Centrum voor Wiskunde en Informatica

PREMO: A case study in formal methods and multimedia system specification

D.A. Duce, D.J. Duke, G. Faconti, I. Herman and M. Massink

Information Systems (INS)

INS-R9708 November 30, 1997

Report INS-R9708
ISSN 1386-3681

CWI
P.O. Box 94079
1090 GB Amsterdam
The Netherlands

PREMO: A Case Study in Formal Methods and Multimedia System Specification

D.A. Duce
Rutherford Appleton Laboratory
Chilton, Didcot, Oxon OX11 0QX, U.K.
Email: D.A.Duce@rl.ac.uk

D.J. Duke
University of York
Heslington, York, YO1 5DD, U.K.
Email: duke@minster.york.ac.uk

G. Faconti
CNR-CNUCE
via S.Maria 36, I-56126 Pisa, Italy
Email: faconti@cnuce.cnr.it

I. Herman
CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
Email: Ivan.Herman@cwi.nl

M. Massink
CNR-CNUCE
via S.Maria 36, I-56126 Pisa, Italy
Email: M.Massink@guest.cnuce.cnr.it

ABSTRACT

SC24, the sub-committee of the International Organisation for Standardization responsible for the area of computer graphics and image processing, is in the process of completing work on a new standard for multimedia systems, called PREMO. For the first time in SC24, formal methods were employed during the development of the standard. The lessons learned from this exercise are interesting for two reasons. First, PREMO spans concerns ranging from the underlying object model through to issues related to media content. The broad scope of this work has presented challenges to the use of formal methods that have not been reported in other industrial applications. Second, the standards development process places restrictions on how formal methods can be applied. This paper describes the approaches that the PREMO specification group adopted to address the technical demands of the application area and, critically, where those approaches were found wanting. The lessons learned from the case study are discussed in the context of recent debates within the formal methods community on making formal methods relevant to software practitioners.

1991 Computing Reviews Classification System: D.1.5,D.2.1,D.2.2,F.4.3,F.4.m,H.5.1,I.3.0,K.4.m,K.6.3
Keywords and Phrases: Multimedia, formal methods, Object-Z, PREMO, standardization
Note: This paper has been submitted as a journal publication. At CWI, the work was carried out under the project INS3.1: "Information Engineering Framework".

1. INTRODUCTION

1.1 Formal Methods in the Context of the PREMO Standard

"The clear advantages of a more mathematical approach to software design has certainly been well documented; the literature contains many excellent examples of applications of formal methods for large, critical, or even business transaction systems. Despite the evidence, however, a large percentage of practitioners see formal methods as irrelevant to their daily work", words written by Saiedian in his introduction to a series of short articles by leading experts in academia and industry under the heading "An Invitation to Formal Methods" [1]. We set the scene for this paper with the aid of a series of quotations from this collection of short articles.

Jones [2], one of the pioneers of formal methods, characterises his position on how to foster the use of formal methods by the phrase 'formal methods light'. He writes ".. it was important to understand the formal basis but to use - in most cases - a less than completely formal approach; this course was proposed on the assumption that one was capable of filling in the formal details where necessary ... I today teach courses on how to sketch abstract models of systems where a minimum of emphasis is put on the notational details, and the central idea is that of presenting an abstract state for a system. It is amazing how much understanding of an architecture is captured in this abstract state". Jones goes on to write "Today, formal methods are mainly used in the safety critical area where their detailed application can be justified because of the danger of loss of life. The use of formal methods in a lighter way is both a key to using them on larger-scale applications and a way of penetrating fields outside the safety-critical area."

Hall's article [3] contains the remarks "... we are making fairly 'shallow' use of formality - we did not attempt any proofs of consistency or of particular properties. Nevertheless, we found the specification enormously useful in pinning down just what it was that we were going to build". He concludes: "I believe the right question to ask is 'what can formal methods contribute to improve the quality and decrease the cost of our systems?' ".

Zave [4] writes "The conceptual gap between application domains and mathematics must be bridged by building mathematical models of the application domains. Within an appropriate model, formal language is extended to include the vocabulary and relationships of the domain. The lack of appropriate application models, on the other hand, constitutes a large barrier to the use of formal methods in an application domain". She concludes "because of the difficulty of the task ... the conclusion is obvious: Finding the best way to use formal methods in an application domain is research, not development. It is an unusual kind of research, although certainly not unheard-of. It is intellectually challenging and rewarding, at least when the standards for results are set high. And it is probably the most effective thing we can do to bring formal methods into widespread use".

Dill and Rushby [5] conclude their article on the lessons to be learnt from the application of formal methods in hardware design with the words "We attribute the growing acceptance of formal methods in commercial hardware design to the power and effectiveness of the tools that have been developed, to the pragmatic ways in which those tools have been applied, and to the overall cost-effectiveness and utility that has been demonstrated. We believe formal methods can achieve similar success in selected software applications by following the same principles".

These quotations point to four factors in the use of formal methods:

1. the choice of an appropriate level of rigour and formality in the choice and use of methods;
2. the importance of choosing the right questions to ask about the use of formal methods;
3. the importance of mathematical models of the application domain to underpin the use of formal methods;
4. the importance of effective tools to support the use of formal methods.

This paper describes how the authors have applied formal methods during the development of an international standard for the presentation of multimedia objects, called PREMO (PResentation Environments for Multimedia Objects) and assesses the results achieved in the context of the factors set out above. The lessons that we draw from the experience of formal methods in the development of PREMO are significant in the context of standards development.

