CWI Quarterly Volume 7 (1) 1994, pp. 27 – 46

# MADE: A Multimedia Application Development Environment

I. Herman, G.J. Reynolds
*CWI*
*P.O. Box 94079, 1090 GB Amsterdam, The Netherlands*

J. Davy
*Groupe Bull*
*7, rue Ampère, Massy 91343, France*

MADE is the acronym for an ESPRIT III project aimed at developing a programming environment for multimedia applications. The resulting software library is based on C++ and will operate on both UNIX workstations and PC–based platforms. This paper gives a technical overview of the project and describes a number of application scenarios where the MADE environment will provide significant help for multimedia programming.

## 1. Introduction

The emergence of multimedia is one of the most significant developments in computing technology in recent years. Glossy multimedia applications are frequently demonstrated, often on a range of platforms. The major workstation hardware vendors feel the need to come to technical fairs with impressive displays that mix graphics, video, imaging, and sound. Technology analysts predict that multimedia related hardware development will be one of the most important boom areas of electronics in the years to come.

However, the majority of available multimedia environments aim at *hypermedia authoring*, i.e., they offer the means to interactively create hypermedia documents. We use the term "document" as a multimedia term, hence, implying 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 a hypermedia document is a powerful one, it does not cover all applications of multimedia. The ability to combine, modify, or even synthesise multimedia data is often necessary for more complex multimedia applications. For example, a user might wish to extract a frame from a video sequence, modify it with standard image processing tools, combine the image with some synthetic graphics, and then exchange the original frame with the modified image. The description of such actions does not fit easily within the model of a hypermedia document, in spite of the sophisticated interaction tools which are often provided as part of authoring environments. We conclude, therefore, that there is a clear need for a programming environment which allows for and actively supports the development of such applications.

Techniques for combining media are extremely disparate and use results from various fields of computing technology, such as, high quality synthetic graphics, image processing, speech synthesis, etc. Some of these techniques are also highly application dependent. Consequently, it is almost impossible to define a closed programming environment which encompasses all techniques and dependencies. The "traditional" answer to this kind of challenge is to use object–oriented techniques: services are offered in the form of objects, which can be extended by the programmer to include any necessary application–dependent tools.

The European Communities' ESPRIT III project MADE (**M**ultimedia **A**pplication **D**evelopment **E**nvironment[1]) has the ambitious goal of defining and implementing a portable object–oriented development environment for multimedia applications. The outcome of the MADE project will be a programming environment, based on C++, running on various UNIX platforms, as well as on MS–WINDOWS environments. This paper gives a technical overview of MADE: it describes its major services and a number of "application scenarios" which make significant use of these services. It is not the purpose of the paper to give a detailed description of the complete project; that would go far beyond the paper's scope. The interested reader should consult the "official" MADE documents to gain a more detailed insight [1, 2, 3, 4, 5, 6].

It should be emphasized that MADE is primarily a development project, with some research elements in it. This paper gives an overview of the project as whole, whereas the more research-like results (e.g., object model, interaction, etc.,) are presented elsewhere [7, 8]. Note also that the MADE project is still an ongoing activity. Consequently, some problems are still open and will be solved later in the project. Consequently, this paper sometimes raises issues without presenting complete solutions.

## 2. General overview

The full MADE environment contains a large number of different objects and related services. The majority of these fall into two important categories, namely: *toolkits* and *utilities*. (Note that the object–oriented nature of MADE makes it possible for an end–user to add new objects to both toolkits and utilities and to extend the functional capability of existing ones.)

