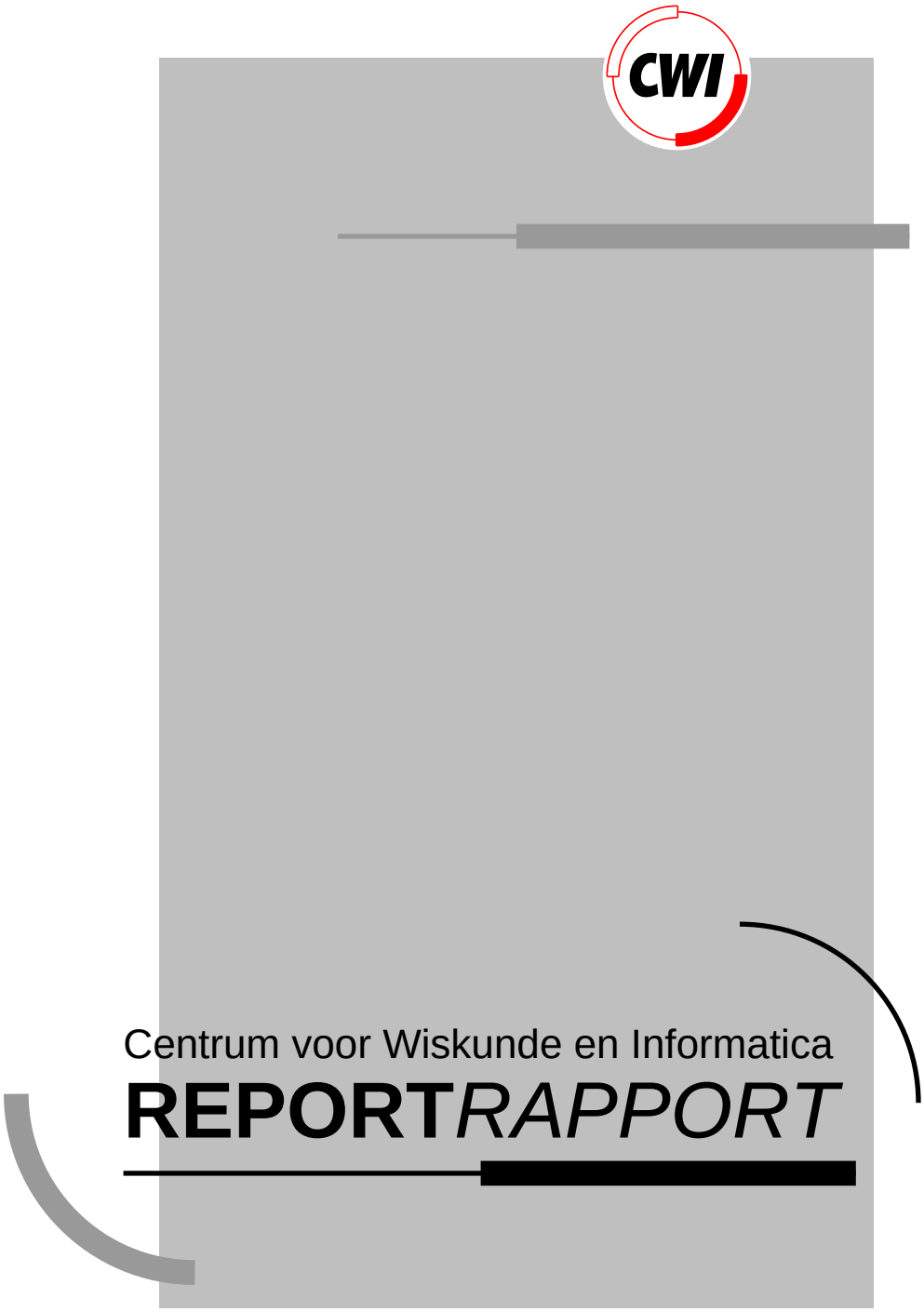

Centrum voor Wiskunde en Informatica

REPORT*RAPPORT*

MADE: A Multimedia Application Development Environment

I. Herman, G.J. Reynolds, J. Davy

Computer Science/Department of Interactive Systems

CS-R9360 1993

MADE: A Multimedia Application Development Environment

Ivan Herman, Graham J Reynolds
CWI
Kruislaan 413, 1098 SJ Amsterdam, The Netherlands
ivan@cwi.nl, reynolds@cwi.nl

Jacques Davy
Groupe Bull
7, rue Ampère, Massy 91343, France
J.Davy@frmy.bull.fr

Abstract

MADE is the acronym for an ESPRIT III project aiming at the development of a programming environment for multimedia applications. The resulting software library is based on C++, and is planned to operate on UNIX Workstations as well as on PC–based platforms. This reports gives a technical overview of the project, and describes some possible application scenarios where the MADE environment can be of a great help for multimedia programming.

AMS Subject Classification (1991): 68N99
CR Subject Classification (1991): H.2.4,H.4.m,H.m,I.3.2,I.4.9,
Keywords & Phrases: Multimedia systems

1. Introduction

One of the most significant developments in computing technology over the past few years is the emergence 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, most of the available multimedia environments aim at *hypermedia authoring*, ie, they offer means to interactively create hypermedia documents. "Document", as a multimedia term, means more than our traditional paper–based understanding. It should be perceived as a potentially complex composition of related media information, thus it is a multimedia document, which can be "read" or viewed in a non–sequential fashion by following semantic connections (or links) between the various media components, hence it is hypermedia.

Although the concept of hypermedia document is very powerful indeed, it does not cover all possible fields of applications of multimedia. The ability of combining, modifying, or even synthectically creating multimedia data is often necessary for more complex multimedia applications. For example, the user might want to extract a frame from a video sequence, modify it with standard image processing tools, combine the image with the output of synthetic graphics, and possibly exchange the original

frame with the modified image. Description of such actions does not fit easily in the model of a hypermedia document, in spite of the sophisticated interaction tools which are usually provided with authoring environments. There is, therefore, a need for a programming environment which would allow for the developments of such applications, too.

The techniques to achieve combination of media are extremely disparate, and they use the results of various fields of computing technology like, for example, high quality synthetic graphics, image processing, speech synthesis, etc. Some of the techniques are also highly application dependent. It is almost impossible to define a closed programming environment which would encompass all needs. The already traditional answer to this kind of challenge is to use object–oriented techniques: services are offered in the form of objects, which are then extensible by the programmer to include any necessary application–dependent tools.

The European Communities' ESPRIT III project MADE (Multimedia Application Development Environment[16]) has set up the ambitious goal of defining and implementing such a portable object–oriented development environment for multimedia applications, based on C++[28]. The outcome of the MADE project should be a programming environment running on various UNIX platforms, as well as on MS–DOS or Windows–NT environments. This report gives a general overview of MADE. It describes its major services, and it also gives an overview of some possible "application scenarios", ie, major application architectures which may use these services. It is not the purpose of the paper to give a detailed technical description of the full project; this would go far beyond the scope of such a report. The interested reader should consult the "official" MADE documents to gain a more detailed insight (eg, [2, 8, 9, 10, 30, 16]).

The MADE project is still an ongoing activity. Consequently, some problems are still open and will be solved only later in the project. For this reason this report sometimes raises issues without presenting complete solutions.

2. General Overview

The full MADE environment contains a large number of different objects and related services. Two important categories of these objects have a major role in the development environment; these are: *toolkits* and *utilities*. (Note that the object–oriented nature of MADE makes it possible for the end–user to add new objects to both toolkits and utilities and/or to extend the functional capability of existing ones.)

The *toolkit* level is a collection of objects that are considered to be fundamental for multimedia programming. It includes, obviously, objects to interface different media. Also, it includes objects which, although not directly involved in handling specific media, play a fundamental role in constructing more complex multimedia applications. Some more details of the toolkit level will be given below (see §3).

