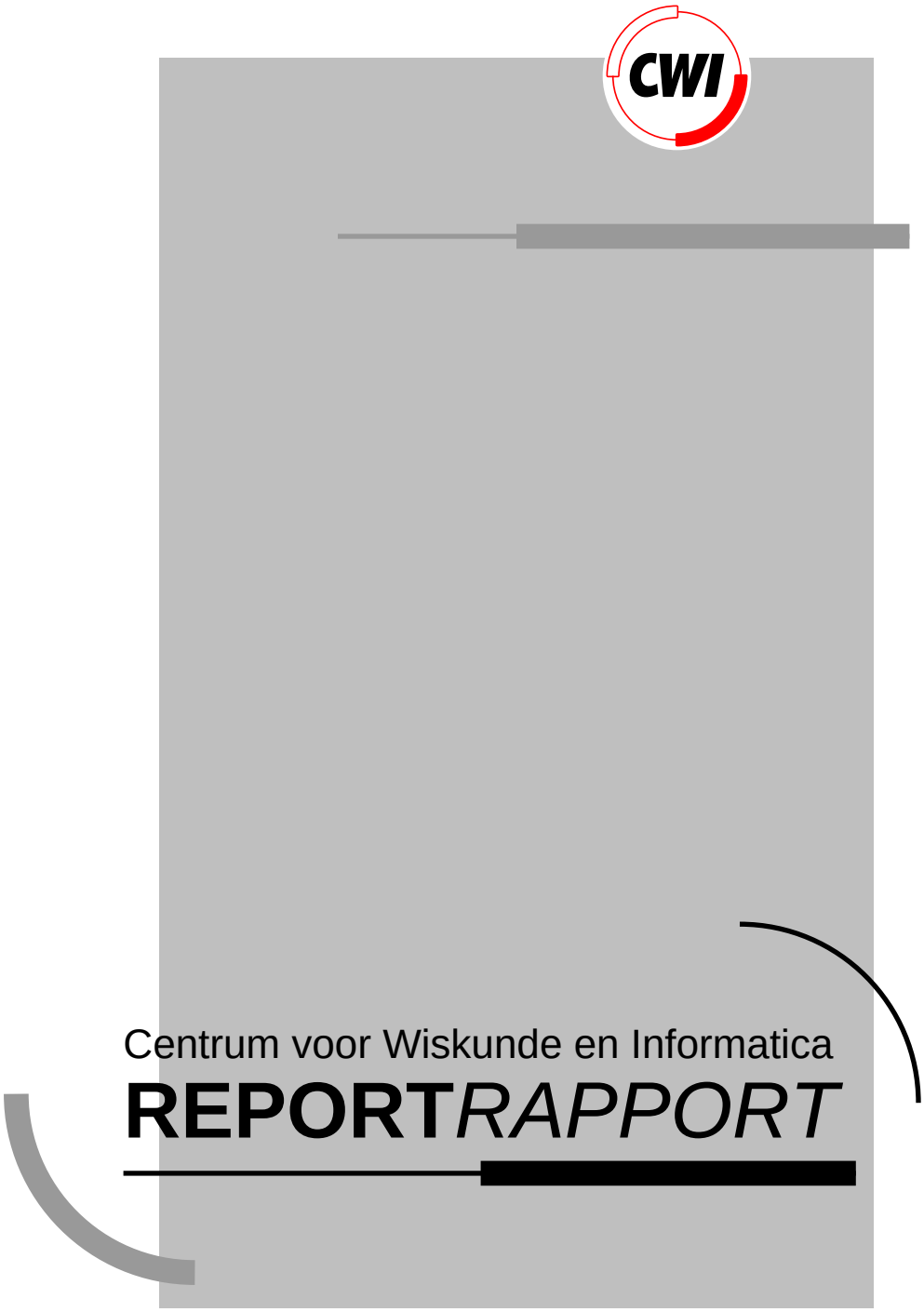

Centrum voor Wiskunde en Informatica

REPORT*RAPPORT*

An Object Model for Multimedia Programming

F. Arbab, I. Herman, G.J. Reynolds

Computer Science/Department of Interactive Systems

CS-R9327 1993

An Object Model for Multimedia Programming

F. Arbab, I. Herman, G.J. Reynolds

CWI

P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

Abstract

The development of multimedia applications is a complex task. Much of this complexity stems from requirements associated with programming multiple media objects and the control of dependences and inter-relationships between these media objects and the user(s). It is clearly necessary to have a basic framework on which to build multimedia applications in the face of such complexities. Such a conceptual model is what we have called an object model and it is the main subject of this paper. The MADE object model represents a novel approach to multimedia application programming that is founded on the two principal concepts of *active objects* and *delegation*. Although these concepts are not novel in themselves, we believe that their combined use in a multimedia development environment represents a substantial enhancement to more traditional approaches to programming in this area.

AMS Subject Classification (1991): 68N15
CR Subject Classification (1991): D.1.5,D.2.2,H.5.1,I.3.2
Keywords & Phrases: Multimedia programming, object models, active objects, concurrency, delegation.
Note: This paper will be published in the journal "Computer Graphics Forum", **12**(3) (Eurographics'93 Conference Issue).

1. Introduction

One of the most spectacular developments in computing technology of the past few years is the appearance of multimedia. Glossy multimedia applications are shown all over the place and on a wide range of different platforms. All major workstation hardware vendors feel the need to come to technical fairs with stunning demonstrations mixing graphics, video, imaging, and sound. Technical analysts predict that multimedia related hardware development will be one of the booming areas of electronics in the years to come.

