Programming Paradigms in an Object-Oriented Multimedia Standard

D. J. Duke† and I. Herman‡

† Department of Computer Science, The University of York, Heslington, York, YO1 5DD, UK
‡ Centrum voor Wiskunde en Informatica (CWI), Kruislaan 413, 1098 SJ Amsterdam, The Netherlands

Abstract

Of the various programming paradigms in use today, object-orientation is probably the most successful in terms of industrial take-up and application, particularly in the field of multimedia. It is therefore unsurprising that this technology has been adopted by ISO/IEC JTC1/SC24 as the foundation for a forthcoming International Standard for Multimedia, called PREMO. Two important design aims of PREMO are that it be distributable, and that it provides a set of media-related services that can be extended in a disciplined way to support the needs of future applications and problem domains. While key aspects of the object-oriented paradigm provide a sound technical basis for achieving these aims, the need to balance extensibility and a high-level programming interface against the realities of efficiency and ease of implementation in a distributed setting meant that the task of synthesising a Standard from existing practice was non-trivial. Indeed, in order to meet the design aims of PREMO is was found necessary to augment the basic object infrastructure with facilities and ideas drawn from other programming paradigms, in particular concepts from constraint management and dataflow. This paper describes the important trade-offs that have affected the development of PREMO and explains how these are addressed through the use of specific programming paradigms.

Keywords: multimedia, object-oriented systems, extensibility, middleware, standards.

1. Introduction

The availability of the Internet as a viable basis for developing distributed computing applications, and the recent development of languages and systems (for example Java and VRML) for utilising this resource are bringing changes to the function, organisation, and design of the middleware systems that provide support for graphics and multimedia applications. Comprehensive, self contained standards such as GKS and PHIGS are being augmented by systems that are designed specifically as a set of components that can be instantiated to the needs of a given application domain, and can be extended systematically to support new application areas. Early examples of this direction can be seen in work on graphics systems such as Doré[1] and, more recently, OpenInventor[2]. In the case of multimedia, the work on MADE[3] or MET++[4] can also be considered as typical examples.

Despite the diversity of application areas, many of these new systems share a common theme: the use of object orientation as the underlying software architecture or technology. Initially, this trend may have just reflected the emergence of object-oriented design and programming into general software development. However, it has been timely in that the concept of objects with local state communicating via message passing provides good architectural support for the development of distributed systems. When considering the design of the next generation of graphics standards, object orientation was seen by ISO/IEC JTC1/SC24 (the ISO subcommittee responsible for the development of graphics and image processing standards) as both a necessary and desirable foundation for making a system that was extensible and distributable. The new standard, known as PREMO, goes beyond previous work within SC24 (such as GKS and PHIGS) to encompass multimedia applications, and with them, a host of technical issues such as maintaining synchronisation within a distributed context.

In this paper we explain how PREMO utilises various programming paradigms, including object-oriented concepts, to provide a distributed and extensible environment for multimedia applications, balancing the needs for efficiency and easy implementability against the demands for a high level, portable application programming interface. In the process, we identify some important trade-offs that have to be made when using various programming paradigms, and explain the rationale for the approach adopted by PREMO.

1.1. A Short Overview of PREMO

This section gives a very short overview of PREMO; for a more detailed presentation the interested reader should consult, for example, Herman et al.[5]†.

Today's application developers needing to realize high-level multimedia applications which go beyond the level of multimedia authoring do not have an easy task. There are only a few programming tools that allow an application developer the freedom to create multimedia effects based on a more general model than multimedia document paradigms, and these tools are usually platform specific. In any case, there is currently no available ISO/IEC standard encompassing these requirements. A standard in this area should focus primarily on the *presentation* aspects of multimedia, and much less on the coding, transfer, or hypermedia document aspects, which are covered by a number of other ISO/IEC or de-facto standards (for example, MPEG). It should also concentrate on the *programming* tool side, and less on, e.g., the (multimedia) document format side. These are exactly the main concerns of PREMO.

It is quite natural that the initiative for a standardization activity aiming at such a specification came from the group which has traditionally concentrated on presentation aspects over the past 15 years, namely ISO/IEC JTC1/SC24 (Computer Graphics). Indeed, this is the ISO subcommittee whose charter has been the development of computer graphics and image processing standards in the past. The Graphical Kernel System was the first standard for computer graphics published in this area; it was followed by a series of complementary standards, addressing different areas of computer graphics and image processing. Perhaps the best known of these are PHIGS, PHIGS PLUS, and IPS (see, e.g., Arnold and Duce[6] for an overview of all these standards). The subcommittee has later turned its attention to presentation media in general as a way of augmenting traditional graphics applications with continuous media such as audio, video, or still image facilities, in an integrated manner. The need for a new generation of standards for computer graphics emerged in the past 4–5 years to answer the challenges raised by new graphics techniques and programming environments and it is extremely fortunate that the review process to develop this new generation of presentation environments coincided with the emergence of multimedia. In consequence, a synergistic effect can be capitalized on.

The JTC1 SC24 subcommittee recognised the need to develop such a new line of standards. It also recognised that any new presentation environment should include more general multimedia effects to encompass the needs of various application areas. To this end, a project was started in SC24 for a new standard called PREMO (Presentation Environment for Multimedia Objects) and is now an ongoing activity in ISO/IEC JTC1 SC24 WG6. The subcommittee's goal is to reach the stage of a Draft International Standard in 1997.

The major features of PREMO can be briefly summarised as follows.

- PREMO is a *Presentation Environment*. PREMO, as well as the SC24 standards cited above, aims at providing a standard "programming" environment in a very general sense. The aim is to offer a standardised, hence conceptually portable, development environment that helps to promote portable multimedia applications. PREMO concentrates on the application program interface to "presentation techniques"; this is what primarily differentiates it from other multimedia standardization projects.
- PREMO *is aimed at a Multimedia presentation*, whereas earlier SC24 standards concentrated either on synthetic graphics or image processing systems. Multimedia is considered here in a very general sense; high-level virtual reality environments, which mix real-time 3D rendering techniques with sound, video, or even tactile feedback, and their effects, are, for example, within the scope of PREMO.
- PREMO *is a framework*. This means that the PREMO specification does not provide all the possible object types for making graphics or multimedia. Instead, *PREMO* provides a general programming framework, a sort of middleware, where various organisations or applications may plug in their own specialised objects with specific behaviour. The goal is to define those object types which are at the basis of any multimedia development environment, thereby ensuring interoperability.