Although it is possible to construct complex applications using the MADE toolkit level only, doing that may be unnecessarily tedious and error–prone. Consequently, another layer has been defined on top of the MADE toolkit, called *utilities*. The idea here is to define and implement objects which include complex functionality and which are considered to be essential for most multimedia applications. Applications programmers may choose to use some of these utility objects; however, the toolkit level is never completely obscured, and the application layer is free to use toolkit objects directly as well (see also Figure 1; some of the terms appearing on the Figure will be described in later sections).

In the early stage of the MADE project a common *object model* was defined and developed, to ensure the smooth cooperation among objects within the MADE library and also to provide a clear approach to some of the technical issues raised by multimedia programming in general. This object model defines a conceptual layer on the top of the implementation language of MADE (ie, C++),

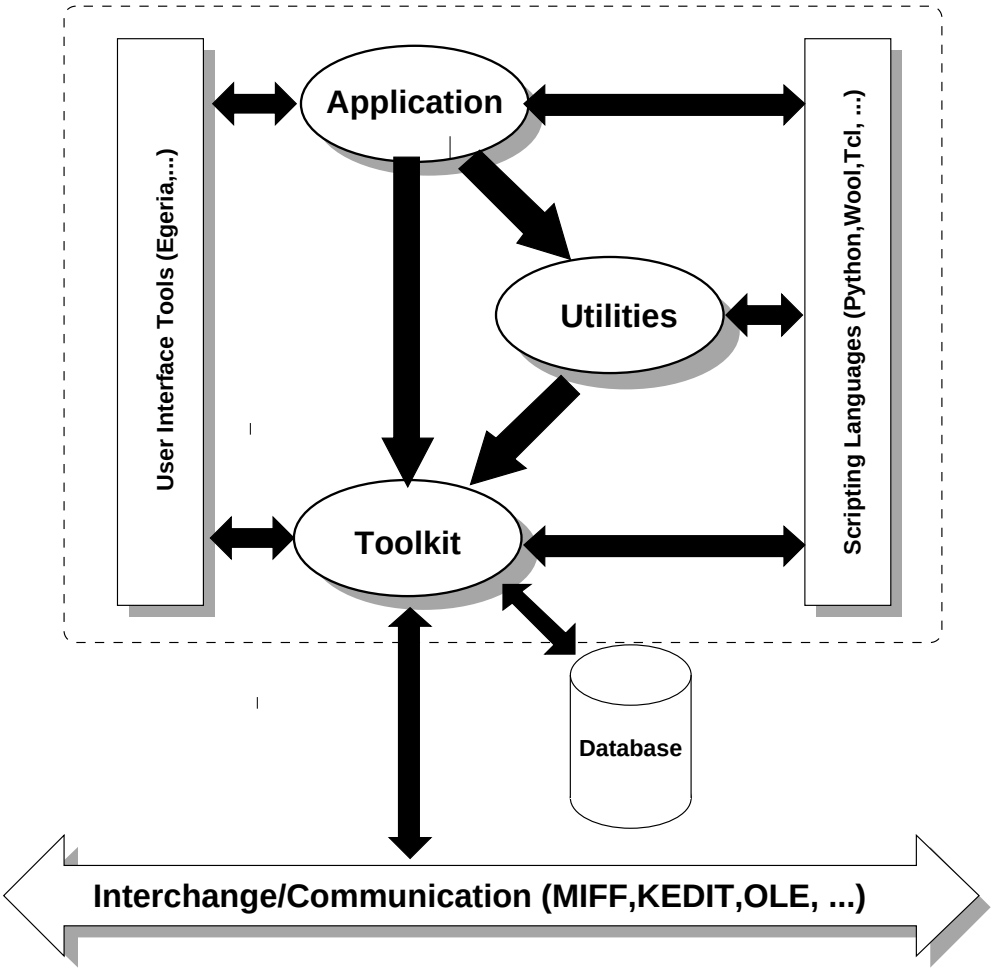

Figure 1: Toolkits and Utilities

and it describes numerous features of objects within MADE. As far as the application programmer is concerned, two characteristics of this model are of a great importance: the use of *active objects* and the presence of *delegation*.

In MADE, objects may be *active*, ie, they may have their own thread of control (within the shared address space of the same UNIX, MS–DOS, or Windows–NT process). This fact is exploited in the implementation of the MADE toolkit library, and is a major tool used in defining synchronization of different media (see §3.2.3 below). Application programmers may have to be aware of this fact if they decide to use the toolkit level objects directly.

The concept of *delegation* of object methods is the other central feature of the MADE object model. Using delegation an object may delegate some or all of its behaviour (i.e., the messages it serves) to any number of other objects, which will then act on its behalf. The notion is not unlike inheritance, but delegation is dynamic, ie, the target of delegation may be set and re–set at run–time. Delegation plays a very important role in controlling constraints in MADE (constraint objects are part of the toolkit), and offers advanced means to describe temporal behavioural control. A more exact semantics of delegation is described in, eg, [20]; see also [1] for a more detailed description of the concept within the framework of the MADE object model.

The object model has been realized in the form of an extension of C++, called mC++. The mC++ compiler generates a set of C++ classes, library and macro calls; this "intermediate" level can also be accessed by programmers directly, in case they do not intend to use yet another programming language (see [17]). Details of the object model are, however, hidden to most application programmers and are only of real interest for toolkit or utility developers. The full technical description of this object model will be omitted here; the interested reader should refer to [1] for a general overview and to [2] for a complete description of the model and of mC++.

The object model is not the only means to achieve smooth cooperation among objects. All MADE objects also include general features that allow them to be used under various circumstances in a unified way. Some examples of these features, which are necessary to understand what is described in later sections, are given below.

All objects in the MADE system may be *permanent*. This means that they may "store" themselves in a database and can restore their content at a later stage of the application's lifetime or even during the execution of some other applications. This feature is present for all MADE objects by default; the only step the application program has to do is to invoke certain implicitly defined member functions. Furthermore, the MADE toolkit level includes a special object which can interface with various database systems. Although this interface obviously cannot cover all known database systems, it does provide an interface to some object–oriented and relational databases. Here again, the general features required by the database access is included in all MADE objects in a database–independent way, and the details of the database access is hidden in the general database management object of MADE (see [29]); interfacing to a new database system means the specification of an appropriate sub–class.

MADE objects, primarily utility objects, are also prepared for distributed access. This not only means that the MADE library includes specific objects for inter–process communication, but also that MADE objects are prepared to "convert themselves" into a format suitable for communication and, conversely, can "reconstruct" their internal state based on data coming from a communication channel. A sophisticated object–oriented communication protocol (called KEDIT[18]) is currently under development for UNIX platforms, which will allow MADE applications to offer object–based services, and will provide means for the transfer of full MADE objects from one MADE application to another. The features offered by the combination of MADE objects and KEDIT are similar to the kind of object services defined by the Object Management Group[1]. On MS–DOS and Windows–NT platforms the OLE protocol will be used to provide similar facilities; this is already a de–facto standard on these environments.

All MADE objects include a general mechanism known as a "dynamic call interface". This interface makes it possible to call a member function of an object by knowing the object's handle and a *string* describing the full signature of the member function. This string can be constructed at run–time, hence the "dynamic" nature of the call. This feature permits MADE objects to be accessed easily from scripting languages, and provides a simple way of constructing interfaces to other programming languages (eg, C or Fortran). It also makes the implementation of distribution support (like KEDIT) fast and easy.

3. Toolkit Objects

The primary goal of the MADE toolkit is the provision of a set of features and facilities basic for multimedia programming. These include control over different media, and other types of objects, which have also been identified as playing a fundamental role.

[1]OMG is an industrial consortium aiming at the definition of object services in general. In their CORBA specification[24], OMG gives a specification for object services in a distributed environment. However, CORBA is still not final, nor is there a reliable implementation available yet. If, by the end of the MADE project, OMG produces a final version of their specification, replacing KEDIT with this specification will be considered.

3.1 Media Objects

The MADE toolkit includes *media objects*, ie, objects whose role is to directly control different media in a unified and hardware/firmware independent way.