However, if we look behind the shiny façades of these applications, we often get a confusing and somewhat disappointing impression. The fact is that most of these programs rely very heavily on the underlying hardware, they are very often highly non-portable, and they lack a unifying programming concept. The situation is reminiscent of the early period of computer graphics, when hardware suppliers invaded the marketplace with mutually incompatible graphics devices, and graphics applications were very much dependent on the computer graphics hardware environment they were developed for.

Just as it was the case for more traditional computer graphics 15 years ago, it is only after several years of development that multimedia research can enter into a more mature phase. Appropriate programming concepts and models must be found and fleshed out in order to have more portable multimedia applications. On a longer term, suitable official and/or de facto standards need to be developed and adopted by the programming community. This fact has been recognised recently by ISO, too; at a meeting in early autumn 1992, at Chiemsee, Germany, an agreement was reached concerning the development of a new generation of graphics standards that will fulfill the needs of multimedia as well. In other ISO standards committees, work on multimedia document standards has been under way for some time. The SGML extension known as HyTime [1] and the ODA extension [2]

are concerned with both multimedia and hypermedia document descriptions. Note that neither of these standards activities concentrate on presentation mechanisms.

The European Communities' ESPRIT III project MADE (Multimedia Application Development Environment [3]) has recently set up the ambitious goal of defining and implementing a portable object-oriented development environment for multimedia applications. The particularity of this project is that it does not aim only at the definition of programming tools to handle, e.g., video and audio, but it also identifies a number of more general objects (i.e., tools) that are necessary to develop advanced multimedia applications. As such, the object base to be developed for MADE contains not only 2D and 3D graphics, audio, video, animation etc., but also distinct interaction objects (for advanced user interaction facilities), time or synchronisation objects (for media synchronisation), constraint objects (to allow for sophisticated constraint management), help, monitoring, authoring, etc. The first real challenge of the project, therefore, has been the definition of a programming model which will allow for the proper use of all these different functional capabilities. (For the sake of brevity, we will refer to the MADE Object Model as "MOM" in the rest of this paper.)

At the earliest stages of the MADE project specification it became clear that a concise framework, i.e., a clear *object model* is highly desirable in order to give a comprehensive model for multimedia programming. It was also recognised that, in fact, a completely new object model had to be *defined*: the models promoted by well-known tools like, e.g., C++ [4], simply did not fully satisfy the requirements of multimedia programming. Out of a series of passionate discussions and pilot development activities at our institute, a new object model finally emerged, which we believe to be an appropriate framework for the tasks the project has set up to fulfill. The MADE Object Model, "MOM", is described in the following sections. MOM has a much wider interest than the MADE project alone; it can lead to new ideas for multimedia programming in general, and it has provided valuable input to the ISO standardisation activities we referred to [5, 6].

This paper is organised as follows. In §2 the general object model is described. The model is centred around two cardinal concepts: the concept of active objects (described in §2.1) and delegation (described in §2.2). A larger example, addressing the major concepts of our model, is presented in §3. We give some hints at the implementation of our concepts in §4, and, in §5 we describe briefly the structuring facilities provided by the MADE environment.

2. The Object Model

Advanced multimedia programming raises a number of non-trivial problems. Apart from the obvious hardware challenges (see, e.g., [7]), certain demands of multimedia are also difficult to honour within the framework of traditional operating systems, e.g., Unix™ (see [8]). Our concerns are, however, different. Namely, provided that the underlying hardware and operating system layers are appropriate, what is the ideal programming framework, i.e., what is a suitable "mental model" for multimedia programming? As in models for other complex applications, the notion of an object is an important means of abstraction in this framework. However, just to say "our framework must be object-oriented" is not enough.

We propose two central concepts that we think should be part of a proper object model for multimedia. These are *active objects* and *delegation*. These are elaborated on in the following sections.

2.1 Active Objects

Synchronisation of different media is known to be a particularly demanding task. Conceptually, different media (i.e., a video sequence and a corresponding sound track) should be considered as parallel activities that have to reach specific milestones at distinct and possibly user definable synchronisation points. In many cases, specific media types may be directly supported in hardware. In some cases, using strictly specified synchronisation schemes, the underlying hardware can take care of synchroni-

sation. However, a general object model aimed at multimedia should offer the possibility to describe synchronisation in general terms as well. This requirement directly leads to the adoption of *active objects* as one of the cardinal concepts in MOM.

2.1.1 Definition of Active Objects All objects in MOM are *active objects*. This means that, conceptually, every object has its own virtual processor with its own thread of control. Active objects communicate with one another by sending *synchronous messages* which are *served* by the callee.

From a callers' point of view, there is only one form of message communication, namely *synchronous message passing*. However, *internally*, a callee has finer control over the acceptance of messages and can use two different forms of message reception regimes. Each method in an object has its own dedicated *message receptor* of which there are two kinds. Messages arriving on a queued message receptor cause the caller to be suspended until the message has been serviced. Messages sent to a sampled message receptor effectively represent an event and do not cause the caller to be suspended, except during the communication of the message. The choice of message receptor type for a method is private and internal to the receiving object, and other objects in the MADE system have no knowledge of it (except perhaps by empirical means).

More precisely, a callee, `A`, has two alternative behaviours for handling the case where several active objects make calls to one of its methods.