We first summarise the major features of the PREMO standard, followed by an outline of our approach to the specification and analysis. PREMO consists of four parts, or components: the object model, the foundation component that defines object types fundamental to distributed multimedia, the Multimedia Systems Services (MSS) component which addresses distribution and networking issues, and the Modelling, Rendering and Interaction (MRI) component which is concerned with media content and processing. The major features of PREMO can be summarised as follows.

- *PREMO is a Presentation Environment.* PREMO aims at providing a standard 'programming' environment in a very general sense. The aim is to offer a standardized, 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, such as MHEG [6, 7], and MPEG [8], which primarily address issues of media content encoding.
- *PREMO is aimed at Multimedia presentation*, whereas earlier 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 object types necessary for making a graphics or multimedia application. Instead, PREMO provides a general programming framework, a sort of middleware, where various organizations or applications may plug in their own specialized 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.

Development of the functional provisions of the standard has been supported by the use of formal description and analysis techniques. This work has focused on three main areas:

1. the PREMO object model;
2. the specification and analysis of inter-media synchronization;
3. the outline of an approach to specifying media content (the current scope of the PREMO project does not deal with the standardization of media content, but instead provides a framework within which media content standards could be incorporated).

The four factors in the use of formal methods identified above will be illustrated with extracts from our work on PREMO in the following way.

- *Level of rigour and formality*. This will be illustrated by our approach to specifying the PREMO object model, for which a two-level approach was adopted, the first level describing the key features of the object model itself, and the second level describing the behaviour of individual objects, to be interpreted in the context provided by the first level. The relationship between the two levels raises issues of rigour and formality.
- *The right questions to ask.* There are two dimensions to the PREMO work. The first concerns the level of abstraction and rigour to be used in the specifications. The second is about focusing on the 'right' aspects of the system to model. The PREMO inter-media synchronization functionality will be used to illustrate this point. The formal specification work has tracked and guided the development of inter-media synchronization functionality in PREMO.
- *Mathematical models*. There is as yet, in our view, no complete mathematical model of multimedia systems in general, though components of such a model exist in the literature. In this paper we illustrate some of the features we would expect to find in such a model, for example ways to represent different kinds of media content, representation of the notion of progression and approaches to modelling object model properties such as multiple inheritance and overloading.

- *Tool support*. This is illustrated with some examples from the work of two of the authors (Faconti and Massink) which uses a number of support tools for the LOTOS notation to explore the properties and behaviour of some aspects of the PREMO specification.

1.2 Standards and Standardization

Our use of formal description techniques has been in the 'formal methods light' style proposed by Jones. Specification was applied during the development of PREMO and concentrated on the key areas that were troublesome and lie at the heart of the PREMO system. In order to explain the rationale for our specification approach, and the results that were obtained, it is necessary to give some background into the process by which ISO/IEC standards are developed.

The ISO description of standards, taken from the ISO web site [9] is "Standards are documented agreements containing technical specifications or other precise criteria to be used consistently as rules, guidelines, or definitions of characteristics, to ensure that materials, products, processes and services are fit for their purpose". There is a potential application for formal methods in establishing fitness for purpose, but that is not a direction that our work has taken. The focus has been on recording agreements on technical specifications in a precise way.

At the international level, there are three organizations which operate and coordinate the processes for producing international standards: the International Electrotechnical Commission (IEC) which is responsible for electrical and electronic engineering technologies, the International Telecommunications Union (ITU) which is responsible for telecommunications and radio communications technologies and the International Organization for Standardization (ISO) which is responsible for all other technology sectors. Information technology standards are covered by a joint technical committee (JTC1) of ISO and IEC which has established common and coordinated working procedures with the Telecommunications Standardization Sector of the ITU (ITU-T).

The work of developing an international standard is undertaken on a voluntary basis by industrialists and academics in the relevant technology sector, working through their national standards bodies who constitute the membership of ISO. There are three key principles in the development of ISO standards: consensus, industry-wide and voluntary. These principles impact the development process, and, as we will discuss, also have an effect on how formal methods can be utilized.

ISO is continually monitoring the effectiveness of its procedures and there have been quite radical changes in recent times to the way in which standards are developed. The ultimate aim is to achieve agreement (consensus) between the member bodies of ISO participating in the work. There are currently six stages in the development process:

1. Proposal stage;
2. Preparatory stage;
3. Committee stage;
4. Enquiry stage;
5. Approval stage;
6. Publication stage.

The first stage confirms the need for a new International Standard and is decided by a vote in the appropriate committee. In the next stage a working group of experts prepares a working draft of the standard. This may go through several iterations until a stable baseline is established. At this stage the document is registered as a Committee Draft and is voted upon and refined until consensus is reached on the technical content. The text is then finalized for submission as a Draft International Standard (DIS). The DIS is then circulated to all ISO member bodies for voting and comment. The ballot period is five months. If the approval criteria are met, the Final Draft International Standard (FDIS) is circulated for a final 2 month yes/no ballot. If approved, the final text (incorporating at most minor editorial changes) is sent for publication.

PREMO has just reached the stage where the DIS text has been completed and submitted to National Bodies for a ballot period. The formal description work has taken place during stages 2 and 3.

The need to work by consensus within an ISO/IEC Committee means that there has to be an openness to compromise, especially in cases where more than one technically sound solution has been proposed to a particular problem or requirement. Corporate and national factors may affect the decision to choose one solution rather than another.