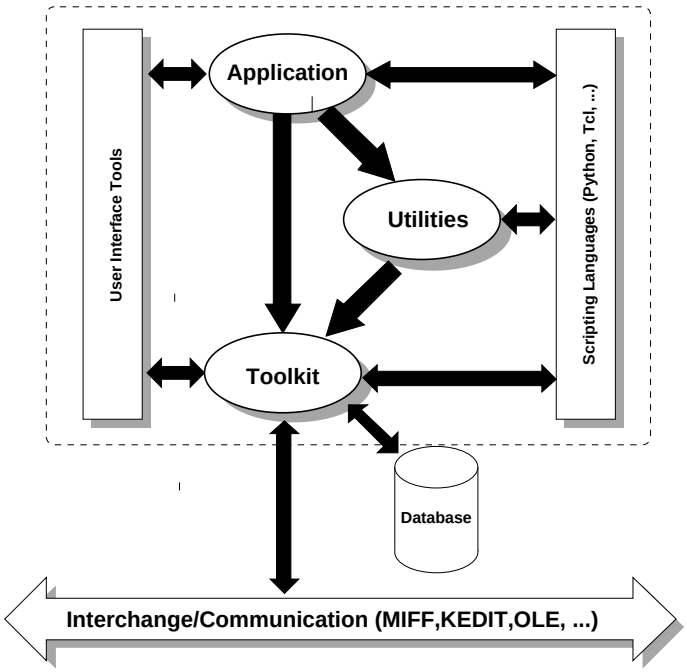


FIGURE 1. Toolkits and Utilities

The *toolkit* category (or level) represents a collection of objects that are considered to be fundamental to multimedia programming. It includes objects that interface with different media. It also includes objects which, although not directly involved in handling specific media, play a fundamental role in constructing more complex multimedia applications. Some details of the toolkit level will be given below (see Section 3).

Although it is possible to construct complex applications using the MADE toolkit level only, doing so may be unnecessarily tedious and error–prone. Consequently, the *utilities* level has been defined on top of the MADE toolkit. This level includes objects which implement more complex functionality and which are considered to be essential for most multimedia applications. Application programmers may choose to use some of these utility objects; however, the toolkit level is never completely obscured, and an application is free to make direct use of toolkit objects if necessary (see Figure 1; some of the terms appearing on the Figure will be described in later sections).

A common *object model* was defined and developed at an early stage of the MADE project to ensure the smooth cooperation between objects in the MADE library and to provide a clear conceptual 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 (i.e., C++), 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*, that is, they can have their own thread of control (within the shared address space of the same UNIX or MS–WINDOWS process). This capability is exploited by the implementation of the MADE toolkit, and is a major tool used in defining synchronisation among media (see §3.2.3 below). Application programmers have to be aware of this situation if they decide to use the toolkit level objects directly.

The concept of *delegation* in the MADE object model applies to an object's methods. 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 then act on its behalf. The notion is not unlike inheritance, but delegation is dynamic, i.e., the target of delegation may be set/re–set at run–time. Delegation plays an important role in establishing constraints in MADE (constraint objects are part of the toolkit), and offers an advanced means of temporal behavioural control. A more exact semantics of delegation is described in [9]; see also [7] for a fuller description of the concept within the framework of the MADE object model.

The object model is realised in the form of an extension of C++, called mC++. The mC++ translator generates a set of C++ classes, as well as library and macro calls; this "intermediate" level can also be accessed by programmers directly if they do not wish to use yet another programming language. Details of the object model are normally hidden to most application programmers and are only of real interest to toolkit and utility developers. The full technical description of this object model is omitted here; refer to [7] for a general overview and to [2] for a complete description of the model and 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 are given below.

All objects in the MADE system can be *persistent*. 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 application. 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 acts as an interface to various database systems. Although this interface does not 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 are included in all MADE objects in a database–independent way, and the specific method of database access is hidden by the general database management object (see [10]). Interfacing to a new database system is achieved by specialisation of the general database

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[11]) 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 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–WINDOWS platforms the OLE protocol will be used to provide similar facilities; this is already an integral part of these environments.

All MADE objects include a general mechanism known as a "dynamic call interface". This interface makes it possible to call an object's member functions, knowing the object's handle and a *string* description of a member function's signature. 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.

## 3. Toolkit objects

The primary goal of the MADE toolkit is the provision of a basic set of features and facilities for multimedia programming. This includes control over different media, as well as other types of objects that have been identified as fulfilling a fundamental role.

### *3.1. Media objects*

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

The toolkit includes four main categories of media objects: graphics objects (for two and three dimensional graphics), animation, audio and video objects. 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. The abstractions for the various media objects have been developed within

[1] OMG is an industrial consortium aiming at the definition of object services in general. In their CORBA specification[12], 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.

the project; however, existing technology practice has greatly influenced our design.

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 synchronisation. The objects and their device specific interfaces must be matched with the synchronisation 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 video ([13]) and the AudioFile server for audio ([14]).