1. Calls are *queued*. This means that the messages invoking the method arrive in the method's designated message queue.

2. Calls are *sampled*. This is equivalent to having a single entry queue as the message receptor for the method; in this case a newly arriving message replaces any current message in the queue. Messages of this type cannot have any additional information associated with them (such as parameters or sender identification) and they cannot have return values. Message passing to a sampled message receptor returns at once to the caller, just as if the message had been served immediately by `A`. For example, the caller will not be suspended even if the callee was busy with some other computation.

Apart from having several outstanding message requests on the same method declared for an object, there may also be several different kinds of message requests waiting to be served. To handle these, methods are classified into priority classes. If several different messages to an object are pending, the object will choose a message for a method of a higher priority over lower priority ones; if there are several messages pending on the receptors of methods of the same priority, the object will choose one of these messages non-deterministically.

An active object has the internal means to suspend itself until it has a message to serve. It also has the means to define which messages it wishes to serve, depending on its internal state.

Our specification of active objects is fairly traditional. The advent of popular "object-oriented" paradigms like C++, often makes it necessary to remind their users that one of the archetypes of object oriented programming, Smalltalk-80 [9], makes use, at least conceptually, of active objects. Our definition follows the same ideas, using paradigms for message passing and for the control of received messages that are well known in the concurrent programming community.

The only unusual feature of our model, compared to traditional message passing protocols, is the introduction of sampled messages. Yet, this feature is not unusual at all in computer graphics. Consider the well-known idea of sampling a logical input device, e.g., mouse position values. A separate object modelling (or directly interfacing) a mouse can send thousands of motion notification messages to a receiver object, and this latter can just "sample" these messages using the sampled message facility. More generally, the introduction of sampled messages enables us to model (using small, dedicated active objects) different sorts of interrupt handling, which is often necessary in multimedia programming.

2.1.2 Use of Active Objects The introduction of active objects gives an efficient and elegant solution to the problems raised by media synchronisation. Synchronisation of different media can be expressed in terms of reference points within each media type (reference points might be video frames, audio samples, etc.), and these can be readily modelled using the notion of event synchronisation [10, 11]. Using active objects, event synchronisation appears to be no more and no less than synchronisation of concurrent processes, i.e., concurrent active objects in our case. This does *not* mean that synchronisation becomes easy. What it *does* mean is that we can now use the terminology, the results, the machinery, etc., of the theory and the practice of concurrent programming. There has been a significant amount of work in this area in, for example, operating systems research, for decades [12]. We believe that this helps programmers to identify the problems properly and more easily, and prevent them from "reinventing the wheel" in some cases.

The idea of using active objects for multimedia programming is, *per se*, not new. Indeed, a group at the University of Geneva has used the same notion for their multimedia research [13,14]. However, their approach to active objects is more restrictive than the one described in §2.1.1. They essentially define source, filter and sink objects only, and build their applications based on these concepts alone. Although these simple active objects can be described easily in our model, we see no reason to restrict active objects this way. On the contrary, a richer functionality leads to application and programming facilities that cannot really be achieved by other means.

To make this last point clear, we must first agree on terminology. We speak of *parallelism* when a problem is solved using several, physically distinct, "real-world" processors. By *concurrency*, however, we mean something much more general. Essentially, concurrency is used when a problem can be decomposed into a set of cooperating, but functionally independent sub-units, (sometimes called *processes, threads,* or *light-weight processes*), which exchange information at well-defined points only. Obviously, parallelism is, in these terms, only a special form of concurrency. However, one can also use the concept of concurrency in environments where only one physical processor is available or, in a more general case, where the number of available processors is lower than the number of independent threads. One of the major conceptual differences between concurrency and parallelism is that, whereas for parallelism processes are "expensive" resources, in the case of concurrency, processes are "cheap", i.e., fast to start up, fast to communicate with, and one may use a possibly large number of them.

Traditional computer graphics has put a lot of effort into exploiting *parallelism*, which has been identified as one of the primary tools to achieve reasonable speedups for a number of practical problems (see, e.g., [15,16] for an overview of such techniques). It is, however, quite disappointing to see how little has been done in exploiting *concurrency* for the purposes of computer graphics, interaction, and, to relate this issue to the main topics of this paper, multimedia. It is a formidable intellectual experience to realize that if one frees oneself from the confines of the sequential paradigm and accepts that logical processes are "cheap", then a number of practical problems and applications can be described and solved incomparably more easily and more elegantly. In other words, independently of parallelism, there often is a pay-off in using concurrency, even if higher speeds are not (necessarily) achieved. Just as a practical example, the basic approach of using threads is very clearly one of the reasons for the undeniable technical superiority of the NeWS windowing system over X Windows [17]. More specifically, the management and the specification of interaction and/or input tools are examples where using concurrency may be extremely fruitful [18–20]. Specification and realization of reconfigurable graphics pipelines may also greatly benefit from using concurrency [21]. The assumption of having cheap processes is not only in line with the direction of future hardware development, it is also compatible with the current trend in the evolution of contemporary software systems. The increasingly more frequent use of "light-weight" processes within conventional operating systems is a clear indication (e.g., OSF/1, based on the Mach system [12, 22] of Carnegie Mellon, or SunOS [23]).