Many of the ISO/IEC standards in the IT area define specifications for products. For example in the field of computer graphics, standards such as GKS and PHIGS are essentially product specifications. They define the functionality of a product and the requirements a conforming implementation must satisfy. They define 'what' must be implemented, rather than 'how'. There is thus a degree of abstraction from an implementation. It is in the nature of the compromises that have to be reached, that this level of abstraction is not always uniform. There are times when it is necessary to tie down some area of the standard in considerable detail, whilst another area might accommodate a broader range of acceptable behaviour.

Because standards development is based on volunteer effort, there is no direct control over the composition of the committee. The members typically come from a wide variety of backgrounds and bring a broad range of experiences to the work. This diversity of background can be an impediment to the use of formal techniques, when participants have no prior experience of the use of formal description. There can also be a significant element of chance in the expertise available to write, read and comment upon formal descriptions within the working group or committee. The volunteer nature of the work does lead to a reluctance to learn new techniques, notations or tools, especially if these do not have a direct impact on the members' normal working practices. The lack of direct funding for standardization work can make it difficult to address such issues.

Within the PREMO project it was fortunate that the working group contained a small number of members who had used formal methods in other projects and were able to enlist the active participation of others with appropriate expertise. This happened at a time when ISO/IEC were starting to encourage the use of formal description techniques in the development of standards, and so agreement was reached that formal description techniques would be used in support of the development of PREMO, following the recommendations of a special Rapporteur Group set up in July 1993 to report on the applicability of formal techniques in this area of standardization [10].

The approach taken was to study key areas of the PREMO functionality in order to provide a solid basis for other parts of the system. In the areas studied, the specification work tracked the development of the functionality of the system and was used to identify problems and new solutions which formed a basis for the formal comments submitted with the votes during the two CD ballots which PREMO has undergone.

1.3 Formal Description Techniques and Standardization

The work has used more than one formal description technique (Z, Object-Z, LOTOS and ACTL) at various stages and for various purposes.

Within computer graphics and interactive systems, there is an extensive literature on the use of formal description techniques to describe particular features of systems, for example [11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24]. A book by Kilov and Ross of Bellcore[25] is an indication of the uptake of formal methods (in this case Object-Z) in industry; further examples are contained in the book by Hinchey and Bowen [26].

The initial choice of Z [27] and Object-Z [28] for the work described here was motivated by three considerations.

1. For reasons that are drawn out later in this paper, it was seen as desirable to keep the structure of the specification aligned as closely as feasible with that of the evolving text in the standard document. It has already been mentioned that PREMO is object-oriented, and as a result the informal text describes object types in terms of attributes, operations and state spaces. Consequently, the use of a state-based formal description technique, such as Z, was seen as most appropriate, while the object-oriented nature of the PREMO functionality affords the use of an object-oriented formal description technique.

2. Within ISO/IEC, the only formal description technique which had the status of International Standard when the work reported here began was LOTOS [29]; this was joined in 1996 by VDM [30]. LOTOS

is a language based on process algebra, though it is coupled to an algebraic specification language, ACTONE. The PREMO work was concerned with the description of the PREMO object model, including communication between objects. LOTOS was considered inappropriate for this work initially, because it is not state-based and not object-oriented.

3. There was expertise in the group in both Z and Object-Z, more so than in process based notations or specification logics. The lack of appropriate expertise in formal description techniques has been recognized as a significant hurdle to overcome in gaining acceptance for these techniques in industry. It is important that experts in a standardization committee who do not have expertise in writing formal descriptions should at least have the opportunity to learn to read them at relatively low cost. This implies that there should be good access to training materials such as books, courses and case studies. There is excellent material on Z available under all 3 categories. As we discuss in this paper, the choice of Z and Object-Z was not necessarily optimal – standards development, like industrial practice, is not immune from the need to make trade-offs and engineering compromises.

At a later stage, LOTOS and LOTOS tools were used to check behavioural properties of some of the PREMO objects (see section 3.4).

2. Object Model

2.1 Overview

Early in the lifecycle of the PREMO project it was decided that the new standard would make explicit use of object-oriented techniques. This was seen as a way of supporting distributed applications, and also assisting in the development of any future extensions. Consequently, the normative part of the standard defines an object model in Part 1, covering concepts such as object types, references to objects, subtyping and inheritance and operation dispatching. The functional provisions of subsequent parts of the standard are defined with respect to this underlying model. It was not the aim of the committee defining PREMO to re-invent well established concepts. At the time that the PREMO object model was being discussed, the Object Management Group (OMG) had also begun development of a standard for object models (later to be subject to formal description [31]). Although much of the PREMO object model is consistent with the OMG framework, multimedia applications require several features that were not provided by the OMG proposal, including the need to associate a specific request semantics (synchronous, asynchronous or sampled) with each operation defined in an object type. Section 2.2 examines the meaning of these 'modes'.

In order to describe PREMO in a formal description technique it is important to be able to describe the behaviour of PREMO objects in the context of the PREMO object model. It was clear that the detailed and low-level requirements of the PREMO object model make it incompatible with the semantics underlying object-oriented specification formalisms, which deliberately abstract away from operational details such as operation invocation mechanisms. Thus, the committee was faced with two choices: either to develop a new specification formalism that encompasses the concerns relevant to PREMO, or to utilize existing formalisms at the probable cost of reducing the level of formality. The development of a standard is not, in itself, a research activity, and the timescales involved meant that the second option, the use of existing FDTs, was the only viable route. In the remainder of this section we describe how a workable compromise between formality and level of detail was achieved.