The toolkit includes four main categories of media objects: graphics objects (for two and three dimensional graphics), animation, sound and video objects. The functionality of these objects is defined in a device–independent way; porting the toolkit to a new environment involves changes on the interface level only. In other words, all of these objects "hide" their respective device–dependencies behind specific, low–level abstract interface objects, thereby cleanly separating their MADE specific behaviour from particular device dependent features. Adaptation of a media object type to a new environment simply requires the definition of a new device–dependent subclass of the appropriate general interface object.

Some of the categories listed above contain relatively simple objects. Their task is to provide a mapping from the MADE library structure onto their respective interface object. This is the case, for example, with video and audio objects. The most critical aspect of the definition of these objects is synchronization. The objects and their device specific interfaces must be matched with the synchronization model of MADE (see §3.2.3 below) and with the requirements and facilities provided be the specific hardware that is used.

Audio and video objects rely on Microsoft's Multimedia Environment for MS–WINDOWS, which is a de–facto standard in this area. On UNIX, portable video and audio services are used: the Video Extension of the X Windows system for analogue video ([5]) and the AudioFile server for audio ([19]). For digital video, the UNIX solution is not yet defined.

2D and 3D graphics require a much higher level of complexity. Indeed, the collections of both the 2D and the 3D graphics objects represent two full–blown subsystems per se, which are also usable stand alone for graphics purposes.

For 2D graphics, the MADE toolkit reuses an already existing object–oriented 2D graphics system, called GoPATH[7], by adapting it to the requirements of MADE. These 2D objects include different two dimensional shapes, associated clipping areas, composition rules, attributes (eg, colour, line type, text font), etc. The programmer has the possibility, via sub–classing, to define new shapes and include these into the full 2D world. GoPATH is currently based on X11R5 for UNIX platforms, and on MS–WINDOWS under MS–DOS; a Windows–NT version is also operational.

The 3D subsystem provided by MADE supports a mapping between general 3D objects (shapes, surfaces, lighting and view control, etc.) and existing 3D packages. A mapping to SGI's GL library is currently being developed. The use of PEX[6] or Open-GL, as a replacement for GL, will be considered in the future. It has to be stressed that it is *not* the goal of the MADE development to define yet another three dimensional graphics package; the emphasis is more to provide an object–oriented layer on top of existing packages, integrated into the MADE environment. On the other hand, due to the object–oriented nature of the MADE toolkit, it is possible to extend, by sub–classing, the basic 3D functionality (eg, to add a proprietary ray–tracing module).

Graphics objects (both in 2D and 3D) do not have a temporal dimension; essentially, they describe static scenes. This is in contrast with the inherently temporal nature of video and audio objects. To alleviate this contradiction, MADE includes separate *animation objects*, which describe, and even automatically generate, sequences of scenes. The methods and algorithms used in animation may be extremely complex and, more importantly, dependent on specific application areas. Although simpler, built–in animation techniques based on animation curves are also available, MADE animation objects also allow for animation scripts, using a scripting language, which is then interpreted by the MADE animation objects.

Animation objects may be active objects, too, thereby subject to the same synchronization be-

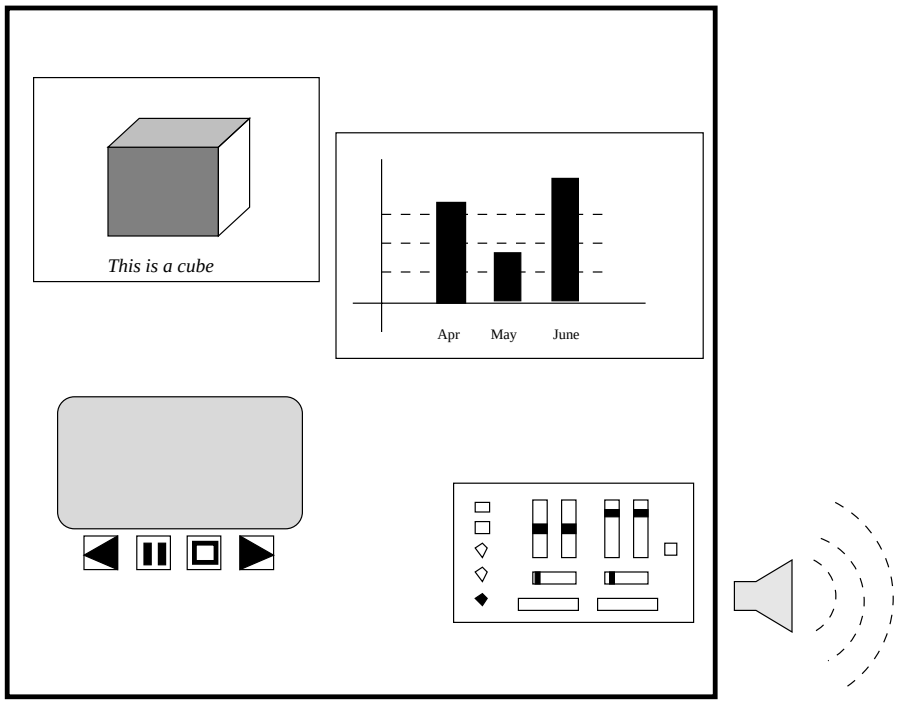

Figure 2: A rudimentary example for multiple media in an application.

haviour and control as audio and video objects (see §3.2.3).

3.2 Combination Objects

It is of course possible to build very spectacular programs that only rely on MADE media objects, using complex and possibly animated 2D and 3D graphics, running a video on the screen and playing audio. However, the shortcomings of such an approach are very soon visible if more complicated application programs have to be devised and implemented on this basis. As the very rudimentary example on Figure 2 already shows: interactive behaviour assigned to graphics objects have to be combined to control video output; visual representations for audio control have to be defined and implemented; 2D and 3D objects have to be combined in one picture, etc.

The basic media objects become really usable if they can be *combined* in variety of ways. Combination of media objects (and MADE objects in general) within an application has received a particular emphasis in the specification of the MADE project in order to enhance the usability of the tools. Five major areas of combination have been identified, and we shall return to each of them in some detail. The five areas of combination are: *imaging*, *structuring*, *synchronization*, *interaction*, and *constraint management*.

3.2.1 Imaging. Some of the media objects produce *images* on the screen. This is the case for 2D and 3D graphics, and for video objects. It is also possible to import and to export digital images in various formats (TIFF, GIF, etc.). A natural consequence of this is the requirement for an application to combine these images directly. For example, one may want to create a complex picture by using a snapshot of a video sequence, annotated with a generated (2D) text, filtered through a pattern read from a TIFF image, etc.

To achieve this functionality, MADE defines a special *image object*, which can be used as a common

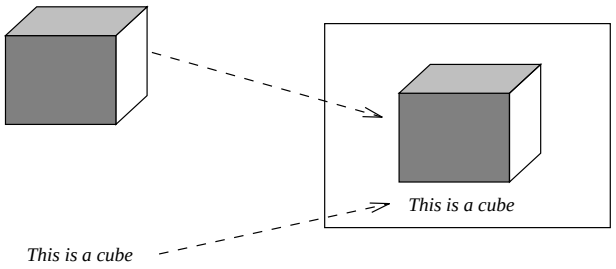

Figure 3: Combination of a 3D and a 2D object in one image

platform for the combination of pictures. 2D and 3D objects can produce such images, they can be generated from various image file formats, and individual frames of video sequences may be converted into images. These images may then be visualized on the screen, or can be converted back into video frames. In the future, complex image processing functions will be defined to operate on these images, combining them, filtering them, etc. Here again, due to the object oriented design, it will be possible for the end–user to add their own application dependent image processing functionalities.

3.2.2 Structuring. The importance of *structuring*, ie, of creating aggregates of different objects in interactive programs, has long been recognized in computer graphics. All graphics packages provide some form of aggregation, like structures in PHIGS[12], the scene database of IRIS Inventor[27], or the so called Go trees in GoPATH[7]. Although the structures used in these examples are all relatively simple (directed acyclic graphs or just trees), the appearance of hypertext and, lately, of hypermedia systems makes it clear that more general aggregation facilities are also necessary.