It is clear that with the inclusion of general active objects into our object model we intend to make a very strong case for using the concept of concurrency for multimedia programming. Although using active objects seemed to be just a necessity for the proper handling of media synchronisation, we do not consider this as a burden. On the contrary, we see the concept of active objects as a positive step

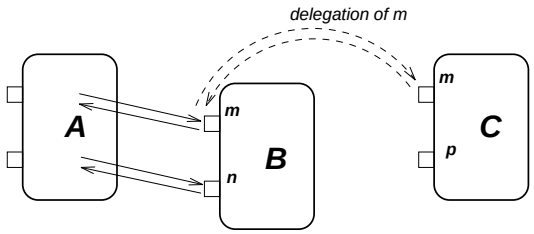

Figure 1: Delegation.

in the direction of creating more modular and more elegant solutions for a number of other problems that may arise. We intend to use active objects to describe complex interactions (by defining a concurrent set of basic interaction objects communicating with one another), to properly define so-called "multiple view" objects (which support the maintenance of consistency among the different views when an object changes through, e.g., user interaction), to interface to external hardware and software modules (i.e., mapping real parallelism onto our objects), and for a number of other purposes.

2.2 Delegation

Interaction among multimedia objects is inherently very dynamic. Logical links are created among objects, but these links are temporary in nature. One of the most important and, at the same time, most complex examples for these links is constraint management. Constraints may be set up among objects, which then affect their behaviour. These constraints are dynamic: a user may set them up and release them at a later point. The "multiple view", to which we referred in §2.1.2, is an example of a dynamically managed constraint. Managing pure geometric constraints is also a traditional problem in computer graphics and related fields.

Ideally, setting up and managing constraints should be "internal". This means that if a third party object communicates with constrained objects, this third party object should *not* have to know whether or not the object it is communicating with is subject to a constraint. This requirement is reminiscent of classical inheritance in traditional object oriented systems: a third party object does not know whether a specific message it sends is served by a method of its direct target, or by a method that this target object inherits from another object. There is a major difference, though: inheritance is static (i.e. it is defined when the type of the object is defined) and what we need for interactive multimedia applications is a dynamic capability. Hence our second major concept in MOM is *delegation*.

2.2.1 Definition of Delegation The precise semantics of delegation is based on the concept of *prototypes* as opposed to classes. A prototype is an object that represents the *default* behaviour for a set of similar objects. Any object can play the role of a prototype. New objects can re-use part of the knowledge stored in a prototype. To create an object that shares knowledge with a prototype, one has to construct an *extension* object, which has a list of personal behaviour idiosyncratic to the object itself, and a list of prototypes. The list of prototypes of an object can change during its lifetime. When an extension object receives a message, it first attempts to respond to the message using the behaviour stored in its personal part. If this is not possible, it has to find a prototype stored in the extension which would respond to the message. This last step is the *delegation* of the message. The exact semantics of delegation is described elsewhere [24]; describing it in full detail would go beyond the scope of this paper.

On more practical terms, with delegation (see also Figure 1) an object may delegate some or all of its behaviour (i.e., the messages it serves) to any number of other objects. For example, if object `B` *delegates* some part of its functionality to object `C`, e.g., method m in Figure 1, and if, at some later time, an object `A` sends a message to `B` invoking m, then the message will be served by `C` instead of `B`.

Furthermore, A does *not* know that actually C has served the message; the delegation relationship is internal to B. Note that it is then entirely possible to define methods for B through which a third party may ask B to delegate its functionally to an object C; in other words, the "target" of the delegation for object B may change during the life-time of B.

Classical *inheritance* is a closely related concept to that of delegation; it is simply a "static" form of delegation. This means that an object A may be defined to *inherit* the behaviour of object B, which is conceptually equivalent to the fact that A delegates its behaviour to B once and for all.

2.2.2 Use of Delegation The concept of delegation, although not really widely known and used, is not a new concept [24, 25]. It has also been proposed for use with graphics systems [25, 26] and in animation [27]. It has, however, never become widely accepted by the computer graphics community. The only example of its usage within the context of multimedia, (that we are aware of), is for the Event Script language of the Athena Muse project [28][1]. This, in spite of the fact that delegation is, in some sense, a more powerful concept than the concept of inheritance and opens up modelling possibilities which are not cleanly describable by other means. Delegation in combination with active objects leads to other potential benefits. Concurrency can be easily exploited via delegation between active objects (for example, an object might clone itself and delegate part of its behaviour to the clone) allowing both objects to proceed concurrently. A caller of the original object need not be aware of the exploitation of concurrency in this case and is not impeded by an redirection of the message since delegation always takes the shortest route between caller and delegatee.

In the remainder of this section, we show some examples for the use of delegation; see also §3 for a more complex example, involving active objects.

In §2.2 we referred to constraint management as one of the primary reasons for introducing delegation into MOM. We now show how constraint management can be described with the help of this concept (see also Figure 2). Suppose the two objects A and B are to be constrained in their behaviour described by their methods f and g, respectively. A separate constraint object may then be defined, denoted, say, by C, which has a counterpart for both methods f and g (say, f' and g'). These methods within C would then behave following the particular constraints defined by C. To "switch on" the constraint for A, the method f should be delegated to f' and, similarly, the method g of B should be delegated to g'. Note that a third party object does not know that delegation has taken place: it will still refer to f of A (and g of B, respectively). Switching the constraint "off" simply means to withdraw delegation. (Obviously, managing multiple constraints involves a more complicated scheme, but the additional complexities involved are not related to the notion of delegation).

Constraint management is not the only possible use of delegation. As mentioned before, delegation has already been used by, e.g., Zeleznik et al. [27] in animation. A practical usage of the concept there is to describe deformed geometric objects in an animated sequence (e.g., an object representing initially a circle, but becoming a more general rational B-spline curve as time passes). Animation being an integral part of the MADE project, such applications for delegation will naturally arise in our case, too.