Our initial approach was to have two levels of specification, linked by (informal) notational conventions. The first level was written in Z, and describes the object model itself. The second level describes the behaviour of PREMO objects themselves and for this the Object-Z notation is used. Certain aspects of the Object-Z specification, for example the meaning of operation invocation, are interpreted with respect to the semantics of the Z object model specification. Conventions in the specification text are used to identify points where specific links between the two levels of the specification are assumed. This situation is similar to the difficulty encountered in implementing object-oriented systems, when the object model of the system being constructed does not correspond to the object model of the language in which it is being written. An example of this was encountered in the MADE project [32], where the MADE object model is not the same as the object system of C++ in which MADE is implemented. The result was that it was necessary to code explicitly some aspects of

the MADE object model in C++, rather than rely on mechanisms intrinsic to C++. The two-level approach to the PREMO specification is described in detail in [33]. An outline of the approach is given here. Full details of the PREMO specifications outlined in this paper can be found in [35, 33, 36].

The PREMO object model defines the semantics of object types and object interactions, and makes a fundamental distinction between an object's identity and an object reference. An object reference is a value that reliably denotes a particular object together with information about the type structure of that object. It also includes the following features:

- An object is considered to be an instance of some object type.
- Objects have a basic characteristic that is their distinct immutable identity.
- Object types can be related to one another in supertype/subtype relationships.
- Operations are applied to objects.

The PREMO object model introduces the concept of non-objects. A non-object is considered to be an instance of some non-object type. Non-objects differ from objects in that they do not form part of an object type hierarchy and do not have an object reference. In this respect the approach taken by PREMO is similar to that of Java [34]. PREMO non-object types include integers, real numbers, and strings. Importantly, *references* to objects are themselves represented as non-object values. In the formal specification of PREMO, non-object data types are represented in Z using given types and basic constructors such as sequences. PREMO defines a disjoint union type over its non-object type family, and this can be modelled by a free-type definition in Z, an extract of which follows:

$$
\begin{array}{lcll}
value & ::= & int\_value\langle\langle \mathbb{Z} \rangle\rangle & \text{– Integers} \\
 & | & real\_value\langle\langle \Re \rangle\rangle & \text{– Real numbers} \\
 & | & char\_value\langle\langle Char \rangle\rangle & \text{– Characters} \\
 & | & seq\_value\langle\langle \mathrm{seq}\ non\text{-}obj \rangle\rangle & \text{– Sequence} \\
 & | & obj\_ref\_value\langle\langle objref \rangle\rangle & \text{– Object references} \\
 & | & \ldots &
\end{array}
$$

In the following subsections we focus on two specific aspects of the object model specification which allow us to discuss and illustrate the tension between level of rigour and ease of expression that was identified at the start of the paper as one of the four factors affecting the use of formal methods in this context.

2.2 Operation Dispatching

Operations are actions that can be applied to an object. Each operation has a signature which consists of a name, a list of parameter types and a list of result types. When an operation request is issued, a specific operation implementation is selected for execution. This selection process is termed *operation dispatching*. The process of selecting which operation implementation to invoke (bearing in mind that an object may contain different implementations of an operation of the same name) is based on a controlling parameter of the actual call, which defines the type with which the object is to be viewed for this call.

The PREMO object model's concept of operation dispatching has a strong operational bias, for example, in the different kinds of service request semantics. Objects may define their operations as being *synchronous*, *asynchronous*, or *sampled*. The intuitive meaning of these concepts is:

- *synchronous*: the caller is suspended until the callee has serviced the request;
- *asynchronous*: the caller is not suspended, and the service requests are held on the callee's side, no return values are allowed in this case.

- *sampled*: the caller is not suspended, at most one pending service request is held by the callee. Sampled is thus similar to asynchronous, the key difference being that in asynchronous mode any number of requests may be held by the callee whereas in sampled mode at most one request may be held and any pending request will be overwritten by later requests.

Although the PREMO concept of operation dispatching is quite close to the model of message passing assumed by Object-Z, some aspects of PREMO, such as the different kinds of service request semantics, have a distinctly operational flavour. To ensure a level of interoperability, PREMO must place certain requirements on the basic facilities provided by an implementation platform. When it comes to defining the behaviour of higher-level services, such as synchronization objects, these low-level requirements become assumptions, on which the appropriate behaviour of such services becomes contingent. What follows now is an outline of how operation dispatching in PREMO was given a formal description. While based on the work reported in [33], the specification has been reshaped and simplified for this paper. In order to illustrate just one facet of the object model, i.e. operation dispatching, we ignore issues of inheritance, subtyping, the type system, and some details related to object interfaces.

Parts of the run-time state relevant to operation dispatching are described formally below, beginning with three definitions: *params* is a sequence of actual parameter values; *request* represents the invocation of an operation on a specific object; *opmode* defines the three operation dispatch modes.

$$params == \operatorname{seq} non\text{-}obj$$
$$request == object \times operation$$
$$opmode ::= async \mid sampled \mid sync$$

A distinction is made between the actual operations (denoted by the type *operation*) and the names of operations (denoted by the type *opname*). The run-time state defines the operation invoked by an object on receipt of a specific message, the mode of each operation, the objects that are suspended pending completion of a synchronous request, and, for each operation invocation (request), a bag (multiset) containing calling objects and parameter values that are pending execution. The schema *Runtime* includes a schema called *ObjectSystem*, details of which are not given here; it is included as a placeholder for those issues (inheritance, etc) that are not covered in this paper.