Issues related to the various programming paradigms in use in PREMO to achieve these goals are the main topic of this paper.

† The reader may also refer to the current draft of the PREMO document itself, which is publicly available. The World Wide Web site `http://www.cwi.nl/Premo/` gives a good starting point to navigate through and access all available documents.

2. Object-Orientation

2.1. The Object Model

From the outset it was decided that PREMO would be defined in an object-oriented framework, spread across a number of components which provide a hierarchy of definitions and services.

To take such an approach seems to be an obvious choice, and is highly motivated both by the demands of industry and the advantages of object-oriented design. However, the term 'object-oriented' is extremely loose, various systems, languages, etc., adopt their own view of what objects really are, what their capabilities are, how they are created and destroyed, etc. When defining an international standard, which must be ruthlessly precise, and independent of the specifics of any operating system or programming language, a well defined object model should be adopted. It was a somewhat unpleasant surprise for the team developing PREMO that such object-model did not exist; a lot of energy was spent to define a proper model which encompasses the need of a multimedia standard. To ensure a proper specification, a separate activity on the formal specification of this object model was also pursued; the results are already published elsewhere[7]. Of course, the PREMO object model does rely on existing systems. The model is largely based on the OMG proposal though with significant modifications (see below) and with a more precise specification.

A PREMO system consists of a collection of objects, each with a local (internal) state, and an interface consisting of a set of operations. Each object is an instance of an object type, which defines the structure of its instances. An object type can be defined as an extension to one or more other object types through inheritance. An important property of the model is that objects are never accessed directly. Instead, a PREMO client requests a facility called an "object factory" to generate an object satisfying specific criteria, and if it is able to comply, the factory will return a handle to the new object called an object reference. All subsequent activities involving the object is then done via the reference, for example invoking an operation on the object, or passing the object as a parameter to another operation. This separation of objects (i.e. physical storage) from their references is vital in supporting the aim of distribution, as an object reference can be used to encode both local address information and the location of a particular object across a network. A consequence of this is that the PREMO environment must provide support for activities, such as invoking an operation on an object, which are often taken for granted as part of an object model. In the case of a remote object for example, an operation invocation must be translated into an appropriate remote invocation mechanism. Such assumptions have a significant impact on the binding of the *PREMO* specification to a specific implementation model.

This object model is fairly traditional. It is also very pragmatic in the sense that it includes, for efficiency reasons, the notion of non-object (data) types, as is the case with a number of object-oriented languages, such as C++ or Java, and in contrast to "pure" object-oriented models, such as Smalltalk. This pragmatism was driven by the fact that a *PREMO* systems should be implementable on various industry-wide environments, for example in C++ or Ada95. This required some kind of restrain in adopting various features, a restrain which undeniably contrasted with the various research results available in academic environments.

2.2. Activity, Distribution

In PREMO, a strong emphasis is placed in the model on the ability of objects to be active. This means that PREMO objects have, conceptually, their own thread of control; objects can communicate with one another through messages, i.e., through the operations defined on the object types. Objects can become suspended either by waiting for an operation invocation to return, or by waiting on the arrival of an operation request. Consequently, operations on objects serve as a vehicle to synchronize various activities. Furthermore, operations may be defined as synchronous, asynchronous, or sampled (this latter is, essentially, an asynchronous operation whose waiting queue is of length 1 only). Whether the concurrent activity of active objects is realized through separate hardware processors, through distribution over a network, or through some multithreaded operating system service, is oblivious to PREMO and is considered to be an implementation dependency. The emphasis on the activity of objects stems primarily from the need for synchronization in multimedia environments and forms the basis of the synchronization model in PREMO. Using concurrency to achieve synchronization in multimedia systems is not specific to PREMO. Other models and systems have taken a similar approach (see, for example,[3] or[8]) and PREMO, whose task is to provide a synthesis for standardization, has obviously been influenced by these models.

The initial vision for PREMO was that all objects would potentially be distributable. However, this then requires that all operations — creation, operation request, etc. — involving an object are handled by a mechanism in the PREMO environment. Such a mechanism imposes a heavy run-time overhead on the use of objects, one which is untenable for applications like solid geometry renderers or ray-tracers that may have to create hundreds of thousands of objects within tight real-time constraints. It thus appears that a system like PREMO requires at least two kinds of objects, 'heavy-weight' ob-

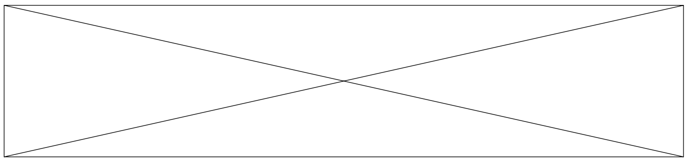

Figure 1: *Top level of PREMO Object Type Hierarchy*

jects that can be distributed, and 'light–weight' objects that can be created and operated on cheaply, but for which there is limited scope for distribution. PREMO reflects these two requirements in the top of the subtype hierarchy, shown in Figure 1. All PREMO objects are subtypes of *PREMOObject*, but only those objects defined as subtypes of *EnhancedPREMOOBject* are distributable (and therefore heavyweight). And it is only objects of this type or its subtypes that can provide services to PREMO clients. Note that this distinction is very often not done in various (distributed) object environments (like CORBA[9]) which usually tend to concentrate only on those objects which are 'visible' over a network.

2.3. Interfaces and Types

PREMO emphasizes the difference between *interface* and *type*. Although, with the widespread usage of Java, this distinction becomes more accepted these days, it was a somewhat unlucky effect of C++ to blur the differences for a long time, except for the very well informed users.

A practical consequence of this differentiation within PREMO is the level of detail in the specification of various objects. Indeed, the PREMO text itself is, essentially, the specification of a large number of object types; if the difference between interface and type were not enforced, the PREMO standard should include *all* possible operations for all types. In other words, if an implementation aims at being compliant with PREMO, but would see the necessity to add new operations to a specific object, this could be done only through subtyping, hence leading to a possible explosion of types. Instead, PREMO defines the behaviour of object types, and defines those and only those operations which are relevant for the behaviour of an object in term of PREMO. In other words, the set of operations described in PREMO may very well form only a subset of all the operations available for an object in a real-life implementation.