2D and 3D graphics requires a greater degree 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 existing object–oriented 2D graphics system, called GoPATH[15], by adapting it to the requirements of MADE. These 2D objects include various shapes, associated clipping areas, composition rules, attributes, 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.

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 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 project to define yet another 3D graphics package; the emphasis is more to provide an object–oriented layer on top of existing packages which is fully 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 (e.g., to add a proprietary ray–tracing module) if necessary.

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 audio and video 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, are dependent on specific application areas. It is not the purpose of the MADE animation objects to encompass all available animation techniques. Although simpler, built–in, animation techniques based on animation curves are available, MADE animation objects support animations defined as scripts, using a scripting language, which are then interpreted by the animation objects. Animation objects are active objects, and are thereby subject to the same synchronisation behaviour and control as audio and video objects (see §3.2.3).

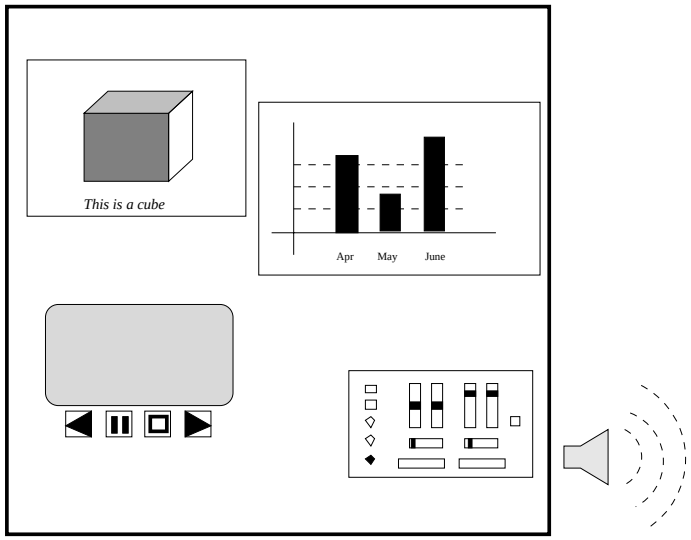


FIGURE 2. A rudimentary example for multiple media in an application.

*3.2. Combination objects*

The MADE toolkit level object represent a relatively low functionality level. It is of course clearly possible to build impressive applications that rely solely on MADE media objects, using complex static and animated graphics, running a video on the screen and playing audio, etc. However, an additional level of functionality is necessary when more complicated application programs have to be devised and implemented. The very rudimentary example on Figure 2 already demonstrates that: interactive behaviour assigned to graphics objects has 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.

Basic media objects become really usable if they can be *combined* easily in a variety of ways. The combination of media objects (and MADE objects in general) within an application has received particular emphasis in the specification of the project in order to enhance the usability of the MADE tools. Five major areas of combination have been identified: *imaging*, *structuring*, *synchronisation*, *interaction*, and *constraint management*. Each of these will be looked at in the following sections.

*3.2.1. Imaging*

Video objects, 2D and 3D graphics objects, and imported image files, may all be visualised on the display screen. Very often, an application may require such "images" to be combined in some way. For example, a complex picture might be created by combining a snapshot of a video sequence, some annotated 2D text, and an imported image used to selectively filter the result.

MADE supports this kind of combination via an *image object*. All MADE media objects that produce displayable output can be directed to produce image objects. These can of course be presented, but they can also be converted into video frames or stored in a particular file format.

*3.2.2. Structuring*

The importance of *structuring*, i.e., of creating aggregates of different objects in interactive programs, has long been recognised in computer graphics. The majority of graphics packages provide some form of aggregation, such as structures in PHIGS, the scene database of IRIS Inventor, or the Go trees in GoPATH[15]. Although the structures used in these examples are relatively simple (directed acyclic graphs or trees), the appearance of hypertext and hypermedia systems makes it clear that more general aggregation facilities are necessary.

The MADE toolkit answers this requirement by including a general graph management facility. *Graph objects* are provided to support the specification, management, and traversal of general graphs, with no restrictions imposed on their types. (Nodes of these graphs may refer to any MADE object).