Runtime

$$ObjectSystem$$
$$interface : object \pfun (opname \pfun operation)$$
$$mode : operation \pfun opmode$$
$$suspended : \mathbb{P}\, object$$
$$pending : request \fun \operatorname{bag}(params \times object)$$

$$\forall o : object \bullet \forall p : operation \bullet$$
$$mode(p) = sampled \Rightarrow count(pending(o, p)) \leq 1$$

In fact, the invariant is rather more complex than that given above, since valid states of the run-time system are determined in part by the structure of the interfaces defined by the object types, and the objects that exist within the system at any point in time. Again, full details can be found in [33]. The specification of operation dispatch is spread over three schemas, one each for invocation, evaluation and return.

Operation invocation involves adding an operation request to the bag of pending requests. If the invoked operation is sampled, any existing item in the bag is discarded, otherwise the bag is extended with the given parameters and the identity of the invoking object. The latter is required for synchronous operations, where the caller must be un-suspended once the request has been serviced.

Invoke

$\Delta Runtime$
$\Xi ObjectSystem$
$sender? : object$
$receiver? : object$
$op? : opname$
$args? : params$

$op? \in \operatorname{dom} interface(receiver?)$
$sender? \notin suspended$
let $reply == (args?, sender?)$ •
let $method == interface(receiver?)(op?)$ •
let $call == (receiver?, method)$ •
let $held == pending(call)$ •
$mode(method) \neq sampled \Rightarrow$
$pending' = pending \oplus \{call \mapsto held \uplus [\![reply]\!]\}$
$mode(method) = sampled \Rightarrow$
$pending' = pending \oplus \{call \mapsto [\![reply]\!]\}$
$mode(method) = sync \Rightarrow$
$suspended' = suspended \cup \{sender?\}$
$mode(method) \neq sync \Rightarrow$
$suspended' = suspended$

The second stage is the evaluation of a pending request. Two outputs ($op!$ and $callee!$) are used to denote a request for which there is a non-empty bag of pending calls. A comprehensive object model that encompassed the states of objects and semantics of operations could extend this schema to define the effect of operation evaluation on the global state.

Evaluate

$\Delta Runtime$
$\Xi ObjectSystem$
$op! : operation$
$callee! : object$

---

let $r == (callee!, op!)$ •
$count(pending(r)) > 0$
$\exists\, args : params;\ caller : object$ •
$(args, caller) \in pending(r)$
$callee! \notin suspended$

An operation is completed by removing the serviced request from the pending bag, and, in the case of synchronous operations, removing the caller from the suspended set.

Return

$\Delta Runtime$
$\Xi ObjectSystem$
$op? : operation$
$callee? : object$

$\mathbf{let}\ r == (callee?, op?) \bullet$
$count(pending(r)) > 0$
$\exists\, args : params;\ caller : object \bullet$
$(args, caller) \mathbin{\triangleleft\!\!\triangleright} pending(r)$
$pending' = pending \oplus \left(\begin{array}{l} \{r \mapsto pending(r)\} \\ \uplus \\ [\![(args, caller)]\!]\} \end{array} \right)$
$mode(op) = sync \Rightarrow$
$suspended' = suspended \setminus \{caller\}$
$mode(op) \neq sync \Rightarrow$
$suspended' = suspended$

Informally, the Object-Z expression $obj.opn(args)$ should be understood as the *Invoke* operation, followed immediately by the operation $Evaluate \gg Return$, where *Evaluate* selects the invoked operation for execution, followed by an immediate return. That is, by default, operations in the Object-Z component of the model are assumed to be synchronous.

Although this sequence of operations can be modelled formally using the schema composition operators of Z, the result does not capture the true behaviour of a PREMO system. As PREMO objects are active, operation invocations may take place concurrently, and/or interleaved with evaluation and return phases. This cannot adequately be captured in the Z model. A thorough account of the dynamics of PREMO objects might be given in terms of the π-calculus [37], or in $\pi o \beta \lambda$ [38], which itself is founded on the π-calculus.

2.3 Specifying Foundation Objects

PREMO addresses the synchronization of media streams in a distributed setting through comprehensive event and stream models described in sections 3 and 4. Such facilities are defined in the standard as object types, in the context of the object model, and, as a consequence of their complexity and importance to meeting the objectives of the standard, have been the target of considerable specification effort.

One of the most difficult technical issues facing those involved in the specification of these objects was how to associate their description with the formalized object model. It had already been decided that Object-Z would be used for describing PREMO object types, as the notation encompassed many of the concepts and ideas fundamental to PREMO. However, differences, such as the operation dispatch semantics described above, had to be addressed at some level. Our initial idea for accomplishing this was to 'invent' some notation for actions within the Object-Z universe that should be interpreted in the context of the Z model. We will comment on the propriety of this approach shortly. First however we illustrate its use through the original specification of a PREMO event handler. PREMO events provide a general mechanism for synchronisation between separate object instances in a running PREMO system. Part 2 of the PREMO standard defines an Event Handler object that allows objects to register interest in specific events. When the handler is notified of an event, an operation is invoked on each object that has registered interest. The following specification fragment, again modified from the material in [33], assumes that the set *event-name* has been defined to represent event names.