Application for delegation arises in authoring as well. This notion refers to the fact that a final multimedia application is very often created off-line, much like a final movie is edited by a film editor, using sequences recorded during production. Tools for such authoring are necessary for a reasonable and general multimedia environment. One of the problems with authoring is, though, that the individual objects used to edit the final application may be quite time consuming (e.g., a long, computer-generated animation sequence). This makes the authoring job unnecessarily tedious and long. Delegation may be used to alleviate this problem: the objects in use may be "delegated" temporarily to some kind of simulation objects, thereby making the authoring job itself faster and handier.

At a panel discussion at SIGGRAPH'92, entitled "Graphics Software Architecture for the Future",

[1] Event Script is, however, *not* meant to be a general-purpose programming environment; it is used primarily for creating user-interface event handlers.

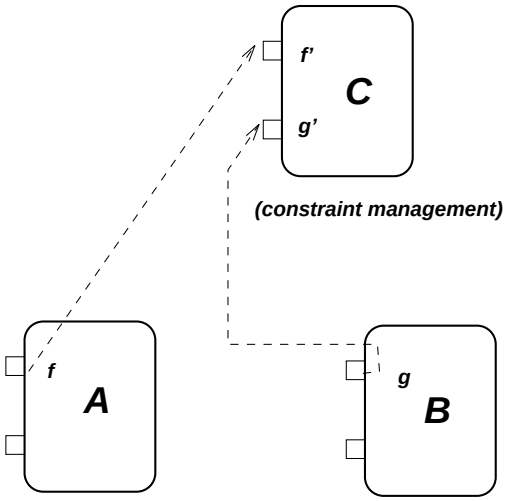

Figure 2: Constraint Management with Delegation.

S. Abi-Ezzi identified *time-critical computing* as an essential concept for future generation graphics systems [29]. In his view, time-critical computing suggests a new view of computer science in which the definition of correctness of a computation requires that the computation be completed within a specified deadline. In cases of overload, it is then perfectly acceptable to gracefully degrade the quality of the computation — obtaining an approximation to the actual result, in order to honour the deadline. This feature seems to be essential for future generation interactive systems, including multimedia.

Description of such graceful degradation with delegation is not only possible but is also very elegant. An object, representing, e.g., complicated graphics, may choose to delegate its own rendering to another object. The target of this delegation, i.e., the renderer itself, may then change at run-time, depending on the average load. When the load is low, the target of the delegation may be a very complex, but time-consuming renderer; however, if the load increases, the target of delegation may be "switched" to another renderer which only approximates the precise outlook of the geometry.

Delegation may also be used to increase the flexibility of user-interface design. One may have, for instance, several video sequences displayed in independent windows on the screen. A separate and general tool, a control panel, can then be dynamically assigned by the end-user to one of the video sequences. In other words, the user may want to change the binding of a video sequence and a control panel run-time. This change can again be described by delegation: the general interface tool refers to an abstract video player, and this latter (dynamically) delegates it own references to one of the real video sequences on the screen.

Other delegation examples that we are considering include run-time debugging and monitoring of object behaviour.

2.3 Classes as Objects

As said before, the exact semantics of delegation is described elsewhere and we cannot elaborate on it in more detail in this paper. There is, however, an important consequence of the precise semantics of delegation as far as the MADE toolkit is concerned. Namely, the concept of delegation removes the distinction between classes and instances. Indeed, *any* object can serve as a prototype. This means that in the MADE model, classes are objects, too[2].

This last fact has important consequences in practice. If classes are just objects, then (part of)

[2] This is analogous to the concept of "metaclasses" in some other object-oriented systems.

their behaviour can also be delegated. In practical terms, if an application defines a certain type of object, it can also dynamically modify the behaviour of *all* instances of that type in one step (and not only the behaviour of a individual instances). This significantly increases the modelling power of MOM.

3. A Larger Example: The Bouncing Ball

In this somewhat manufactured example, we exploit the use of delegation to demonstrate a form of sophisticated user control over a simple application scenario. The scenario is as follows.

A window is displayed showing a ball bouncing against some barrier. On each bounce, the ball emits either a specific sound, generates a shower of coloured sparks, or does nothing. The choice over which "collision response" will be used is controlled by the user through an interaction object (i.e. a widget). Other interaction objects provide control over the specific sound emitted (e.g. volume, pitch, sound file, etc.,) and the graphical representation of the shower of sparks (e.g. colour, spark density, etc.).

The ball is modelled by an active object that includes behaviours (methods) for updating its position, rendering itself, determining whether it has collided with the barrier, and providing a response if a collision has occurred. The barrier is modelled by a separate active object that includes a behaviour for determining whether a given position lies on the barrier. We are not concerned in this simple scenario with how the ball's position is updated or how it is rendered. We assume the ball determines whether it has collided with the barrier by communicating with the barrier object at each position update.

The default collision response behaviour of the ball object is to trivially do nothing. To produce a more interesting response the user chooses one of the other two behaviours mentioned above. This has the effect of sending a *new response* message to the ball instance, which then delegates its collision response behaviour to another object that provides an implementation of the chosen behaviour. To keep things simple, we assume that this object is an interaction object that presents some controls over the specific response. For example, the sound interaction object described in the initial scenario. If, at some later stage, the other collision response behaviour is desired, this simply translates into sending another *new response* message to the ball, informing it to delegate its collision response behaviour to the other response object (e.g. the spark interaction object).