To answer these demands, the MADE toolkit includes a general graph management facility in the form of *graph objects*. These objects allow for the specification, management, and the traversal of general graphs, with no restrictions imposed on their types. Nodes of these graphs may refer to any MADE object. Graph management is achieved via special MADE objects; consequently, recursive graphs may also be defined via the same mechanism. Individual MADE objects can be referred to by several graph nodes (ie, they can be *shared*).

Graph objects provide a sound basis for the type of structuring required by graphics as well as for complex hypermedia navigation systems. They are fully integrated into the full MADE structure, which has a number of advantages. As an example, graphs provide an automatic defence against concurrent access of structures by active objects, they can be exported and imported using the very same mechanism as for all other MADE objects (ie, complete structures may be stored in databases), etc.

3.2.3 Synchronization. The issue of synchronization has always been one of the central problems of multimedia applications; it is therefore necessary for the MADE toolkit to offer a consistent solution to this issue.

The fundamental synchronization scheme used in MADE is called *reference point* synchronization. For each, so called, *synchronizable* MADE object a series of media specific reference points may be defined (for example, video frames, audio samples, etc.). Each reference point contains internal "instructions" for synchronization, references to other synchronizable objects that are to be synchronized with it, etc. Synchronizable objects are active objects; when they reach a reference point, synchronization is performed by exchanging messages with other active objects, waiting for their replies, etc. The reference point model has been greatly inspired by [3]; its details in the MADE environment are specified in [8]. Audio, video, and animation objects are obvious examples of synchronizable MADE

objects[2]. The MADE programmer may create new, application–specific synchronizable objects, too.

Based on this synchronization model, the MADE toolkit also includes a higher–level mechanism for time–based synchronization. This mechanism defines different types of *schedulers* which the application may use as building blocks for more complex time–based synchronization scenarios (see [8] for further details). These schedulers all assume the existence of a special synchronizable object within MADE, namely a *timer*. The approach of building time–based synchronization on the top of a more general mechanism, instead of considering it as a basic feature, allows the MADE library to be used in environments which do not necessarily offer real–time facilities.

3.2.4 Interaction Objects. Multimedia applications are very often highly interactive; it is therefore essential to give very good tools to construct complex interaction scenarios involving MADE objects.

The MADE project does *not* aim at developing a completely new user interface management system. Instead, MADE objects may be embedded into an existing user interface environment, like the Athena Widget set of X Window System, the Motif toolkit, MS-Windows or, in the future, Windows–NT. Nevertheless, not all user interaction can be adequately managed by these tools; many complex interaction scenarios will involve MADE objects directly (eg, for direct manipulation). The scheme developed in MADE for achieving these complex interaction scenarios is based on the notion of *sensors* and associated *interaction objects*.

Sensors are best understood in the context of graphics: in this context they define sensitive areas on the screen, which can be "activated" by external interaction, typically mouse events. Sensors are associated with MADE objects via interaction objects. In effect, they provide a sensitive region which acts as a focal point for interaction with these objects. For some objects, sensors cannot be attached to the object itself, but, instead, a visual representation of the object is used, in the form of graphics object. This might be the case, for some sensors attached to audio objects. The notion of sensor is general enough to accommodate regions involving higher dimensions including time. It can also be applied in association with interaction input devices that provide non–geometric input measures, such as audio input devices, pressure sensitive devices, etc.

Sensors forward events to interaction objects; it is part of the sensor's initialization procedure to decide which interaction object it is connected to. The interaction objects react on these events, following some patterns which describes the behaviour of the interaction object. Several sensors may be connected to the same interaction object.

In very simple cases, interaction objects perform straightforward and predefined tasks (like, for example, reshaping a graphics object). In other cases, much greater complexity may be required, perhaps providing control over several MADE objects and receiving events from several sensors (eg, the video control board depicted in Figure 2 reacts on the sensors of the graphical objects describing the four push–buttons, may control the visual appearance of these buttons and, of course, controls the video object proper; see also Figure 4). To describe such complex interaction behaviour, MADE introduces a type of interaction object that implements a general finite state machine (see [11]). These objects have a default finite state machine for a specific interaction scenario; however, the user can also assign a script to an interaction object, which, conceptually, includes a complete scripting interpreter (see also §4.1.2). Such a script automatically overrides the default behaviour of the interaction object. This high degree of openness, with respect to the end–user, is a very valuable feature of the MADE interaction management.

3.2.5 Constraint Management. Provision of a general purpose constraint system within MADE for all of the potential uses of constraints in a multimedia development environment would justify

[2] To be very precise, certain animation objects, which describe random animation, cannot be properly synchronized, but these objects represent a small minority vis-à-vis animation objects in general.

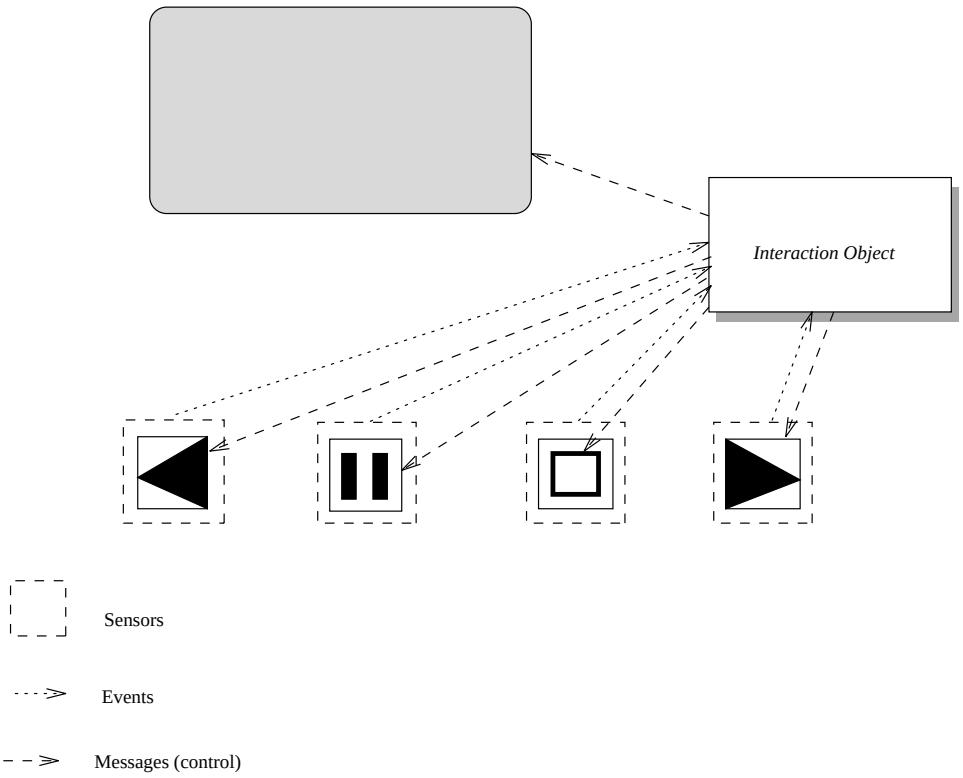

Figure 4: Use of Interaction Objects. Sensors forward events to objects, which controls the geometric appearance of buttons and the real video object.

a development project in its own right. Fortunately, there are various restricted types of constraint satisfier that, while not being as capable in some aspects, still provide useful functionality for dealing with certain categories of constraint.

The approach followed in the specification of constraints within MADE (see also [30]), is to consider those applications of constraint systems that are of direct relevance to the multimedia part of MADE. In effect, this restricts the scope of the constraint satisfier to the topics of geometric layout, user interface control, animation, and media synchronisation. For example, the MADE presentation facilities include a composition editor/player which may make use of constraints when defining the hypermedia document structure and presentation characteristics. What this classification means is that any predefined constraints that are defined as part of the toolkit can be organised into "constraint families" (to avoid the use of the term "class") which are relevant for specific multimedia aspects.