EventHandler

PREMOObject

$register : event\text{-}name \leftrightarrow (opname \times object)$

INIT

$$register = \varnothing$$

The state of an event handler is a relation between event names, and (operation name, object) pairs. Initially, this relation is empty. Informally, whenever an event named *e* is dispatched to the handler, the operation *op* should be invoked on object *obj* for every pair (op, obj) where $e \mapsto (op, obj)$ is in the register.

register

$$\Delta(register)$$
$$e? : event\text{-}name$$
$$opn? : opname$$
$$obj? : object$$

$$opn? \in \mathrm{dom}\, \Omega interface(obj?)$$
$$\Omega mode(opn?) \neq sync$$
$$register' = register \cup \{e? \mapsto (opn?, obj?)\}$$

The 'register' operation extends the internal callback register with a new operation/object pair for a specific event. However, the registered operation must be defined in the interface of the given object, and must not be synchronous. This precondition involves structures in the object model specification, which are referenced informally, using the prefix 'Ω' to identify informally that these names are defined in the Z part of the specification.

unregister

$$\Delta(register)$$
$$e? : event\text{-}name$$
$$opn? : opname$$
$$obj? : objref$$

$$register' = register \setminus \{e? \mapsto (opn?, obj?)\}$$

send

$$e? : event\text{-}name$$

$$\forall obj : object;\ opn : opname \bullet$$
$$\quad e? \mapsto (opn, obj) \in register$$
$$\quad \Rightarrow$$
$$\quad \exists\, \Omega Invoke \bullet$$
$$\qquad sender? = \mathrm{this} \wedge$$
$$\qquad receiver? = obj \wedge$$
$$\qquad op? = opn \wedge$$
$$\qquad args? = \langle e? \rangle$$

The final operation, *send*, represents a signal to the event manager that the event $e?$ has occurred. In response, the manager must invoke the appropriate operation on each object that has registered an interest in the event. This is modelled in the specification by asserting that, for each interested object, some model of the *Invoke* operation holds with the input parameters of *Invoke* bound to appropriate values. Again, the reference to the object model definition is marked by Ω.

The use of one versus two levels of specification raises interesting issues of consistency between levels,

and the extent to which proof techniques could be developed to discharge properties that are contingent on more than one level. An obvious concern in this approach is the inherent informality, in particular whether it even makes sense to view such a specification as formal. One of us (DJD) developed the first semantics for Object-Z [39, 40] and through this we had some confidence that the modifications to the semantics of Object-Z captured by the Z object model could, if necessary, provide a rigorous semantic basis for the modified notation. In evaluating this approach, it is important to bear in mind that we knew from the outset that we were concerned only with *modelling* PREMO, and had neither the opportunity nor ambition to verify properties of the specification. Had it been desirable to carry out formal (or even rigorous) arguments, the approach to the entire specification activity would have been very different indeed. Some answers to the questions about consistency and proof techniques may come from work seeking to address the construction of models from partial specifications, e.g. [41], an approach which is seen by Clarke and Wing [42] as an important future challenge for formal methods research. However, in the PREMO specification, the two levels do not quite fit into the model of partial specification. Rather than providing complementary views of a single artefact, the object model is providing a framework in which the specification of object types should be interpreted. It is not clear at present whether research on method integration will encompass such an approach.

Issues of logic and semantics aside, our view of this initial approach to the foundation objects was that the notational clutter involved in referencing the object model simply was not justified in terms of insight into the standard. In later work, for example [36], we have chosen to separate the two levels of specification completely. Object types are specified as Object-Z classes, using the concept of operation invocation directly from Object-Z. Our initial experience in formalising aspects of PREMO was that most insight was derived from the internal constraints on an object type, for example invariants and preconditions, a point which reflects the quote by Jones, given in the introduction, that much understanding of an architecture can be derived by examining the abstract state. The corollary of this viewpoint was that the link between the object model and specific object types becomes rather less important, as the object model and Object-Z differ mainly in dynamic aspects of system behaviour such as operation invocation, which we decided could be abstracted over.

The improvements in readability that this approach brings are well illustrated by comparing the operation *send* in the initial specification of the event handler given above to the form that this operation finally took in the draft standard, where it was renamed *dispatchEvent*. Details of the final specification are given in [36]. Dispatching an event to the event handler invokes the callback operation of all objects that have registered interest in the event. The event management facilities were extended during the development of PREMO to allow each callback object to define a constraint on the event, such that callback is only invoked if the constraint is satisfied. This specification of this part of the functionality is enclosed in brackets and should be ignored for the comparison between *send* and *dispatchEvent*.

dispatchEvent

$$
\begin{array}{l}
newEvent? : Event \\
\hline
\forall\, e : registered \bullet \\
\quad eventName(e) = newEvent?.eventName \\
\quad (matchMode(e) = AND \Rightarrow \\
\qquad \forall\, c : \operatorname{ran} constraints(e) \bullet \\
\qquad\quad newEvent?.eventData \ sat \ c \\
\quad matchMode(e) = OR \Rightarrow \\
\qquad \exists\, c : \operatorname{ran} constraints(e) \bullet \\
\qquad\quad newEvent?.eventData \ sat \ c) \\
\quad \Rightarrow \\
\quad notify(e).callback\,[newEvent? / callbackValue?]
\end{array}
$$

3. MEDIA SYNCHRONISATION

3.1 The Synchronization Problem

Many of the tasks involved in multimedia systems involve processing multiple streams of media data under real-time constraints that can change dynamically as the result of user interaction or external factors such as network load. These represent a significant technical challenge in the design of any multimedia system or application, and from the outset, it is useful to divide the problem into two components:

- the problem of coordinating the presentations of multiple media streams, known as *inter-media synchronization*; and
- the task of maintaining the presentation of data at a sufficient rate and quality for human perception, which is usually referred to as *intra-media synchronization*.