2.4. Processor versus Data Types

A fundamental question that must be addressed within any object oriented graphics or multimedia system concerns the allocation of fundamental behaviour, such as transformations and rendering, to object types within an API. Two quite distinct approaches emerge. The first is to attach behaviour to the object types that are affected by that behaviour. For example, geometric objects and other kinds of presentable media data can be defined with a 'render' method, with the interpretation that such an object can be requested to produce a rendering of itself. Such an approach can be extended to collections of presentable objects, and fits well with the concept of an object as a container for data along with the operations that manipulate that data. The second approach is to define objects whose principle purpose is to act as information processors, and which receive the data that they operate on as parameters to operation requests or through some other communication mechanism. In this case, a 'renderer' object would receive presentable objects as input through some interface, and produce a rendering of those objects via some output mechanism. Separating operations from the data that they manipulate may appear to violate a central tenant of object-oriented design. However, it has three important benefits for PREMO.

First, a direct and desired consequence of a distributed model is that one model or data set may be rendered by several processes working in parallel at various locations. It is difficult to see how this can be realised efficiently in an architecture in which each model object renders itself. Either such objects must be able to support multiple concurrent threads internally, or any object that is to be rendered must first be copied. In contrast, treating renderers as a type of object means that multiple renderers can be created (relatively) easily to operate on a given database of model objects. This database can either be shared by several renderers, or there may be several copies of the data. Strategies for managing the distribution, update, and access control of data within such a system are well known, and thus this system is rather more practical and flexible than the alternative.

Second, there is a strong requirement that PREMO be extensible, and this property is actually enhanced by separating renderer functionality from the object types that are rendered. To see why this should be so, suppose that we have 2 type of renderer, R and S, and three kinds of renderable object, say A, B and C. There are two distinct ways of organising the design of this system,

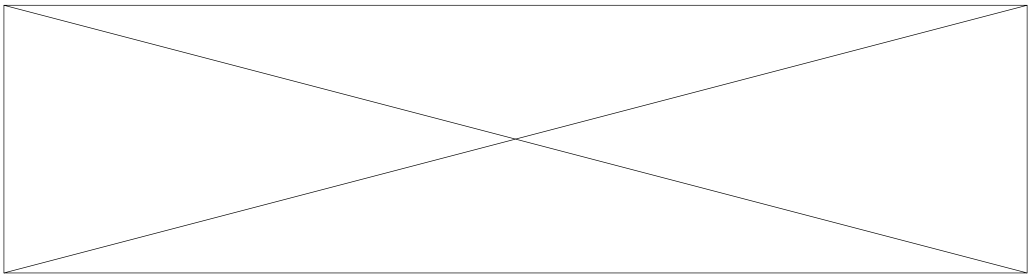

Figure 2: *Organising Data Types and Processes*

as illustrated in Figure 2. The first (a) is to associate each operation with the data object that it acts on, while the second is to structure the design around the processes that act on the data. Cook[10] has used such a framework to contrast object-oriented structures with abstract data types by relating data constructors to data observers. Here, the constructors are represented by object types, while the observers are the processes that act on the data.

Consider now the effect of extending the system shown in Figure 2. Two kinds of extension can be identified: (i) adding a new kind of renderer, and (ii) adding a new kind of modelling object. Adding a new renderer is relatively straightforward in case (b) as it involves no modification of existing code, but more difficult in case (a) as the code for the renderer must be distributed across each of the existing modelling object types. The existence of an inheritance hierarchy above the modelling object types would not ameliorate this effort, as the behaviour of the renderer may depend specifically on the details local to each of the modelling object types. For the same reason, adding a new modelling object type is simpler in case (a) when compared to (b). Critically however we would argue that it is more likely that a PREMO system will be extended by new renderers, or by new modelling object types and renderers specific to that object type, than it would by just adding a new modelling object type. For example, a PREMO system might be extended with a component for constructive solid geometry. This would introduce new modelling primitives, and a renderer specifically designed for dealing with those primitives.

The third benefit of adopting a 'renderer as object' architecture for PREMO is that it supports an approach to application development based on interconnecting a number of processing devices. Once such a network has been defined, it can be used for a variety of data sets or models, and can be readily modified. This approach is already well established in the multimedia community, see for example Gibbs and Tsichritzis[8]. In contrast, in an architecture where modelling objects render themselves, the control of processing and flow of data is encoded within specific operations, making it difficult to develop an application that can be modified or extended without wholesale reprogramming of those operations.

The object model of PREMO has been developed to support an architecture in which renderers and other devices for processing media data are viewed as 'resources' that can be connected and combined to form a network capable of meeting specific presentation requirements. There are two fundamentally different approaches by which such inter-connectivity can be realised. The first of these is to provide a set of homogeneous building blocks that are designed from the outset to interoperate. A systematic way of achieving this goal is to design all of the processing and data representations from the ground up, within a *common infrastructure*. In the case of PREMO, we would in effect need to provide a 'renderer construction toolkit' so that for example a video renderer and an audio renderer would have compatible interfaces for specifying temporal properties of their behaviour. In practice however it would be unreasonable to expect wholesale redevelopment of existing media technologies within the framework of PREMO. Therefore a second approach has been adopted, in which PREMO provides a superstructure within which suitably defined media devices can be embedded and interconnected. These devices may utilise their own interfaces and implementation, provided that they conform to PREMO's basic requirements to enable interconnection and use.

2.5. Properties

Properties are used to store values with an object that may be dynamically defined and are outside of the type system. Properties are pairs of keys (i.e., strings) and a sequence of values which are conceptually stored within a PREMO object (to use another terminology, each PREMO object has an associated dictionary). Operations are introduced to define, delete, and inquire values from a sequence associated with a key. Properties can be used to implement various naming mechanisms, store information on the location of the object in a network, create annotations on object instances, and play an essential role in negotiation mechanism in PREMO (see Section 3. below). The existence of some properties (i.e., the keys) may be stipulated by the standard, but clients can attach new properties to objects at any time. Properties may also be declared as 'retrieve only'.