For the time being at least, only one–way constraints are proposed for MADE. While multi–way constraints provide greater expressive power to the constraint user, they also require more complex constraint satisfaction algorithms and may involve more effort on the part of the programmer to set up specific constraint objects. This decision need not be considered final, and can be reviewed after experience with the proposed constraint objects has been gained.

4. Utilities

As said before, *utilities* offer a higher level of functionality which make the implementation of more complex multimedia applications easier and faster. In fact, the functionality of some of the utilities is so complex that, by "wrapping" them into a simple program, they can be used as a separate

application programs in their own right.

The major categories of utilities are as follows:

1. *application program interface utilities*: visual metaphors, scripting, user interface builders, user monitoring;
2. *monomedia editors*: 2D and 3D graphics editors, animation, video, and audio editors;
3. *composition utilities*: framework for hyperdocument management, synchronization editors, interaction and graph object editors;
4. *miscellaneous*: class browsers, generic on–line help facilities, object monitoring.

Different MADE utilities may and do rely on one another, too. For example, the visual metaphors, to be presented below (see §4.1.1), are reused by monomedia editors (see §4.2), or, to take another example, the user interface of some of the editors may be developed with the help of MADE user interface builders and scripting languages usable from within MADE.

Utilities, together with MADE toolkit objects, offer a set of building blocks which can be used in various ways to create different types of MADE application program architectures. Some of the most common scenarios will be described in §5 below; however, to make these scenarios understandable, some of the most important MADE utilities are presented below in somewhat more details.

4.1 Application Program Interface Utilities

Application program interface utilities give a set of tools that help an application programmer to prototype or to develop a final MADE application. Although the facilities provided by some of these utilities are fairly standard these days, it is nevertheless necessary to provide them in the context of the MADE environment, too. Note that not all tools are presented in this report, only some of the most important ones.

4.1.1 Visual Metaphors. The visual representation and control of media objects is not always obvious. Indeed, to control certain attributes of media objects, relatively complex visual tools, with associated interaction, have to be developed. These tools may then be used on different levels: in program development, in authoring, or in the final playback of authored documents. These visual metaphors play an essential role in defining complex interactions operating on the objects; indeed, it is sometimes much easier to attach a sensor to these metaphor objects, rather than to try to define a sensor on the object proper (see §3.2.4).

There are numerous examples for such visual metaphors. Just to give some examples:

- Video control board for stopping, playing, rewinding, providing fast forward and backward motion, etc.
- Audio panel containing volume control, channel control, etc.
- Control boards for the manipulation of graphics object attributes (colour, lighting, shading attributes, etc.)

All these objects, collectively called *visual metaphor* objects, are part of the MADE utility library. Other utilities (primarily the editors, see §4.2), reuse these objects, thereby providing a common look–and–feel among MADE utilities. MADE applications may of course choose to ignore these objects and to implement similar user interface facilities by themselves.

4.1.2 Connection to Scripting Languages. Several MADE objects make use of scripting languages; animation and interaction objects have been mentioned in the preceding sections, and there are others, too. It is also perfectly feasible to create full–blown applications, either in a prototype or even in a final form, where the "user–level" program is in fact a script.

MADE does not introduce it own scripting language. Instead, all objects that make potential use of scripting access the interpreter functionality via an abstract general scripting interface. This general scripting interface is then specialized to access specific languages and their interpreters. This lets the final choice over which scripting language is used be made by the MADE application developer or even the end–user. Furthermore, several scripting languages can coexist within the same MADE application (see [10]).

In order to be usable for MADE, a scripting language should have an embeddable interpreter. Ie, it should be possible to link the interpreter to C/C++ and C/C++ functions should be accessible from the language somehow. Conversely, functions of the scripting language should be accessible from C/C++. Note that the availability of the dynamic call interface of MADE objects plays an essential role in interfacing such interpreters: it is not necessary to create a special "stub" for each MADE object in the scripting language; indeed, MADE objects can be created, and their methods invoked, based only on their signature.

There are several general embedded interpreters available. Currently, the MADE toolkit includes an interface to Wool, a Lisp dialect implemented by Bull ([23]), and to Python, a language developed at CWI ([31]). In the future, interfacing to Tcl ([25]) or other emerging languages will also be considered.

4.1.3 User Interface Builder. The MADE utility workpackage also includes the definition and implementation of a user interface builder utility for UNIX platforms. This utility is based on an existing Bull product called EGERIA[4] which is to be adapted for the MADE environment in the course of the project. This utility is considered as a completely separate part of the MADE environment; it is aimed at the fast specification of the user–interface part of a MADE application and is based on the Motif toolkit.

On MS–WINDOWS environments, Visual C++[22] will be used as a user interface builder. For the integration of MADE objects and utilities, subclasses of the "Microsoft Foundation Classes" will be developed and accessible directly from Visual C++. This has already been validated with the 2D editors of GoPATH[7].

User interface builders may also be available for the scripting languages usable with MADE. In fact, EGERIA is based on Wool, and can therefore be used as a user interface generator for Wool–based applications; a similar development (being carried out independently of the MADE project) for Python may be used in later stages of the project.

4.2 Monomedia Editors

The role of monomedia editors is relatively straightforward: they offer means for the creation, modification, and also for the display of media objects. There is nothing particularly unusual or new in these utilities, except that they all abide to the architectural demands for MADE editors, as described above. Note that these editors make use of the visual metaphors described in §4.1.1 to give a unified outlook.

MADE editor objects may be used in various application settings. This includes being activated alongside with other MADE objects, eg, other editors. In this case, editor objects may be active objects, and the mechanism provided by the MADE object model will ensure that data managed by several editors will not be corrupted by concurrent access. Editors may also be wrapped up into separate application programs to run as stand–alone processes. In this case, editors may operate on MADE objects residing in a database or they can manage objects received via a communication

channel using, eg, the KEDIT protocol (see §2).

The *2D graphics editor* is based on an existing program, called `godraw` (related to GoPATH, mentioned earlier). The facilities supported by this editor are relatively straightforward, and are in line with other 2D graphics editors, available for different platforms.

The *3D graphics editor* emphasizes two aspects of 3D editing: editing of scenes by composing 3D objects in space, and simple 3D solid modelling to create 3D bodies. It includes dialogues to control attributes like texture, colour, reflectance, opacity, etc.

The *audio editor* offers facilities to "cut" and "paste" audio tracks, apply (possibly user–specified) filters on the sound tracks, and modify their characteristics. A MIDI editor will also be available.

The *video editor* offers similar facilities that of the audio editor: "cut" and "paste" of video sequences, modification of its characteristics (if the underlying hardware permits it), retrieve and frames as images, etc.

A separate *animation editor* is also provided, which allows for the interactive creation and editing of animation curves, and animation scripts.

Note that, under MS–WINDOWS, Microsoft's Multimedia Environment already contains some multimedia editors; to avoid duplication, these editors will be reused as much as possible.

4.3 Composition Utilities

Composition editing and playback is the mechanism within MADE for developing and viewing multimedia/hypermedia documents, both from the point of view of an author of such documents and also from the point of view of the final user(s) of a MADE application based on the document concept. The composition editing and playback utility is one of the main integrating components of the MADE application environment. It is through the definition of an abstract document structure that a hypermedia document is created and it is the presentation of this hyperdocument which the end user may interact with. During both the authoring and playback modes of operation the composition utility makes direct use of the other MADE utilities for viewing or editing particular media objects, for presenting help information, for navigating the hyperdocument structure, and perhaps also for monitoring the user's actions. The composition utility drives the operation of these other utilities based on a composition graph (ie, the internal representation of the hyperdocument).

An essential aspect of the composition facilities is the ability to define and manipulate an abstract document structure[3] The abstract document structure is a representation of logical components which describes not only the specific types of media involved in the presentation, but also the semantic connections between media, the synchronisation constraints associated with the presentation of the logical components, geometric and other presentation attributes for each component, and specific interaction entities to be used in reading the multimedia document.