Both forms have received significant attention in the multimedia literature, see for example [43] or [44] for further information and references on the topic. Only inter-media synchronization is discussed in this paper, and for brevity the term synchronization is henceforth used to refer to this form only. An efficient implementation of inter-media synchronization represents a major load on a multimedia system, and it is one of the major challenges in the field. What emerges from the experience of recent years is that, as is very often the case, one cannot pin down one specific place among all the computing layers (from hardware to the application) where the synchronization problem should be solved. Instead, the requirements of synchronization should be considered across all the layers, i.e., in networking technology, operating systems, software architectures, programming languages, etc. and user interfaces.

PREMO is a framework for a broad range of multimedia applications, and as such, cannot meet the specialised needs of all domains. Instead the standard provides a number of interwoven mechanisms in the form of object types for event handling, synchronization, stream management, and temporal primitives that can provide a basis for application developers to build on. Specifications of the PREMO inter-media functionality have been developed at a relatively high level of abstraction. An Object-Z specification was developed from an initial proposal for the synchronization facilities, and provided useful insight into the object types and state spaces being considered. However this specification was not amenable to behavioural analysis and so a LOTOS model was developed which, by abstracting further from some of the operational complexity, was tractable to mechanical analysis. By allowing certain properties of operations to be explored, the LOTOS specification revealed some problems in earlier drafts of the Object-Z specification, and in PREMO itself.

An overview of the two models, and the technical challenges encountered, is presented in the remaining parts of this section. Fragments of the Object-Z specification illustrate both the strengths and limitations of the state-oriented approach in this context, and lead to the description of a LOTOS model. The main point of this section is the need to use appropriate models to address specific questions in the development of a system. In the final part of the section, we balance the use of the two techniques and review our experience in the light of the comments by Dill and Rushby quoted in the Introduction.

3.2 Basic Notions in PREMO Synchronization

The PREMO synchronization model is based on the fact that objects in PREMO can be active. Different continuous media (e.g., a video sequence and corresponding sound track) are modelled as concurrent activities that may have to reach specific milestones at distinct and possibly user definable synchronization points, at which events may be dispatched to other objects within the system. Although a large number of synchronization tasks are, in practice, related to synchronization in time, the choice of an essentially "timeless" synchronization scheme as a basis offers greater flexibility. While time-related synchronization schemes can be built on top of an event-based synchronization model, it is sometimes necessary to support purely event-based synchronization to achieve special effects required by some applications. Examples of how the various synchronization objects may be used can be found in [45].

In line with the object-oriented approach of PREMO, the synchronization model is realized through a number of related object types constructed over an event-based 'kernel'. Three object types provide this foundation:

- synchronizable objects, which have an internal progression space and which form the supertype of, e.g., various media object types;
- synchronization elements, which can be placed on a user-definable subset (the span) of the coordination space of a synchronizable object to generate events; and
- event handlers, which may be used to manage complex synchronization patterns among synchronizable objects through the synchronization elements placed on the span of synchronization objects (see Section 3.3).

A synchronizable object is a finite state machine that controls the position and progress through an ordered set of coordinates, some of which may contain synchronization elements that can be used to organize the behaviour of a system constructed using such objects. The intention is that object types representing different kinds of media (video, sound, etc.) will inherit from this object type and specialize the coordinate system and state machine in an appropriate way. This issue is explored in Section 5. In the standard, the internal progression space of a synchronizable object is represented as a generic type, represented in this section by the symbol '*C*', that can be instantiated to one of the following,

- extended real ($\mathbb{R}_\infty$), or
- extended integer ($\mathbb{Z}_\infty$), or
- extended time (Time_∞);

where "extension" means the inclusion of positive and negative "infinity". The use of infinity is useful within the PREMO standard in describing media streams originating from 'live' sources such as microphones, where the temporal extent of the stream is unknown at the time that various objects are created. The obvious extension of the notions "greater than", "smaller than", etc., on these types allows the behaviour of synchronizable objects to be defined more succinctly. *Time* is used here as an abstract type, with no commitment made either to a discrete, dense, continuous or discontinuous foundation. Attributes that define the extent of the progression space can be set through operations defined on these objects. Section 5 describes how objects that assign specific semantics to the progression space, such as time or video frame numbers, may be defined as specializations of synchronizable objects.

We note that the formal representation of these extended types within Object-Z is a non-trivial, though solvable, problem. For example, Z does not define a type for the real numbers, and constructing a model of the reals within the type theory of Z is a complex undertaking. Also, the symbol '∞' is overloaded in the definitions given above. As it happens, we do not need to utilise specific properties of the real numbers, and it could be argued that a specification of PREMO would benefit from a more abstract description than $\mathbb{R}_\infty$ etc, for example by the introduction of a single abstract type to cover the three cases mentioned above. This again highlights the 'shallow' use of formal methods (borrowing Hall's terminology from the Introduction) made within PREMO. The cost of increasing the level of rigour would be to increase the separation between the formal and normative descriptions, a message that we return to in the conclusion of this paper.

The discussion on specification issues that follows requires some knowledge of the operational model of synchronization that the specification is attempting to capture. It is somewhat ironic that the formal descriptions are intended to provide just this kind of knowledge, but as it is the specification rather than the actual behaviour that is the focus of discussion, in the present circumstances an informal explanation will suffice.

Synchronization elements, which can be attached to the coordinate space of a synchronizable object, consist of three components:

- a reference to an event handler object that implements an interface that includes a *callback* operation;
- a reference to an event that should be signalled; and
- a boolean-valued 'wait' flag.

