

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, pp. 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, pp. 11-17.
- [3] M. Bordegoni, *Multimedia in Views*, Internal Report CS-R9263, CWI, Amsterdam, The Netherlands, 1992.
- [4] A. Davison, *POLKA: A Parlog Object-oriented Language*, Ph.D. Thesis, Department of Computing, Imperial College, London, May 1989.
- [5] S. Gibbs, C. Breiteneder and D. Tschritzis, Data Modelling of Time-Based Media, *SIGMOD'94*, Mineapolis, Minnesota, USA, May, 1994, ACM Press, pp. 91-102.
- [6] Y. Goldberg, W. Silverman and E. Y. Shapiro, Logic Programs with Inheritance, *FGCS'92*, Tokyo, Japan, June 1-5, Vol. 2, pp. 951-960.
- [7] P. V. Hentenryck, V. A. Saraswat and Y. Deville, *Constraint Processing in cc(FD)*, Technical Report, Computer Science Department, Brown University, 1992.
- [8] F. Horn and J. B. Stefani, On Programming and Supporting Multimedia Object Synchronisation, *The Computer Journal*, Vol. 36, No 1., 1993, pp. 4-18.

- [9] 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.
- [10] J. Jaffar and J-L. Lassez, Constraint Logic Programming, *14th ACM Symposium on Principles of Programming Languages*, Munich, Germany, January 1987, pp. 111-119.
- [11] K. M. Kahn, Concurrent Constraint Programs to Parse and Animate Pictures of Concurrent Constraint Programs, *FGCS'92*, Tokyo, Japan, June 1-5, 1992, Vol. 2, pp. 943-950.
- [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] 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.
- [14] G. A. Papadopoulos and F. Arbab, Coordination of Systems with Real-Time Properties in MANIFOLD, *COMPSAC'96*, Seoul, Korea, August 21-23, 1996, IEEE Press (to appear).
- [15] M. Papathomas, G. S. Blair and 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, July 5, 1994, LNCS 924, Springer Verlag, 1995, pp. 162-175.
- [16] B. Prabhakaran and S. V. Raghavan, Synchronisation Models for Multimedia Presentation with User Participation, *Multimedia Systems*, Vol. 2, 1994, pp. 53-62.
- [17] V. A. Saraswat, *Concurrent Constraint Programming*, Ph.D. Thesis, Carnegie-Mellon University, January 1989, appeared as ACM Doctoral Dissertation Award, MIT Press series on Logic Programming, 1993.

- [18] 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, LNCS, Springer Verlag, 1994.
- [19] E. Y. Shapiro, The Family of Concurrent Logic Programming Languages, *Computing Surveys*, Vol. 21 (3), 1989, pp. 412-510.
- [20] R. Steinmetz, Synchronization Properties in Multimedia Systems, *IEEE Journal on Selected Areas in Communications*, Vol. 8, No. 3, 1990, pp. 401-412.
- [21] D. Tschritzis (ed.), *Object Frameworks*, Internal Report (collected papers), Centre Universitaire d'Informatique, Université de Genève, Switzerland, 1992.
- [22] D. Tschritzis (ed.), *Visual Objects*, Internal Report (collected papers), Centre Universitaire d'Informatique, Université de Genève, Switzerland, 1992.
- [23] A. Wærn, *Weighted Abduction for Reactive Diagnosis*, Research Report R92:11, SICS, Sweden, 1992.