The authoring and presentation of a hyperdocument is not only determined by the media and the composition utilities. There may be a number of alternative styles (or metaphors) for presenting a particular hyperdocument that are dependent not on the specific document itself but on the application domain in which the MADE application exists.

A specific goal of the composition utilities of MADE as a whole is to separate the presentation metaphor used for authoring and viewing a MADE hyperdocument from the underlying composition graph. The aim is to accommodate different styles of authoring and different forms of visually structuring the hypermedia information. Within the MADE project, a prototype authoring application will be developed, with a specific application area and presentation metaphor. However, this application should be considered merely as a test of the MADE concepts; it is perfectly possible for another

[3] This abstract document structure is also referred to in this specification as a composition graph.

application to choose a radically different presentation scheme and implement it on the "top" of the MADE composition utilities.

Another important aspect of the composition editing and playback facility is making provision for use of an interchange format that represents the abstract document structure in a more persistent form. An interchange format enables the reuse of existing compositions, either fully or in part, and enables the exchange of documents among MADE applications. This aspect is not straightforward, however, and there are a number of decisions to be made on which specific format should be adopted. The main contenders at the moment appear to be HyTime[13] and MHEG[14]. A third choice would be to develop a MADE specific format (temporarily denoted as MIFF), perhaps based partly on either of the above or some other less well known format. Other possibilities include the Microsoft's AVI format ([21]) and the MOVIE format defines as part of Apple's QuickTime environment ([32]). At the time of writing, the choice of the appropriate format is still to be made.

The composition utilities include some sub–modules with well specified tasks. These include:

An *interaction editor*, used to create or modify interaction objects (see §3.2.4). This involves defining sensors associated with MADE objects (or with their associated visual metaphor), specifying the objects the interaction object has to control, and editing the corresponding script. The definition and/or the modification of sensors may involve, eg, graphics editing, which means that the interaction editor may also start up a 2D graphics editor internally. In this setting, interaction objects provide a possible internal representation for hyperlinks.

The role of the *synchronization editor* is to interactively define the synchronization patterns among several synchronizable MADE objects. This may involve the specification of reference points, setting references of other object the synchronizable object has to synchronize with, defining the details of this synchronization, etc. Time objects are also managed by this editor; the user may indeed prefer to use the notions of time, scheduler, and time–constraints for the purpose of synchronization, rather than the concept of reference points. (As described in §3.2.3, both mechanisms are available within the MADE toolkit.)

The choice of the interchange format will greatly influence whether, in the synchronization editor, the emphasis will be placed on reference point on time–based synchronization. HyTime, for example, expresses all synchronizations using an abstract notion of time; quite naturally, if the HyTime format, or a subset of it, is chosen, this will determine the final shape of the synchronization editor, too.

The *graph* or *layout editor* gives a visual interface for the direct manipulation and visualization of the composition graph (ie, the hyperdocument structure).

Finally, the *composition editor* is the most complex composition utility, which combines and controls all other composition utilities as well as the monomedia editors, and MADE toolkit objects. It is this module which lies at the heart of all composition utilities, which is responsible for providing all the general functionalities described above.

5. Application Architectures

The notion of *multimedia application* is a very broad concept and application programmers may make use of a package like MADE in different ways. Also, the concept of a *user* of MADE (or of similar packages) has become a somewhat fuzzy notion; there are, in fact, different types of users (toolkit or utility developers, C++, script programmers, hypermedia document authors, etc) which are all, in some way or other, "users" of the MADE environment. Without claiming to be exhaustive, this section will give some, very typical examples of application program architectures.

Note that the full MADE ESPRIT project includes the development of some pilot applications, too. It is not the purpose of this paper to give a thorough description of the whole ESPRIT project, hence these applications are not described here. Suffice it to say, however, that the application program

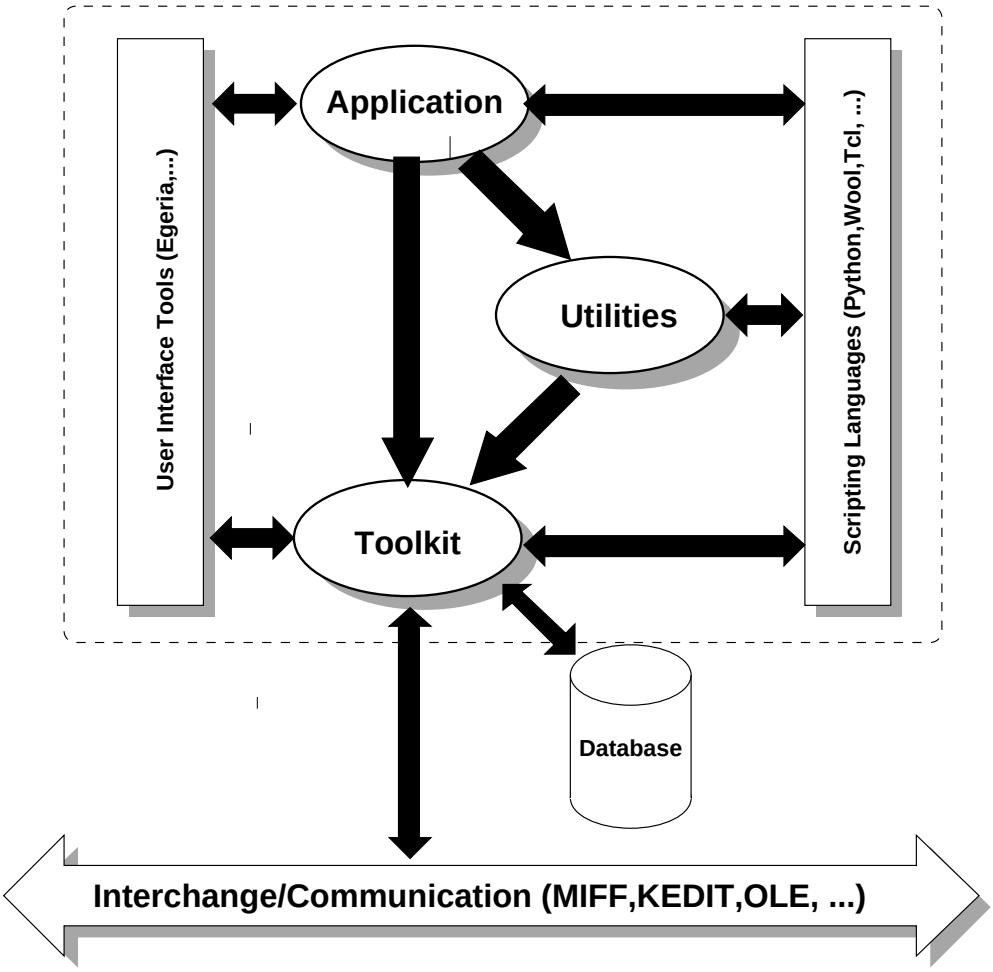

Figure 5: Traditional Programming with MADE

architectures, as presented below, are all represented in these various pilot applications.

5.1 "Traditional" Programming

The MADE toolkit objects, plus some of the utility objects, form a powerful, albeit "traditional" programming environment for C++ programmers. This means that applications may be developed in C++ or C, and then linked to a set of run–time MADE libraries.

Figure 5 (which is identical to Figure 1) gives a faithful picture of a traditional program using MADE. The application program (which is usually a single UNIX, MS–DOS, or Windows–NT task) uses different toolkit objects either directly or indirectly, via some utility objects. A more elaborate application would also make use of an external database, accessed via the MADE database object facilities.

The application program may interchange data with other applications via, eg, the MIFF exchange format. Alternatively, the application program may offer *services*, in the form of a sophisticated multimedia server, using either the KEDIT protocol or OLE. Other applications may then either directly manipulate MADE objects via this protocol or full MADE objects may be transferred back and forth, and manipulated upon, by different modules.