Graph objects provide a sound basis for the structuring required by graphics as well as for complex hypermedia navigation systems. They are fully integrated into the MADE environment, which has a number of advantages. For example, graphs provide an automatic protection against uncontrolled concurrent access of structures by active objects, they can be exported and imported using the same persistency mechanism as is defined for all other MADE objects (i.e., complete graph structures can be stored in databases), etc. It is then the role of application programs and/or higher level MADE components (like the the so-called Composition Utilities, see §4.3 below) to model the notions of links and anchors, using interaction and the graph objects.

*3.2.3. Synchronisation*

Synchronisation has always been one of the central problems of multimedia applications and the MADE toolkit offers a consistent solution to this issue.

The fundamental synchronisation scheme used in MADE is called *reference point* synchronisation. For each so–called, *synchronisable* object, a series of media specific reference points can be defined (for example, video frames, audio samples, etc.). Each reference point contains internal "instructions" for synchronisation and references to other synchronisable objects with which they are to be synchronised. Synchronisable objects are active objects: when they reach a reference point, synchronisation is performed by exchanging messages with other active objects, waiting for their replies, etc. The reference point model has been inspired by [16]; its details in the MADE environment are specified in [3].

Audio, video, and animation objects are obvious examples of synchronisable objects[2]. A MADE programmer may also create new, application–specific,

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

synchronisable objects.

The MADE toolkit also includes a higher–level mechanism for time–based synchronisation, based ultimately on the reference point model. This mechanism defines different types of *schedulers* that an application may use as building blocks for more complex time–based synchronisation scenarios (see [3] for further details). These schedulers all assume the existence of a special synchronisable object within MADE, namely a *timer*. The approach of building time–based synchronisation 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 offer real–time facilities.

*3.2.4. Interaction objects*

Multimedia applications are very often highly interactive; it is therefore essential to have tools to support the construction of complex interaction scenarios.

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. Nevertheless, not all user interaction can be adequately managed by these tools; many complex interaction scenarios will involve MADE objects directly (e.g., 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 must instead be attached to a visual representation of the object in the form of graphics object. This might be the case for 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 initialisation procedure to decide which interaction object it is connected to. The interaction objects react to these events by following a pattern of behaviour that is defined as part 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 (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 (e.g., the video control board depicted in Figure 2 reacts on the sensors of the graphical objects

objects in general.

describing the four push–buttons, may control the visual appearance of these buttons and, of course, controls the video object proper or perhaps a combination of objects; see also Figure 3). To describe such complex interaction behaviour, MADE introduces a type of interaction object that implements a general finite state machine (see [8]). 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). The assignment of 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 would justify a development project in its own right. Fortunately, there are restricted types of constraint satisfiers that, while not being universal, still provide useful functionality for dealing with important categories of constraints.

The approach followed in the specification of constraints is to consider those applications of constraint systems that are of direct relevance to the multimedia part of MADE [6]. 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 can make use of constraints when defining the hypermedia document structure and presentation characteristics.

For the time being, 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.

## 4. Utilities

*Utilities* offer developers a higher level of functionality that simplifies the implementation of both basic and more complex multimedia applications. In fact, the functionality of some of the utilities is such that, by "wrapping" them into a simple program, they can be used as a separate applications in their own right.

There are four main utility categories:

1. *application program interface utilities*: user interface 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, synchronisation editors, interaction and graph object editors;
4. *miscellaneous*: class browsers, generic on–line help facilities, object monitoring.

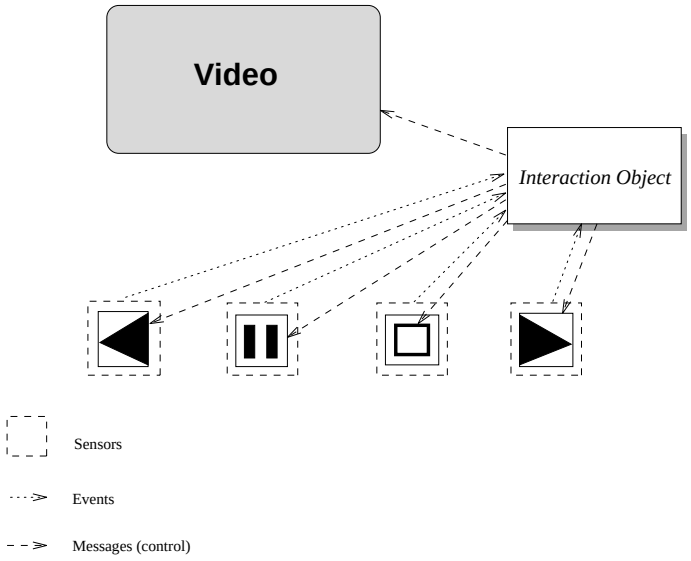


FIGURE 3. Use of Interaction Objects.