Why using properties? The fundamental reason lies, in fact, in the conservative nature of the PREMO object model. Indeed, in PREMO, operations on a type are defined statically, when defining ("declaring") the object. Once the object type has been defined, and an object instance of that type is created, no new operation can be added to that object instance dynamically.

On the other hand, it has been advocated elsewhere that more dynamic object models should be used for graphics or multimedia (see, e.g.,[3] or [11]). Indeed, the use of delegation or, on a more "modest" level, a more dynamic view of ob- jects like, for example, the approach adopted in Python[12] (which allows the addition of operations dynamically), would be more appropriate for graphics and multimedia systems. These features would play an important role, for example, in constraint management, in the adaptability of objects, etc. While we agree with this view, the experiences in the MADE project[3] have also shown that implementing such features on the top of languages or environments which are not prepared for such features represents a significant burden and leads to a loss of efficiency. And, unfortunately, none of the widespread object-oriented systems or languages (C++, OMG specifications, Java, etc.) implement delegation or anything similar. As a consequence, and after some discussions, the adoption of such features was rejected for the development of PREMO.

Properties aim at offering a replacement for such advanced features on a lower level. Although properties do not allow adding new operations to an object instance, the mechanism can at least be used to simulate adding and manipulating new attributes (essentially, data) to object instances. Obviously, implementation of properties do not represent a significant problem. The experience with the specification has also shown that the dynamicity offered by properties seem to be quite appropriate for PREMO, some examples will be shown later in the paper. Consequently, properties play a somewhat less elegant, but very useful role in PREMO in increasing the dynamic nature of object instances.

2.6. Language and Environment Binding

Although the PREMO standard makes extensive use of object-oriented concepts, it does not mandate that an implementation of the standard also uses this technology. In principle, it should be possible to implement PREMO within FORTRAN. However, for practical purposes, it is likely that implementations will utilize object-oriented programming technologies, and this raises some interesting issues. There is a wide variation in the concepts and facilities provided by languages that claim to be object-oriented, and many of these contain at least some non-trivial differences from the object model of PREMO. For example, the description of PREMO uses multiple inheritance in a number of areas. For a binding to C++, this presents no difficulties, as this language provides this feature. A Java implementation is somewhat more difficult in this respect, as the language does not support multiple inheritance of classes (object types); on the other hand it does allow a class to implement more than one interface. In this context, it would be necessary to determine an implementation strategy in which particular functionalities of PREMO were defined as interfaces rather than Java classes, at the possible cost of replicating code. Implementation in Smalltalk, which does not support multiple inheritance of classes and which has no concept of interface would be rather more difficult.

In fact, binding PREMO to a specific object-oriented language is one half of the problem. The PREMO object model requires that the environment of a PREMO system provides certain services, for example facilities to create objects and to invoke operations on remote objects. Such services are not necessarily available in a programming environment (Java and its core packages seem to be more complete in this respect), so in addition to a language binding a PREMO implementation may need to provide an *environment binding* to a broader framework. Thus a C++ binding would need to be augmented, for example, by a binding to CORBA[9].

It must be emphasized that these problems are *not* inherent to the PREMO specification, but to the fact that facilities provided by the so-called object-oriented languages and programming environments are extremely diverse, and this makes them very often conceptually incompatible with one another, too. In other words, any object-oriented system specification, which tries to be language and environment independent (which is the case for PREMO) would face similar problems.

3. Negotiations, Adaptability

PREMO does not include explicit management for general constraints. This decision was not taken easily, and was the result of long and sometimes passionate discussions within the PREMO team. There is indeed a classic tension between the general requirements of constraint management and the essence of object-orientedness: whereas the latter advocates information hiding, the former requires a complete knowledge of all the attributes related to an object. It was recognised that there is no widely accepted object model which would solve this

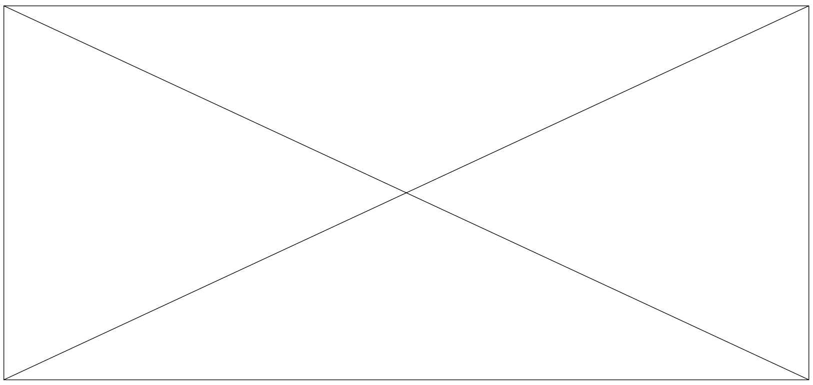

Figure 3: *Type properties, capabilities, constraining properties*

problem in a satisfactory manner an in general terms. PREMO being an international standard, i.e., a platform for general consensus, the development team has finally decided not to include a fully general mechanism for constraint management.

There are, however, some related areas which PREMO does address, and which are absolutely necessary in a multimedia system; these are described in the present section. There is a general, underlying philosophy adopted by PREMO: because of the diversity of the field, PREMO does not include any specific mechanism, algorithm, etc., for, e.g., quality of service issues; instead, PREMO offers the necessary "hooks" to implement various policies, which are often related to a specific application area.

The fundamental mechanism in PREMO is built upon the general notion of properties, briefly described in Section 2.5 already. Properties are the basic building blocks for various configuration and negotiation mechanisms. Such negotiations may be necessary to have, e.g., an optimal control over media flow, to control the quality of service of various multimedia devices, to ensure proper coding and decoding of media data when necessary, etc. As a general principle, the parameters governing the behaviour of objects are described in terms of properties, rather than attributes, if they may be subject to further, dynamic negotiations.