Perhaps the most important aspect of this example is that the particular behaviour of a specific ball instance is controlled dynamically, at execution time, by the user. The implementation of the ball object does not include any information about the various response behaviours proposed in the scenario. The only capability it must support is the mechanism to delegate its response behaviour to some other designated object. Furthermore, at some later stage, the range of collision responses can be expanded to include other new behaviours (e.g. splashes of rain water) without any impact on the ball object itself.

The scenario can of course be made more complex. For example, we can instantiate more bouncing balls. Since each ball instance has its own collision response method that can be delegated using the mechanism we have described, one can easily construct a system with many balls each with a different collision response behaviour. In a similar manner to the small example described in §2.2.2, we might use a single "generic" interaction object, which can be "switched" between particular ball instances by delegation. Finally, we may wish to substitute the position update behaviour of the ball with one that constrains the balls motion within a gravity field imposed by the barrier object. This could be achieved by delegating the update method of the ball to a method contained within the barrier object. In this case, methods that are part of the ball object may need to be invoked by the delegatee (the barrier) in order to correctly determine the ball's new position.

4. MOM in the Real World

It is beyond the scope of this paper to describe the implementation details of MOM; we give only some important highlights here.

Our implementation aims primarily at Unix™ and MS-DOS™ environments running C++. Consequently, our general concepts have been mapped onto C++ classes. This mapping has been done within the framework of an extended C++-like language called mC++, which serves as the programming interface of the MADE toolkit. In the course of the adaptation of the conceptual model for mC++, certain abstract notions become more concrete and some restrictions are imposed on the general framework. This is especially true for the concept of delegation: C++ is based on inheritance, and the coexistence of the two notions within the same language cannot be achieved without some compromise.

In the conceptual model, all objects are active. The direct translation of this concept into mC++ means that every objects in mC++ has its own thread of control. An alternative approach taken, for example, in Concurrent C++ [30] keeps the concepts of thread and object as separate entities. From our point of view, this leads to a less elegant solution that can lead to difficulties when integrating threads and message passing with regular C++ classes.

To ensure efficiency, mC++ defines four different types of objects which can all be considered as special cases of the general notion of active objects. The four types of objects are as follows:

1. *active objects* are objects which have their own thread of control, with responsibility for servicing messages as defined in their specification. Active objects may delegate and may be delegated to and may have prototypes.

2. *mutex objects* (for mutual exclusion objects) are objects whose methods are executed in the thread of their callers. However, mutual exclusion and additional synchronisation means are provided to ensure that executing a method is done in a mutually exclusive manner. Mutex objects may delegate and may be delegated to and may have prototypes.

3. *unprotected objects* are objects whose methods are executed in the thread of their callers. No additional protection is provided, but these objects may also be delegated and may be delegated to. They can also have prototypes.

4. *C++ objects* are objects whose methods are executed in the thread of their callers. No additional protection is provided and delegation constructs are not allowed, i.e., they are the traditional C++ objects. They cannot have prototypes.

Mutex, unprotected, and C++ objects can be considered in this categorisation as *passive*.

For the first three types of objects, mC++ also defines an explicit but separate object known as a *prototype* that represents each class. These objects provide class-level behaviour, which can then be delegated, too. Instances of mC++ objects can be thought of extensions which provide additional behaviour to that defined in the prototypes. This is in accordance with the principles described in §2.3.

Inheritance in the context of mC++ simply means the use of C++'s inheritance mechanism. This is restricted in that inheritance hierarchies of non-homogeneous types of objects are allowed only if they conform to the following hierarchical relationships.

- An active object may only inherit from another active object.
- A mutex object may only inherit from an active object or another mutex object.
- An unprotected object may inherit from active, mutex, or unprotected objects.
- C++ objects may inherit from any mC++ object.

However, inheritance from an active object class to a mutex or unprotected class does not pass-on the notion of activity to these classes. Similarly, inheritance from a mutex object class to an unprotected class does not pass-on the notion of mutual exclusivity. Additional rules that control this inheritance relationship are straightforward but would be inappropriate to be included in this paper.

A precise syntax has been defined to describe all these objects,providing constructs to describe both their internal and external behaviour. The adopted syntax has borrowed much from that used in Concurrent C++ [30] although it is not identical to it. Some elements of the so-called μSystem [31] have also influenced our design. Unfortunately, none of these tools (or others) could be used directly to implement our concepts, hence the necessity to design our own language. Again, description of these syntactic and semantic details would go beyond the scope of this paper.

5. General Facilities

In §2 a pure object model is described. Yet this is not enough to offer a general programming environment for multimedia. The object model gives the conceptual framework of how objects behave and interact in general; additional utilities are then necessary to make the model really useful in practice. It is beyond the scope of this paper to give an exhaustive list of all different utilities which are in our general programming environment. We concentrate here on what is probably the most important of them all: structuring.

5.1 Object Structuring

Creating object structures is a well-known idea in graphics. However, structures in graphics packages have always been defined with only graphics in mind, and the fact that the applications may need some similar, but not necessarily identical, facilities for non-graphics purposes have usually been disregarded.

In MADE, we support the general notion of a graph with objects as its nodes. To this end, we define an *edge* to be the basic building block for the creation of structures. This means that facilities are provided to create a new edge between two objects; these edges have then a separate identity (they are separate objects instances). In the simplest case, edges may be defined for any pair of objects. Using inheritance, however, it is perfectly possible to define edges which are usable between two objects of some specified types only.