The different MADE utilities may rely on one another. For example, the user interface metaphors (§4.1.1) are reused by monomedia editors (§4.2) and the composition utilities (§4.3).

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 these architectures will be described in §5 below.

*4.1. Application program interface utilities*

Application program interface utilities consist of a set of tools that help an application programmer to prototype and develop a final MADE application. Although the facilities provided by some of these utilities are not new, it is necessary to provide them in the context of the MADE environment. Note that only the more important tools are presented in this paper.

*4.1.1. User interface 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 behaviour have to be developed. These tools can be used on different levels: in program development, in authoring, or in the final playback of authored documents. These user interface metaphors play an essential role in defining complex interactions operating on the objects. It may sometimes be 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 user interface metaphors, including:

- 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 *user interface 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 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 in final form, where the "user–level" program is a script.

MADE does not introduce its 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 specialised 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 [5]).

In order to be usable for MADE, a scripting language should have an embeddable interpreter. That is, 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 embeddable interpreters available. Currently, the MADE toolkit includes an interface to Tcl ([17]), and to Python, a language developed at CWI ([18]).

*4.1.3. User interface builder*

The MADE utilities workpackage includes a prototype authoring toolset that incorporates a user interface builder for use on UNIX platforms. This based on an existing tool that combines Tcl and Motif, extended to included specific user interface entities for multimedia applications. A similar development (being carried out independently of the MADE project) for Python may be used in later stages of the project.

On MS–WINDOWS environments, Visual C++ 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 accessed directly from Visual C++. This has already been validated for the 2D editor of GoPATH[15].

*4.2. Monomedia editors*

The role of monomedia editors is relatively straightforward: they offer the means to create, modify, and display 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 and they incorporate the notion of configurability. Each monomedia editor is able to be configured at start up in one of a few modes of operation. For example, its possible to configure an editor for use as a player-only tool. This mechanism is used extensively by the composition utilities (see §4.3). Note that MADE 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, e.g., other editors. In this case, editor objects may be active objects; 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. Here, editors may operate on MADE objects residing in a database or they can manage objects received via a communication channel using, e.g., the KEDIT protocol (§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* emphasises 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 to that of the audio editor: "cut" and "paste" of video sequences, modification of its characteristics (if the underlying hardware permits it), retrieve and replace 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 (i.e., 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 and interacting with 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 tool is being developed with a specific presentation metaphor. It will, however, be possible for another authoring tool to choose a radically different presentation scheme and implement it on the "top" of the MADE composition utilities.

The composition utilities also make provision for using an interchange format to represent 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. There are a number of contenders at the moment, HyTime[19] and MHEG[20] are standardised formats, and there are a other proprietary ones. A third choice would be to develop a MADE specific format (temporarily denoted as MIFF),

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

perhaps based partly on either of the above or some other industry format. At the time of writing, Apple's BENTO[21] format has provisionally been chosen for use as the MADE interchange format (MIFF).

The composition utilities include some sub–modules with well specified tasks. These include facilities for interaction, synchronisation, layout and composition.

An *interaction editor* creates or modifies 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, e.g., graphics editing, which means that the interaction editor may also start up a 2D graphics editor internally. In this setting, interaction objects could provide an internal representation for hyperlinks.

The role of the *synchronisation editor* is to interactively define the synchronisation patterns among several synchronisable MADE objects. This may involve the specification of reference points, setting references of other object the synchronisable object has to synchronise with, defining the details of this synchronisation, 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 synchronisation, rather than the concept of reference points. (As described in §3.2.3, both mechanisms are available within the MADE toolkit.)

Inside the synchronisation editor, the choice of the interchange format will greatly influence whether the emphasis will be placed on reference point or time–based synchronisation. HyTime, for example, expresses all synchronisations 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 synchronisation editor, too.

The *graph* or *layout editor* gives a visual interface for the direct manipulation and visualisation of the composition graph (i.e., 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, and 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, typical examples of application program architectures.