PREMO defines a special subtype, called *Property-Constraint* objects, which offers a set of additional features centred around properties. Figure 3 gives a very schematic view of the notions involved. The figure represents the range of values belonging to *one* property key, whose existence is part of the object type specification. The *capability* associated with this key describes the possible range of values which may belong to this key. This is a read-only information which belongs to a specific *type*. An instance of this type may have a *native property* value for this key, which describes the possible range of values this *instance* can associate to this key. Obviously, the native property value represents a subset of the capability. Capabilities and native property values give a dynamically accessible information on the possible behaviour of an object instance, which can be used in negotiations procedures. PREMO stipulates that, although the actual values of the property may be changed through the invocation of the various property management operations, it is always possible to access the native property values, for all properties whose existence is defined as part of the object type definition. Capabilities and native property values are (retrieve only) properties themselves, i.e., they do not introduce any new notion on the object model level. (Note that capabilities could also be defined as traditional object attributes; the choice is, in this sense, arbitrary.)

PropertyConstraint objects also offers additional facilities to constraint the actual values associated to a key within the range of the native property values of the object. The *constraintProperty* operation, defined for this type, allows a client to set the values associated to a key, automatically checking whether the values represent a subset of the native property values of the object. Finally, these objects have a *select* operation, which determines an optimal range of values for a given key within the range of the (possibly constrained) current values. Note that the *select* operation involves an internal, semantic knowledge of the object, and specific subtypes are supposed to provide an implementation for this operation which reflect the specific features of the object type.

A simple example will show how these notions operate in practice. An audio object type may be defined in the PREMO framework; this type may operate on ulaw, alaw, and Macintosh sound formats. A property is defined for the object, denoting the audio format to be used; eventually, this property has to be set by the user for a specific value.

A *capability* is assigned to this property, which lists ulaw, alaw, and Macintosh. However, when an object is instantiated, it may not operate on, say, a Macintosh sound file, because the necessary hardware is not available. Consequently, the *native property value* for this key will list ulaw and alaw only. A client may inquire this and adapt its own behaviour to this possible choice. Finally, by calling the *select* operation, the client instructs the audio object to set the audio format(s) which is the best suited on a specific environment for this specific instance (e.g., it may restrict the audio format to ulaw).

This property selection and negotiation mechanism appears at various places in the PREMO specification (actually, there are object types, for example the so-called *Format* objects, whose sole purpose is to serve as an interface for such mechanism!). Here are some examples, defined in PREMO, which can be subjected to a negotiation mechanism:

- object types a specific object factory may be able to create;
- various video and audio encoding parameters and formats;
- internet address ranges for distributed access;
- list of input and output primitives a graphical renderer may accept and/or produce;
- quality of service requirements;

One object may have several properties, each of them being subject of the negotiation procedure described above. However, certain combinations of property values may not be acceptable. As an example, audio sample size and sample rates cannot be set independently from one another. To make therefore the negotiation procedure feasible, *PropertyConstraint* objects include yet another property, called *ValueSpaceNameK*, which describe the allowable combinations of other properties. To refer to the same example, this property might include a sequence like:

$$<< \text{“}\mathit{SampleSize}\text{”}, 8 >, < \text{“}\mathit{SampleRate}\text{”}, 8 >>,$$
$$<< \text{“}\mathit{SampleSize}\text{”}, 16 >< \text{“}\mathit{SampleRate}\text{”}, 40 >>$$

which would indicate the fact that a sample size and rate pair of $< 8, 8 >$ or $< 16, 40 >$ are permissible but, for example, a $< 8, 40 >$ is not.

4. Processing Networks

PREMO is concerned with a range of media types, and therefore abstracts away from the details of media processing found, for example, in standards such as GKS and PHIGS, and from the details of media data representation defined for example by MPEG or MIDI. Instead, media processing elements are viewed as “black boxes” that can be interconnected through a high-level interface to construct a network of such elements appropriate for a given application. This “dataflow” approach is not new to PREMO, it appears in published approaches to multimedia systems (for example,[8]), and has also been used in visualisation systems such as AVS and IRIS Explorer to allow interactive construction of applications from a component or module toolkit.

4.1. Multimedia System Services

One of the parts of PREMO, called the Multimedia System Services, defines the building blocks to build up processing networks. Figure 4 gives a very rough overview of some of the notions defined in this part of PREMO; it would of course go far beyond the scope of this paper to give a detailed description of all the objects involved.

The “nodes” in the dataflow network are defined to be so called *VirtualDevice objects*. These objects have “openings”, called ports, which act as input and output for the virtual device. Each virtual device, though being an object itself, is also an aggregate of several specialized objects, all defined by PREMO. These objects allow the client to set up and control the way these devices operate. More specifically, each port has an attached *quality of service descriptor* object and a *format* object; these objects act as a depository of specialised property values (e.g., to define the video or audio format which is produced and/or accepted by a port). The client can set these properties, and hence the properties of the virtual device a whole, using the mechanism described in Section 3. above. Using this mechanism, the client has the possibility to set up specialised processing networks, adapted to the task at hand.

Media stream flows among virtual devices; this flow is controlled by separate constituent objects, called *Stream-Control*. These objects act as a controlling point for a very sophisticated, event-based synchronization mechanism. This synchronisation mechanism is described elsewhere, and the reader is invited to consult either[13], or the PREMO document itself for the details of the multimedia synchronisation. For the purpose of this paper, suffices it to say that the activity of objects, as referred to in Section 2.2, plays a fundamental role in this mechanism.

There are other objects defined in the Multimedia System Services, which aim at a better control of the full processing network. For example, *Virtual connections* act as and abstraction to set up specific networks; *Groups* (see below) provide a single entry point for a group of virtual devices. Note that all the objects can be spread over a real network, i.e., they can form the basis for a really distributed Multimedia environment.