Using edges, facilities are then given to create complete structures, i.e., graphs. A specific structure is organised by a special object type whose methods correspond to the management of its appropriate graph. Using the inheritance hierarchy again, objects for special types of graphs can be defined (e.g., directed acyclic graphs, trees etc.). Note that these graph management objects can automatically perform strong type checking against the objects if the edges they refer to are restricted to certain types only.

Using these facilities, it is easy to build graphics object structures, similar to PHIGS structures [32], to the scene database of IRIS Inventor™ [33], to the so called GO trees in the GO graphics package [34][3], or to other graphics structures in various graphics systems. It must be stressed, however, that by defining an edge in its full generality, these structures are not restricted to purely geometric use (as it is largely the case for the examples cited). Structures may and will be used to efficiently implement (dynamic) hyperlinks among objects, to describe document structures and thereby facilitate the implementation of filters for HyTime [1] and ODA [2], to access databases, etc. Defining edges as basic building blocks also enables us to add to the MADE toolkit additional tools to check, analyse, transform, etc., structure graphs, using the machinery of known graph-theoretical algorithms.

Note that the structuring facility is completely orthogonal to delegation. In other word, a given object's position in a structure is totally independent of whether the object behaviour has been delegated or not. This simple fact has very important consequences. Referring to the example on animation in §2.2, a full scene may be described by a DAG with (some of) the nodes referring to geometric objects in general. However, the exact geometric nature of the object may be defined by delegation (see Figure 3), and hence becomes easily re-definable at run-time following some deformation, for example. A similar method may be used to assign a graceful degradation to an object, to support the concept of time-critical computing, described in §2.2.2. (Note that analogous constructions in PHIGS,

[3] GO will constitute the 2D graphics sub-system of the full MADE toolkit, hence its importance for us.

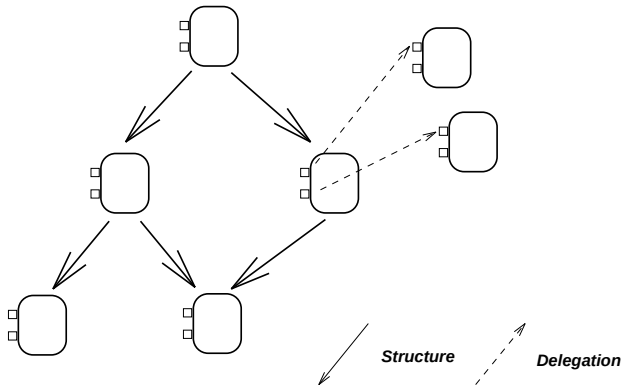

Figure 3: Structures and Delegation.

for instance, require deletion of a node and addition of a new one.)

6. Conclusion

In this paper, we present a general object model for multimedia programming. This object model relies very heavily on two concepts: active objects and delegation. We believe that this object model may serve as a basis not only for our own ESPRIT project but for multimedia programming in general. We also believe that the ideas outlined in this paper constitute a valuable set of inputs towards a more general standardisation effort which is currently being undertaken by national and international standardisation organisations in order to define a general framework for interactive multimedia programming interfaces.

Acknowledgements

This work has been financed by the European Communities' ESPRIT III project No. 6307, MADE. Much of the ideas were generated by the concerns of the GO project [34] (which was part of the ESPRIT II MULTIWORKS project), whose results will become part of the full MADE toolkit. The ideas described in this paper have emerged through long discussions we had at CWI and reflect the ideas of several people, including Paul ten Hagen, Michal Haindl, Frans Heeman and Arno Siebes. Our ideas have also been discussed with our partners in the MADE project; the contribution of Jacques Davy, Oliver Jojic, François Leygues, Philippe Smadja (BULL, France) and of Nuño Guimarães (INESC, Portugal) are very much appreciated.

References

1. S. Newcomb, N. Kipp, and V. Newcomb, "The "HyTime" – Hypermedia/Time-based document structuring language," *Communication of the ACM*, vol. 34, pp. 67–83, November 1991.
2. P. Hoepner, "Synchronizing the presentation of multimedia objects — ODA extensions," in *Multimedia — Systems, Interaraction and Applications* (L. Kjelldahl, ed.), EurographicSeminar Series, pp. 87–100, Berlin – Heidelberg – New York – Tokyo: Springer Verlag, 1992.
3. J. Davy (ed.), Paris, *MADE 1, ESPRIT III Project 6307, Technical Annex*, March 1992.
4. B. Stroustrup, *The C++ Programming Language*. Reading, Massachusetts: Addison-Wesley, second ed., 1991.
5. F. Arbab, I. Herman, and G. Reynolds, "An object model for multimedia programming," Tech. Rep. ISO/IEC JTC1/SC24 WG6 N003, International Standard Organization, New Orleans, 1993.