Note that the full MADE ESPRIT project includes the development of some pilot applications. 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 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 1 gives a faithful picture of a traditional program using MADE. The application program (which is usually a single UNIX, or MS–DOS 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, e.g., 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 be manipulated by different modules.

Various objects, such as the interaction and animation objects, can 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, e.g., Motif to create the user–interface; this is when a graphics user interface application builder may play an important role.

*5.2. Hyperdocument editing and playback*

Figure 4 illustrates the possibility of 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 central hub 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

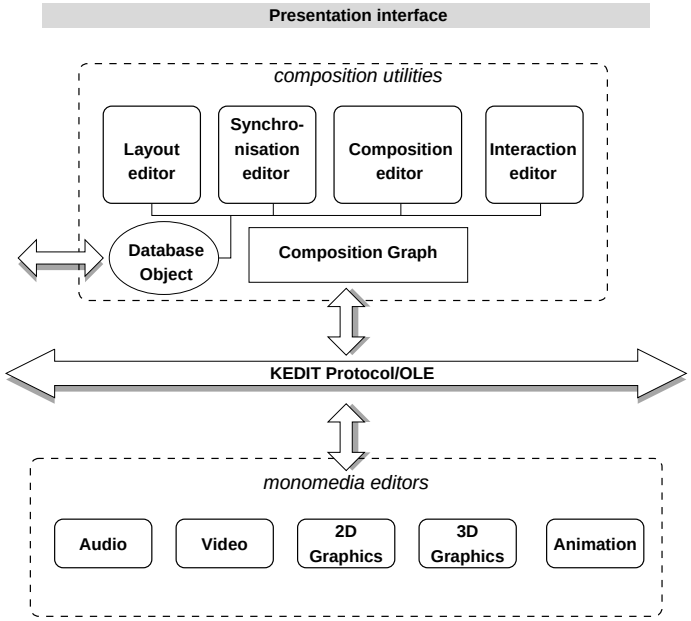


FIGURE 4. Composition Utilities in a MADE Application

the media editors may be considered as separate applications or, in other terms, as separate service providers. These applications may be realised 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 configured 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 visualisation applications, are also possible, which may use the services of media editors, just as a full hypermedia authoring

[4] In fact, creation of an engine for a specialised set of HyTime documents is one of the pilot applications that is part of the full ESPRIT project.

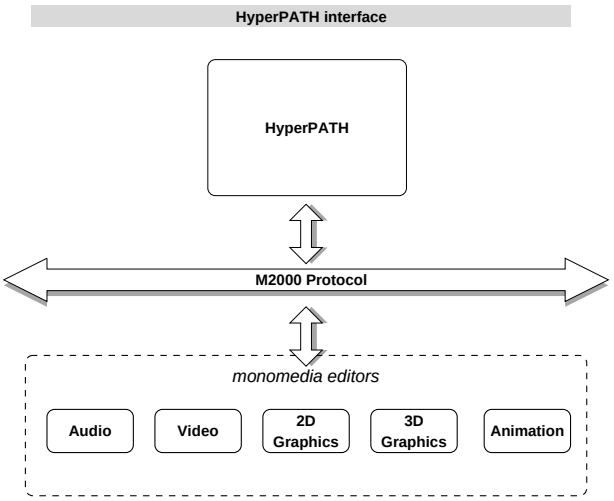


FIGURE 5. Use of an External Composition Tool: HyperPATH

tool does, but with a fundamentally different user–interface.

The application architecture shown on Figure 5 illustrates another possibility for an authoring environment. As said earlier, media editors, realised 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" (i.e., not closely MADE dependent) hyperdocument authoring tool may be used instead of the MADE composition utilities. The example used in Figure 5 is HyperPATH, formerly known as Multicard ([22]), a hypermedia editing tool developed by Bull. (The M2000 protocol referred to in the figure is the internal communication protocol defined for HyperPATH.)

## 6. STANDARDISATION

In a somewhat unexpected way, activities in the MADE project have become very much relevant recently for an ongoing standardisation process within ISO. Indeed, after several years of preparations, the ISO/IEC committee JTC 1/SC 24 (the committee which developed graphics standards in the past) has engaged into a project for the standardisation of a presentation environment for multimedia programming. The scope and purposes of this new project, called PREMO[23] 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 synchronisation, 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.