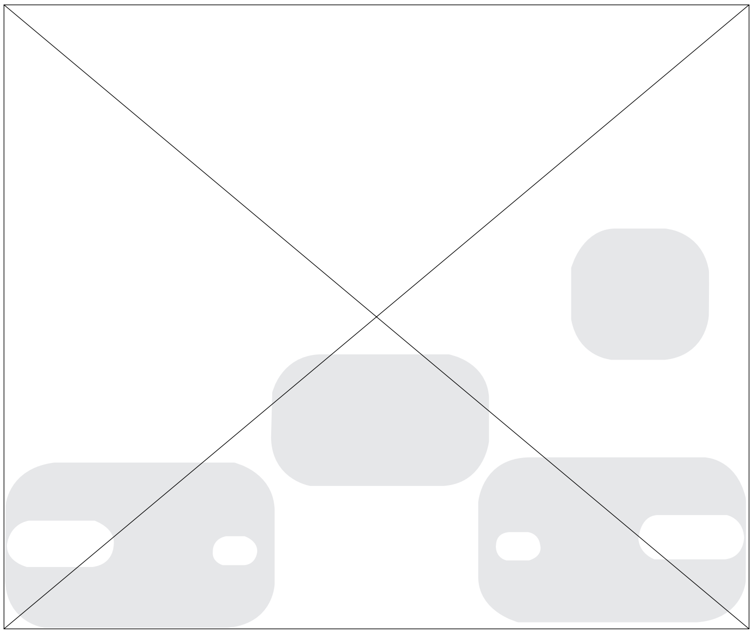

Figure 4: *Multimedia system services client interaction*

Figure 5 contains an example of a small network. It represents a video mixer combining input from a local MPEG file and a remote camera, and displaying the results on a local monitor. Processing is carried out by objects whose type is derived from *VirtualDevice*. Media data is communicated from one device to another via streams, shown as thick lines in the diagram. Streams are established and maintained by objects derived from the *VirtualConnection* type; where a connection involves processes running at different locations, a connection adaptor may be required to mediate communication.

It is often convenient for clients to interact with a single object, and PREMO provides a *Group* object type to support management of a collection of devices and connections. *Groups* are PREMO objects which control a number of other virtual devices, and their respective network. By default, the constituent devices remain hidden to the external client; instead, groups provide a single entry point to stream control, as well as other services. If using the basic group interface only, the client does not have to know about the details, or indeed the interfaces, of these constituent devices. Of course, this restrictive approach is not always desirable; subtypes of *Groups* may add additional operations which essentially expose the object references of the constituent devices. PREMO provides a number of object types that specialize *Group* (e.g., *LogicalDevice*), and, as the *Group* object type is itself a subtype of *VirtualDevice*, objects involved in processing multimedia streams can conveniently be organized into hierarchies.

4.2. Rendering Networks

The Multimedia Systems Services described above provide the foundation for rendering within PREMO, which is significantly more abstract than that found in earlier ISO standards such as GKS and PHIGS. The wide range of software available to today's developer — including implementations of the above Standards — means that it is not sensible to attempt to provide a common interface for graphics renderers, let alone for processing components that involve other combinations of media with their own concerns. Instead, the Modelling, Rendering and Interaction (MRI) Component of PREMO defines a collection of object types that are intended to allow developers to interface modelling and rendering and software to other devices via the MSS framework. In a sense, the purpose of the MRI component is to provide middleware "connectors" or "hooks" to link application or domain-specific components to the system facilities. It achieves this in two main ways:

- it provides a number of object types derived from the VirtualDevice type of MSS that provide generic functionality, and defines a minimum number of constraints and properties that a client may rely on when negotiating the construction of a rendering network;
- it defines a hierarchy of object types for representing the data (primitives) processed by MRI devices.

These points are expanded in the remainder of this section.

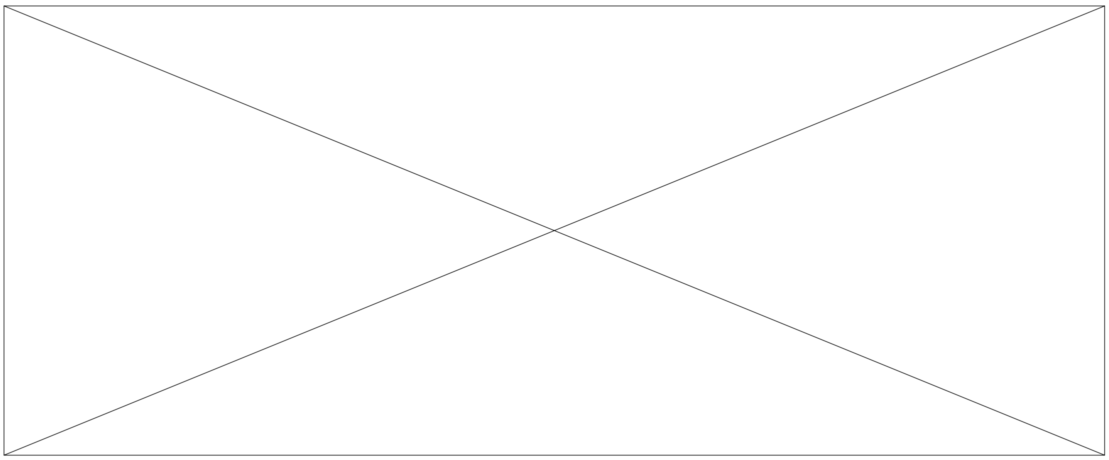

Figure 5: *A Small PREMO Application*

Devices for a Modelling and Rendering Network

Modellers and renderers are defined as subtypes of the *VirtualDevice* object type. This allows them to be integrated directly into a network of devices that may include media–specific input and output devices as well as more abstract processing nodes. An example of such a network is shown in Figure 5. As virtual devices, modellers and renderers contain a number of ports that allow either input or output of data in a particular format. A subtype of the *format* object type, called *MRI_Format*, is defined for data streams that carry modelling and rendering primitives. PREMO applications may specialize this format object type to define the input and output format of a renderer or modeller that can utilise a richer collection of primitives.

Support for modeller–renderer networks is provided through a number of specialisations of VirtualDevice. These include the *Scene* and *Synchronizer* object types. As PREMO supports distributed applications, there are situations where multiple modellers and renderers may be utilising a common set of primitives that defines some presentation, either creating or modifying it, or rendering the set for presentation. To mediate the concurrent activity of multiple readers and writers, PREMO provides a *Scene* object type as a form of virtual device that can be located within a processing network. One responsibility of a scene object is to provide concurrency control to prevent interference. In this respect a scene object is similar to a conventional database server, and in keeping with the overall design philosophy of PREMO, it is assumed that the environment of a PREMO system will supply a suitable mechanism for controlling concurrent access, for example in the form of multi-granularity locking.