6. F. Arbab, P. ten Hagen, M. Haindl, F. Heeman, I. Herman, G. Reynolds, and A. Siebes, "Specification of the MADE object model," Tech. Rep. ISO/IEC JTC1/SC24 WG6 N004, International Standard Organization, New Orleans, 1993.
7. E. Fox, "Multimedia: Application and practice," Tech. Rep. ISSN 1017–4656, EG92 TN 6, Eurographics Technical Report Series, Aire-la-Ville, September 1992.
8. D. Bulteman, G. van Rossum, and D. Winter, "Multimedia synchronization and UNIX – or – if multimedia support is the problem, is UNIX the solution?," in *Proceedings of the EurOpen Autumn Conference* (A. Finlay, ed.), (Budapest), pp. 127–144, September 1991.
9. A. Goldberg, *Smalltalk-80: The Interactive Programming Environment*. Reading, Massachusetts: Addison-Wesley, 1984.
10. N. Guimarães, N. Correia, and T. Carmo, "Programming time in multimedia user interfaces," in *Proceedings of the UIST'92 Conference*, (Monterey, California), 1992.
11. N. Guimarães, N. Correia, and T. Carmo, "Xt based support for continuous media," in *Proceedings of the Xhibition'92 Conference*, (San Jose, California), 1992.
12. A. Tanenbaum, *Modern Operating Systems*. Englewood Cliffs, New Jersey: Prentice-Hall International, Inc., 1992.
13. S. Gibbs, L. Dami, and D. Tsichritzis, "An object-oriented framework for multimedia composition and synchronization," in *Multimedia — Systems, Interaraction and Applications* (L. Kjelldahl, ed.), EurographicSeminar Series, pp. 100–111, Berlin – Heidelberg – New York – Tokyo: Springer Verlag, 1992.
14. V. de May, C. Breiteneder, L. Dami, S. Gibbs, and D. Tsichritzis, "Visual composition and Multimedia," *Computer Graphics Forum (Eurographics'92)*, vol. 11, no. 3, pp. C9–C21, 1992.
15. F. Crow, "Parallel computing in graphics," in *Advances in Computer Graphics VI* (G.Garcia and I. Herman, eds.), EurographicSeminar Series, Berlin – Heidelberg – New York – Tokyo: Springer Verlag, 1991.
16. S. Green, *Parallel Processing for Computer Graphics*. Research Monographs in Parallel and Distributed Computing, London: Pitman, 1991.
17. W. Roberts, M. Slater, K. Drake, A. Simmins, A. Davidson, and P. Williams, "First impression of NeWS," *Computer Graphics Forum*, vol. 7, pp. 39–58, 1988.
18. P. ten Hagen and H. Schouten, "Parallel graphical output from dialogue cells," in *Eurographics'87 Conference Proceedings* (G. Maréchal, ed.), (Amsterdam), North–Holland, 1987.
19. D. Duce, R. van Liere, and P. ten Hagen, "An approach to hierarchical input devices," *Computer Graphics Forum*, vol. 9, pp. 15–26, 1990.
20. D. Soede, F. Arbab, I. Herman, and P. ten Hagen, "The GKS input model in MANIFOLD," *Computer Graphics Forum*, vol. 10, no. 3, pp. 209–224, 1991.
21. G. Reynolds, "A token based graphics system," *Computer Graphics Forum*, vol. 5, pp. 139–145, June 1986.
22. D. Black, "Scheduling support for concurrency and parallelism in the Mach operation system," *IEEE Computer*, pp. 35–43, May 1990.
23. SUN Microsystems, *SunOS Manuals, Lightweight Processes, revision A*, 1990.
24. H. Lieberman, "Using prototypical objects to implement shared behavior in object oriented systems," in *Proceedings of the First ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, (Portland), pp. 214–223, ACM Press, September 1986.
25. A. Borning, "Classes versus prototypes in object-oriented languages," in *1986 Proceedings of the IEEE/ACM Fall Joint Computer Conference*, (Dallas, Texas), pp. 36–40, November 1986.

26. P. Wißkirchen, *Object-Oriented Graphics*. Symbolic Computation, Berlin – Heidelberg – New York – Tokyo: Springer Verlag, 1990.

27. R. Zeleznik, D. Conner, M. Wloka, D. Aliaga, N. Huang, P. Hubbard, B. Knep, H. Kaufman, J. Hughes, and A. van Dam, "An object–oriented framework for the integration of interactive animation techniques," *Computer Graphics (SIGGRAPH'91)*, vol. 25, pp. 105–111, July 1991.

28. M. Hodges, R. Sasnett, and M. Ackerman, "A construction set for multimedia applications," *IEEE Computer*, vol. 22, pp. 37–43, January 1989.

29. A. van Dam (chair), S. Abi-Ezzi, C. B. ad R. Carey, and M. Tarlton, "Graphics software architecture for the future. Notes of panel discussion at SIGGRAPH'92," *Computer Graphics (SIGGRAPH'92)*, vol. 26, pp. 389–390, July 1992.

30. N. Gehani and W. Roome, "Concurrent C++: Concurrent programming with class(es)," *Software — Practice and Experience*, vol. 18, pp. 1157–1177, 1988.

31. P. Buhr and R. Stroobosscher, "The μSystem: Providing light–weight concurrency on shared–memory multiprocessor computers running UNIX," *Software — Practice and Experience*, vol. 20, pp. 929–964, 1990.

32. T. Howard, W. Hewitt, R. Hubbold, and K. Wyrwas, *A Practical Introduction to PHIGS and PHIGS PLUS*. Workingham – Reading: Addison–Wesley, 1991.

33. P. Strauss and R. Carey, "An object-oriented 3D graphics toolkit," *Computer Graphics (SIGGRAPH'92)*, vol. 26, pp. 341–349, July 1992.

34. J. Davy, "Go: A graphical and interactive C++ toolkit for application data presentation and editing," in *Proceedings of the* 5^{th} *Annual Technical X Conference on the X Window System*, January 1991.