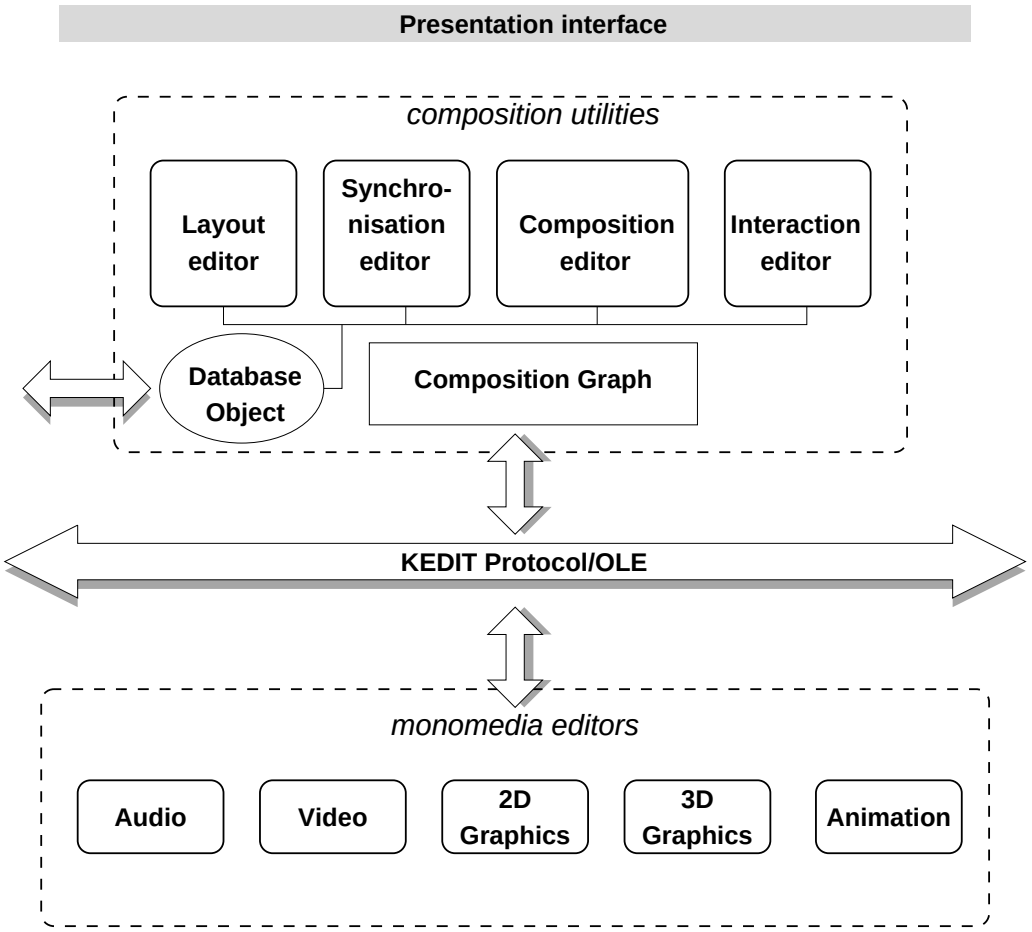

Figure 6: Composition Utilities in a MADE Application

Various objects, such as the interaction and animation objects, may use scripting languages, which may be revisable by the end–user. In fact, the skeleton of the application program may also be written in a scripting language instead of C or C++; the script would then manipulate MADE objects (written in mC++) via the appropriate MADE–script interpreter interface.

Another possibility is to use C++ and, eg, Motif to create the user–interface; this is when a graphics user interface application builder, like EGERIA, or Visual C++, may play an important role.

5.2 Hyperdocument Editing and Playback

Figure 6 illustrates a possibility for hypermedia document manipulation using the full–blown composition utilities described in §4.3. The programming environment offered by MADE in this setting is hypermedia document authoring; quite naturally, the user community for such an environment differs radically from the community of "traditional" programmers. (Very often, to make the distinction, members of this community are referred to as "authors", as opposed to "users".)

In this authoring environment, the composition utilities are conceptually separate from the media editors. The composition utilities act as the coordinating components of the complete architecture. Effectively, there is an inter–editor message facility that is used to both control the operation of the media editors and to provide information to the composition utilities representing actions performed by

the user through dialogues with the media editors. In this setting the media editors may be considered as separate applications or, in other terms, as separate service providers. These applications may be realized following the scheme described in the previous section.

This organisation implies that media objects or references to objects are passed between the composition utility and the media editors in order to "render" them. Similarly, edited media objects may need to be passed back to the composition editor and placed into the multimedia database.

Note that a simpler version of the architecture, including a simpler version for each of the media editors, may be defined to be used for "playback" only.

5.3 Other Application Schemes

The application architectures presented in the preceding two sections represent, in a way, the two extremes of a large palette. Intermediate architectures, making use of only part of the full MADE functionality are also possible and feasible. It is possible to create, for example, a HyTime–like engine based on the MADE toolkit and some of the utilities only (although these utilities may be distributed services rather then linked to the HyTime engine)[4]; interactive modelling applications, or scientific visualization applications, are also possible, which may use the services of media editors, just as a full hypermedia authoring tool does, but with a fundamentally different user–interface.

The application architecture shown on Figure 7 illustrates another possibility for an authoring environment. As said earlier, media editors, realized as MADE applications, may be used as independent servers, provided that the external communication protocol is understood by the "wrapper" around the MADE editor objects. In such a case, an "external" (ie, not closely MADE dependent) hyperdocument authoring tool may be used instead of the MADE composition utilities. The example used in Figure 7 is HyperPATH, formerly known as Multicard ([26]), a hypermedia editing tool developed by Bull. (The M2000 protocol referred to in the figure is the internal communication protocol defined for HyperPATH.)

6. Standardization

In a somewhat unexpected way, activities in the MADE project have become very much relevant recently for an ongoing standardization process within ISO. Indeed, after several years of preparations, the ISO committee ISO/IEC JTC 1/SC 24 (the committee which developed graphics standards in the past) has decided to engage into a project for the standardization of a presentation environment for multimedia programming. The scope and purposes of this new project, called PREMO[15] are indeed very close to the project specifications of MADE: an object–oriented presentation environment for multimedia objects, including graphics, video, audio, etc., which incorporates specific means for the synchronization, interaction, and combination of such media.

Fortunately for the MADE project (and, hopefully, for the PREMO project, too), contacts between MADE project members and the relevant ISO committee could be set up very quickly, due to some earlier ISO activities of several participants of the MADE project. Concepts developed within the MADE project have been included into the PREMO activities, and, conversely, some of the issues that have arisen at the PREMO meetings have provided valuable input in the design work of MADE. It can be expected that this fruitful interaction will help to shape the outcome of the MADE project in the future, too.

[4] In fact, creation of an engine for a specialized set of HyTime documents is part of the full ESPRIT project.