In order to render a multimedia presentation it will at some point be necessary to use media-specific devices. Some object in a network has to be responsible for rendering the primitives that will be used by such devices, and where more than one media is in use, this involves the renderer generating multiple output streams. The data carried on these streams needs to be synchronised to reflect any pattern of coordination required in the presentation, represented for example by the *TimeComposite* primitive discussed below. This means that some object in a multimedia system has to be aware of the primitives being processed and has to be able to manipulate the streams used by the renderer by placing suitable synchronization elements on the streams. PREMO defines an object type called *Synchronizer* to encapsulate this functionality. Since this object type has to be aware of a group of devices and configurations, it is defined as an (indirect) subtype of the *Group* object type mentioned in Section 4.1. Specifically, it inherits from an object type called *LogicalDevice*, which in turn inherits from *Group* and *VirtualDevice*. This use of multiple inheritance means that a *Synchronizer* can coordinate the behaviour of its sub-components using an interface that allows the device to be integrated with other components of a wider rendering network.

Primitives

PREMO cannot and does not attempt to describe a closed set of primitives for modelling and rendering. Instead, it defines a general, extensible framework that provides a common basis for deriving primitive sets appropriate to specific applications or renderer technologies. Modellers, for example, may use specific representations such as constructive solid geometry or NURBS surfaces. Such techniques may require an enriched set

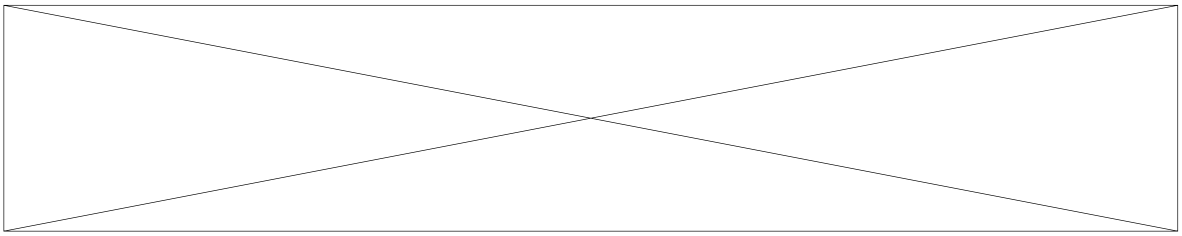

Figure 6: *Three levels of the PREMO Primitive Hierarchy*

of basic primitives. The aim of the primitive hierarchy defined in this part is to provide a minimal common vocabulary of structures that can be extended as needed. Figure 6 provides an overview of this hierarchy; object types written in italic are subtyped one level further in the Standard.

Briefly, form primitives are those where the appearance of the primitive is constructed by the renderer. These include geometric primitives (polylines, curves etc.), and audio primitives for speech and music. Modifier primitives alter the presentation of forms, for example visual primitives encompass shading, colour, texture and material properties that affect (for example) the appearance of geometric primitives. Forms and modifiers are combined within structured primitives. An aggregate is conceptually a set of primitives where some members of the set may be interpreted in application dependent ways; it is thus up to an application subtyping from Aggregate to impose a specific interpretation on such combinations. Of particular importance, given that PREMO is concerned with multimedia presentation, is the TimeComposite primitive and its subtypes which allow a time-based presentation to be defined by composing simpler fragments. Subtypes of *TimeComposite* provide for sequential and parallel composition, as well as choice between alternative presentations as determined by the behaviour of a state machine. Additional control over timing is achieved via temporal modifiers, and subtypes of *TimeComposite* define events that can be used within the PREMO event handling system to monitor the progress of presentation. *Reference* primitives enable the sharing of primitive hierarchies by names that can be defined within structures, while *Captured* primitives allow the import of data encoded in some format defined externally to PREMO.

Although some aspects of the PREMO primitive hierarchy resemble those of object-oriented graphics systems such as OpenInventor[2], it must be remembered that the PREMO hierarchy serves quite a different role from that of models published in the literature or used in implementations. PREMO is not a self-contained specification (let alone implementation) of a multi-media system, but rather a common framework that can be specialised to meet the requirements of a wide variety of applications. In this context there is clearly no "best" primitive hierarchy–different application areas or technologies will have different and sometimes contradictory requirements. The approach of the PREMO Committee has been to synthesise a minimal framework that builds on the facilities provided by PREMO, such as negotiation and event handling, and which can then be extended to suit specific needs.

5. Components

PREMO represents quite a large body of object type specifications. Also, PREMO defines a *framework*, and it is expected that other standard bodies and/or application developers would add their own object types to the ones already defined by PREMO. However, such extensions may not want to make use of all object types PREMO defines. This calls for a proper way of clustering meaningful subsets from the full body of PREMO or related objects. Such a clustering may lead to unresolved type and service dependencies, if not done carefully enough. PREMO includes a set of formalism to make this clustering process easier and trackable.

PREMO defines *components* and *profiles*. A component in PREMO is a set of related object types that comply with the PREMO Object Model. Components organize these object in terms of *profiles*, whereby some set of the types defined in the component are collected together for a particular view of their usage. A profile may be tailored towards a particular constituency or application domain, for example. An example for a component is the Multimedia System Services described above.

A component may contain one or more profiles. The specification of a profile makes explicit the dependencies that the profile has with respect to other profiles within its own component and with profiles defined in other components. These dependencies between profiles is expressed as follows.

- A profile ***P*** belonging to component ***A*** may depend on profile ***Q*** of the same component if there are object types in ***P*** that are either:

a) subtyped from object types defined within ***Q*** (*type dependency*), or
b) whose behaviour depends on operations defined by object types in ***Q*** (*service dependency*).
This form of dependency is referred to as *internal dependency*.

- A profile ***P*** belonging to component ***A*** may depend on profile ***R*** of component ***B*** if there are object types in ***P*** that are either:
  a) derived from other object types defined within ***R*** (*type dependency*), or
  b) whose behaviour depends on services provided by object types defined within ***R*** (*service dependency*).
  This form of dependency is referred to as *external dependency*.

The various possible dependencies are non-exclusive; a component profile may have internal and external dependencies that may be in terms of both type and service dependencies.

The specification of a profile also includes the list of types which can be used to resolve type or service dependencies by other profiles or by applications in general. In other words, a profile specification may include:

- Types which may be subtypes by types in other profiles;
- Types which cannot be subtyped, but only their services can be accessed.

A profile can thereby restrict the usage of a type to, e.g., as a service provider only, i.e., the operations of the type are available for operation requests, but no subtyping of this type is possible.