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. J. DAVY (ed.) (1992). Paris, *MADE 1, ESPRIT III Project 6307, Technical Annex.*
2. F. ARBAB, P. TEN HAGEN, M. HAINDL, F. HEEMAN, I. HERMAN, G. REYNOLDS and A. SIEBES (1993). Specification of the MADE object model, Tech. Rep. T/OM S1, Version 0.5, Esprit Project 6307 (MADE).
3. N. GUIMARÃES and N. CORREIA (1993). Specification of the MADE time objects, Tech. Rep. T/TO S0, Esprit Project 6307 (MADE).
4. M. HAINDL, I. HERMAN and G. REYNOLDS (1993). Presentation scheme — preliminary specification, Tech. Rep. T/PRS S0, Version 0.1, Esprit Project 6307 (MADE).
5. I. HERMAN, F. HEEMAN and F. LEYGUES (1993). *Interfacing scripting languages*, Tech. Rep. Version 1.3, Esprit Project 6307 (MADE).
6. J. VAN HINTUM and G. REYNOLDS (1993). *Constraint objects*, Tech. Rep. T/COO S0, Version 0.1, Esprit Project 6307 (MADE).
7. F. ARBAB, I. HERMAN and G. REYNOLDS (1993). An object model for multimedia programming, *Computer Graphics Forum (Eurographics'93 Conference Issue)*, vol. 12, pp. C101–C114.
8. F. HEEMAN, I. HERMAN and G. REYNOLDS (1994). Interaction objects in the MADE multimedia environment. In *Proceedings of the 1st Eurographics Symposium on Multimedia*, (Graz), Springer Verlag.
9. H. LIEBERMAN (1986). 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.
10. F. VAN DIJK and A. SIEBES (1993). *Specification of the database object*, Tech. Rep. T/DBO S1, Version 0.1, Esprit Project 6307 (MADE).

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

11. P. KAPLAN and A. BAIRD-SMITH (1993). *The KEDIT protocol*, Tech. Rep. U/PAT/KED/P.0, Esprit Project 6307 (MADE).
12. OBJECT MANAGEMENT GROUP (1992). *The Common Object Request Broker: Architecture and Specification.* OMG Document Number 91.12.1, Revision 1.1.
13. D. CARVER (1991). *X video extension protocol*, version 2, DEC Technical Report, MIT X11 Contributions.
14. T. LEVERGOOD, A. PAYNE, J. GETTYS, W. TREESE and L. STEWARD (1993). *AudioFile: A network–transparent system for distributed audio applications*, Tech. Rep. CLR 93/8, Digital Equipment Corporation, Cambridge Research Laboratory, Cambridge, MA.
15. J. DAVY (1991). 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.*
16. G. BLAKOWSKI, J. HÜBEL and U. LANGREHR (1992). Tools for specifying and executing synchronized multimedia presentations. In *Second International Workshop on Network and Operating System Support for Digital Audio and Video* (R. G. HERRTWICH, ed.), no. 614 in Lecture Notes in Computer Science, (Heidelberg), pp. 271–282, Springer Verlag.
17. J. OUSTERHOUT (1992). *An Introduction to Tcl and Tk.* University of California, Berkeley.
18. G. VAN ROSSUM (1993). *Python Reference Manual.* Centrum voor Wiskunde en Informatica, Amsterdam.
19. INTERNATIONAL STANDARD ORGANIZATION (1992). *Information Technology — Hypermedia/Time–based Structuring Language (HyTime), ISO/IEC 10744:1992(E).*
20. INTERNATIONAL STANDARD ORGANIZATION (1993). *Information Technology — Coded Representation of Multimedia and Hypermedia Information Objects (MHEG), ISO/IEC CD 13522.*
21. J. HARRIS and I. RUBEN (1993). *Bento Specification.* Apple Computer. Inc., revision 1.0d5.
22. A. RIZK and L. SAUTER (1992). Multicard: An open hypermedia system. In *European Conference on Hypertext ECHT'92*, (Cambridge), Cambridge University Press.
23. INTERNATIONAL STANDARD ORGANIZATION (1993). *Presentation Environment for Multimedia Objects (PREMO), Initial Draft ISO/IEC JTC 1 SC 24 WG 6 OME 35.*