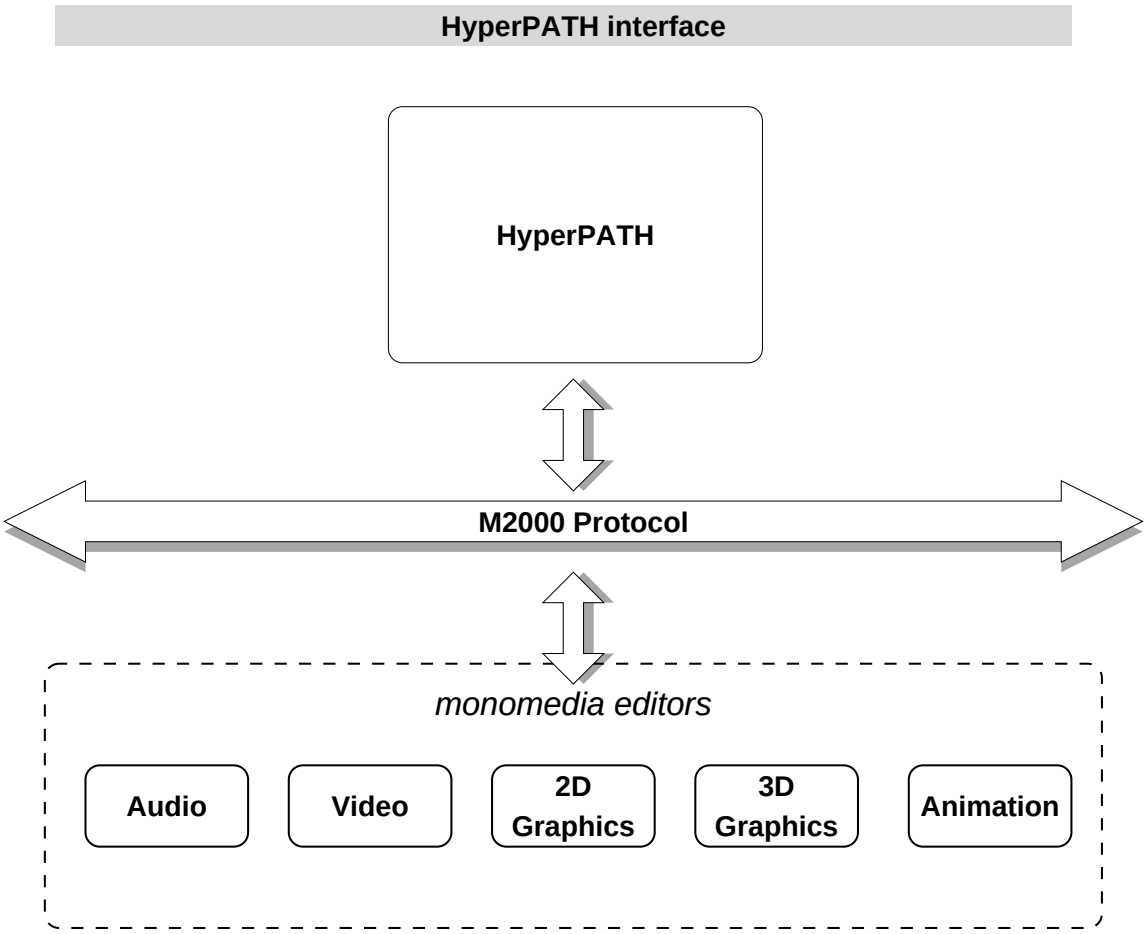

Figure 7: Usage of an External Composition Tool: HyperPATH

Acknowledgements

Obviously, MADE is a large–scale teamwork project, involving experts from a number of industrial and academic institutions[5]. Although only some of the partners are involved in the specification details of the MADE framework (others being responsible for the pilot applications), the team of experts is still rather voluminous. Instead of trying to list everybody and thereby incurring the danger of forgetting, and therefore offending, somebody, we prefer to omit such a long list. We would just like to express our gratitude to the full MADE team altogether.

References

1. F. Arbab, I. Herman, and G.J. Reynolds. An object model for multimedia programming. *Computer Graphics Forum (Eurographics'93 Conference Issue)*, 12(3), September 1993.
2. F. Arbab, P.J.W. ten Hagen, M. Haindl, F.C. Heeman, I. Herman, G.J. Reynolds, and A. Siebes. Specification of the MADE object model. Technical Report T/OM S1, Version 0.5, Esprit Project 6307 (MADE), March 1993.
3. G. Blakowski, J. Hübel, and U. Langrehr. Tools for specifying and executing synchronized multimedia presentations. In R. G. Herrtwich, editor, *Second International Workshop on Network and*

[5] Namely: Groupe Bull (France), CWI (The Netherlands), INESC (Portugal), INRIA (France), FhG-IAO (Germany), BaE (UK), NR (Norway), ESI (France), Iselqui (Italy).

Operating System Support for Digital Audio and Video, number 614 in Lecture Notes in Computer Science, pages 271–282, Heidelberg, 1992. Springer Verlag.

4. V. Bouthors. *Egeria Reference Manual*. Bull SA, Paris, version 2.1 edition, August 1992.
5. D. Carver. X video extension protocol, version 2. Technical report, DEC Technical Report, MIT X11 Contributions, 1991.
6. W. Clifford, J.I. McConnell, and J. Saltz. The development of PEX. In D.A. Duce and P. Jancène, editors, *Eurographics'88 Conference Proceedings*, Amsterdam, 1988. North–Holland.
7. 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.
8. N. Guimarães and N. Correia. Specification of the MADE time objects. Technical Report T/TO S0, Esprit Project 6307 (MADE), June 1993.
9. M. Haindl, I. Herman, and G.J. Reynolds. Presentation scheme — preliminary specification. Technical Report T/PRS S0, Version 0.1, Esprit Project 6307 (MADE), July 1993.
10. I. Herman, F.C. Heeman, and F. Leygues. Interfacing scripting languages. Technical Report Version 1.3, Esprit Project 6307 (MADE), June 1993.
11. I. Herman, F.C. Heeman, and G.J. Reynolds. Interaction objects — functional specification. Technical Report T/IAO S1, Version 0.2, Esprit Project 6307 (MADE), June 1993.
12. T.L.J. Howard, W.T. Hewitt, R.J. Hubbold, and K.M. Wyrwas. *A Practical Introduction to PHIGS and PHIGS PLUS*. Addison–Wesley, Workingham – Reading, 1991.
13. Internation Standard Organization. *Information Technology — Hypermedia/Time–based Structuring Language (HyTime), ISO/IEC 10744:1992(E)*, 1992.
14. International Standard Organization. *Coded Representation of Multimedia and Hypermedia Information Objects (MHEG)*, ISO/IEC JTC 1/SC 29 N354 edition, February 1993.
15. International Standard Organization. *Presentation Environment for Multi–Media Objects (PREMO); Initial Draft ISO/IEC JTC 1 SC 24 WG 6 OME 35*, June 1993.
16. J. Davy (ed.), Paris. *MADE 1, ESPRIT III Project 6307, Technical Annex*, March 1992.
17. O. Jojic and J. Davy. C++ API implementation. Technical Report T/OM-C++/P.0, Esprit Project 6307 (MADE), July 1993.
18. P. Kaplan and A. Baird-Smith. The KEDIT protocol. Technical Report U/UIE-KEDIT/S.0, Esprit Project 6307 (MADE), July 1993.
19. T.M. Levergood, A.C. Payne, J. Gettys, W. Treese, and L.C.S Steward. AudioFile: A network–transparent system for distributed audio applications. Technical Report CLR 93/8, Digital Equipment Corporation, Cambridge Research Laboratory, Cambridge, MA, June 1993.
20. 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*, pages 214–223, Portland, September 1986. ACM Press.
21. Microsoft Inc. *AVI — Microsoft Technical Note*, November 1992.
22. Microsoft Inc. *Users' Guides, Microsoft Visual C++, Development System for Windows, Version 1.0*, 1993.
23. C. Nahaboo. *Koala Project, Wool2 Reference Manual, V2.3 Beta3*. Groupe Bull, Paris, November 1992.
24. Object Management Group. *The Common Object Request Broker: Architecture and Specification; OMG Document Number 91.12.1, Revision 1.1*, 1991.

25. J.K. Ousterhout. *An Introduction to Tcl and Tk*. University of California, Berkeley, October 1992.

26. A. Rizk and L. Sauter. Multicard: An open hypermedia system. In *European Conference on Hypertext ECHT'92*, Cambridge, 1992. Cambridge University Press.

27. P.S. Strauss and R. Carey. An object-oriented 3D graphics toolkit. *Computer Graphics (SIGGRAPH'92)*, 26(2):341–349, July 1992.

28. B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, Massachusetts, second edition, 1991.

29. F. van Dijk and A. Siebes. Specification of the database object. Technical Report T/DBO S1, Version 0.1, Esprit Project 6307 (MADE), June 1993.

30. J.E.A. van Hintum and G.J. Reynolds. Constraint objects. Technical Report T/COO S0, Version 0.1, Esprit Project 6307 (MADE), June 1993.

31. G. van Rossum. *Python Reference Manual*. Centrum voor Wiskunde en Informatica, Amsterdam, July 1993.

32. P. Wayner. Inside QuickTime. *BYTE*, pages 189–197, December 1991.