The separation between service and type dependencies is essential, albeit rarely seen in the literature or in systems. A proper notational conventions is also included in the PREMO specification to describe these dependencies. The profile specification of a PREMO component makes provision for PREMO implementations to offer automatic configuration mechanisms. Such mechanisms may allow for an implementation of a component and/or a profile to interoperate with other component implementations.

6. Assessment and Conclusions

The design of an International Standard is a challenging process in which a synthesis of existing "best practices" must be achieved while bearing in mind the kind of future technical problems that the Standard will have to address. It might seem that development of PREMO would have been assisted by the maturing of object-oriented technologies into mainstream development methods and the opportunities offered by the growth of the Internet and its supporting infrastructure. However, our experience has been that obtaining a suitable foundation in the form of an object model for PREMO was a major problem. Distributed multimedia comes with technical issues such as lightweight versus heavyweight objects and operation invocation that could not be addressed adequately by any one object model being proposed or developed. At another extreme, it was difficult, particularly at early stages in the development process, to identify what specific needs of the Standard were best addressed by object types, or where alternative solutions were feasible and desirable. A good example of this is the extensive facilities for property and constraint management, which provide (we believe) an effective bridge between the object world of local states and internal control, and the world of inter–object constraints and their external management.

Also, the dataflow model of devices and networks provided by the multimedia system services and utilised in the modelling and rendering component provides a highly flexible framework for applications development that abstracts away from the low-level interface issues of the objects needed to support such a network. Our conclusion here are twofold. The first point is probably unsurprising; while object-oriented technologies offer useful and arguably powerful mechanisms for building complex systems (for example inheritance and polymorphism), obtaining a well-defined object model appropriate to the demands of a given application domain is a non-trivial task — object-orientation is no substitute for careful design. The second point is rather more subtle, and may be more significant in the longer term. By adopting the object-oriented paradigm from the outset as the basis for PREMO, we found that we became "locked" into an object-oriented way of thinking about problems. The result was that the PREMO Committee spent a considerable amount of time trying to develop approaches and interfaces for negotiation and rendering *within* an object-oriented framework before we began to realise that solutions to these issues were best found by thinking in quite different terms. Any programming paradigm can become an intellectual straight-jacket that can make it difficult to reach novel but effective alternatives.

Returning to the object-oriented paradigm, another important lesson learnt from the development of PREMO is that it is no longer enough to develop a language binding for an Application Programming Interface. Important features of the PREMO object model rely on facilities that the environment of a PREMO application is expected to provide, for example for the creation and management of objects through an object factory mechanism. Such facilities fall within the scope of object model and object services proposals such as OMG's and JOSS series and CORBA, or the various API specification for Java. Thus, in addition to a language binding linking the types and services of a

Standard to a host language, there is also need for an "Environment Binding" to define how requirements on the environment in which an application will run should be realised through the facilities provided by those various architectures.

Acknowledgment

PREMO is obviously the result of teamwork, and is the outcome of a concentrated effort of the full PREMO Rapporteur Group within ISO/IEC JTC1/SC24. The authors would like to acknowledge the participation of members of this Rapporteur Group.

The original ideas behind the Multimedia System Services have been developed by the Interactive Multimedia Association (IMA) which then donated their specification to ISO. The original IMA document was adapted to PREMO by one of the authors (I.H.) and Jim Van Loo (Sun Micro-systems), and have been revised later by the full Rapporteur Group.

Finally, a large portion of the work presented in this paper has been carried out under the auspices of the ERCIM Computer Graphics Network, funded under the European Communities CEC HCM Programme. The authors would like to acknowledge the participation of members of this programme, and particularly those involved in Task 1.

References

1. M. Kaplan, "The Design of the Dore System", In *Advances in Object-Oriented Graphics I*, E.H. Blake and P. Wißkirchen (Eds), Eurographic Seminar Series, Springer Verlag, 1991.
2. J. Wernecke, *The Inventor Mentor*, Addison Wesley, 1994.
3. I. Herman, G.J. Reynolds, and J. Davy: "MADE: A Multimedia Application development environment". In *Proc. of the IEEE International Conference on Multimedia Computing and Systems, Boston*, L.A. Belady, S.M. Stevens, and R. Steinmetz (Eds.), IEEE CS Press 1994.
4. P. Ackermann, "Direct Manipulation of Temporal Structures in a Multimedia Application Framework", In *Proceedings of the ACM Multimedia'94 Conference*, D. Ferrari (Ed), ACM Press, 1994.
5. I. Herman, G.J. Reynolds, and J. Van Loo: "PREMO: An emerging standard for multimedia. Part I: Overview and Framework", In *IEEE MultiMedia*, **3**, pp. 83–89, 1996.
6. D.B. Arnold and D.A. Duce, *ISO Standards for Computer Graphics: The First Generation*, Butterworths, 1990.
7. D.A. Duce, D.J. Duke, P.J.W. ten Hagen, I. Herman, and G.J. Reynolds: "Formal Methods in the Development of PREMO". In *Computer Standards & Interfaces*, **17**, pp. 491–509, 1995.
8. S.J. Gibbs and D.C. Tsichritzis, *Multimedia Programming*, Addison-Wesley, ACM Press series, 1995.
9. R. Otte, P. Patrick, M. Roy, *Understanding CORBA*, Prentice Hall, 1996.
10. W.R. Cook, "Object-Oriented Programming Versus Abstract Data Types", in: *Foundations of Object-Oriented Languages: Proceedings of REX School/Workshop*, J.W. de Bakker, W.P. de Roever, and G. Rozenberg (Eds), Volume 489 of Lecture Notes in Computer Science, pp. 151–178, Springer Verlag, 1990.
11. D. Brookshire Conner and A. van Dam, "Sharing between graphical objects using delegation", in: *object-oriented Programming for Graphics*, C. Laffra, E.H. Blake, V. de May, X. Pintado (Eds), Focus on Computer Graphics Series, Springer Verlag, 1995.
12. A. Watters, G. van Rossum, and J.C. Ahlstrom, *Internet Programming in Python*, M&T Books, 1996.
13. I. Herman, N. Correia, D.A. Duce, D.J. Duke, G.J. Reynolds, and J. Van Loo, "A Standard Model for Multimedia Synchronization: PREMO Synchronization Objects", In *Multimedia Systems*, to appear in **4**, 1997.