When a synchronization element is attached to the coordinate space of a synchronizable object, it defines a reference point. Once a synchronizable object is placed into 'PLAY' mode, it begins to traverse its coordinate space in a well defined order. Even if the space is discrete, not all coordinates need be visited; playback rates and other constraints may cause the object to process only a subset of any media content attached to the space. However, as the location of the playback position within the span conceptually moves from one presentation position to the next, any reference points between the current and future positions must be visited in the order in which they occur in order to preserve the integrity of the presentation. Upon reaching a reference point, the synchronizable object invokes the callback operation on the event handler referenced in the synchronization element, using the event reference as an argument to the call. Finally, it may suspend itself if the 'wait' flag is set to TRUE. Through this mechanism, the synchronizable object can stop other objects, restart them, suspend them, etc. The interface of the synchronizable object type defines operations to add and remove synchronization elements from the progression space, to alter the parameters that determine traversal of the space, and to switch the state machine between modes such as 'PLAY', 'STOP' and 'PAUSED'.

3.3 Object-Z Specification

Synchronizable objects The specification of the *Synchronizable* object type illustrates the importance of achieving the right level of abstraction in a specification, and the difficulties of achieving this in the context of a standard such as PREMO where a number of important constraints on the detailed operation of an object type are involved. A full specification of the *Synchronizable* object type is a document in itself; the reader is referred to [36] for the details. The vignettes given here are intended to illustrate the difficulties that we encountered and the solutions that were adopted. They also provide a reference point for a second approach to the specification of this object type that is described in the section that follows.

The first fragment of the specification defines *Synchronizable* to be a generic subtype of *EnhancedPREMOObject* and *CallbackByName*; the use of these inherited facilities in the realisation of synchronization patterns is described in [46]. The generic parameter *C* represents the coordinate system of the progression space, restricted to be an extended integer, extended real, or extended time.

Synchronizable $[C :: \mathbb{Z}_\infty \mid \mathbb{R}_\infty \mid TIME_\infty]$

EnhancedPREMOObject
redef (*initialize*, *initializeOnCopy*)
CallbackByName

The attributes that determine how progression is made through the underlying coordinate space are introduced here in groups. The term 'attribute' is used in both PREMO and Object-Z to denote a component of the state of an object that can be both accessed and updated from outside of the object, i.e. without recourse to methods defined explicitly in the interface. The first group consists of attributes that represent the user-definable start and end positions that determine the span over which progression takes place, and requirements on how progression should proceed.

$startPosition : C$
$endPosition : C$ [user-definable boundary]
$repeatFlag : \mathbb{B}$ [should the presentation cycle?]
$nloop : \mathbb{N}$ [total number of loops required]

$startPosition \leq stopPosition$

In PREMO, an attribute can be understood as implicitly defining a pair of operations, an accessor and a modifier; in Object-Z, the concept of attribute is defined in terms of the semantics. Unfortunately, the match between Object-Z access concepts and PREMO access concepts is not complete. For example, PREMO also uses the concept of read-only attribute and internal (protected) operations, derived in part from the access control facilities provided by C++ and Java [47]. There are no formal counterparts to these in Object-Z, although

we do use a common Z convention of prefixing 'Φ' to the names of specification fragments that represent incomplete descriptions of operations. For the work on PREMO, we found it sufficient to link this aspect of the formal and normative specifications via explanatory prose, but were it necessary to construct a formal argument about some property of system behaviour, some of the assumptions about accessibility of object state would needed to have been made explicit.

Attributes in the next cluster are read-only, i.e. their valued can be inspected by clients, but cannot be updated directly. One use for this constraint is where a value is intended to be constant over the life of the object (for example, the maximum and minimum positions within a progression space will be fundamentally linked to the media over which progression is taking place). A second reason is that the encapsulation provided by objects makes it possible to ensure that certain invariants between parts of the state are maintained, which in turn rules out granting clients the ability to make ad-hoc changes, for example information about the current mode and position of a synchronizable object is important to clients in, for example, planning the management of quality of feedback, but no client should be able to change these values other than through well-defined operations such as *play*, *stop*, etc.

$$
\begin{array}{l}
maximumPosition : C \\
minimumPosition : C \qquad \text{[fixed bounds of the span]} \\
currentDirection : Direction \\
loopCounter : \mathbb{N} \qquad \text{[number of loops completed]} \\
currentState : SyncMode \qquad \text{[PLAY PAUSED, WAITING, STOPPED]} \\
currentPosition : C \\
\hline
minimumPosition \leq startPosition \\
stopPosition \leq maximumPosition
\end{array}
$$

The use of attributes creates one more small, but none the less irritating, problem in the application of formal description within PREMO. As mentioned earlier, an attribute is understood in the normative text as a shorthand for pair of operations, one to set the value of a variable, and one to return its value. Such operations can raise exceptions, for example, if the new value of an attribute violates a constraint. This is difficult to document formally in Object-Z, as the set and get methods are not an explicit part of the text, and illustrates the trade-offs involved in specification, in this case between explicitness and readability. One argument is that specification should prompt and assist a designer to be explicit about the behaviour of a system, and this in turn requires being explicit about the components of the system that contribute to its behaviour. The cost is that a model can quickly become cluttered with detail.

The remainder of the state contains *specification variables*. They are not mentioned explicitly in the normative part of the standard, but are introduced in the specification to clarify important concepts and to simplify the description of the behaviour of the object type. The ability of a client to either set the presentation into an infinite cycle, or to specify a number of iterations, means that a given location within the coordinate space may be visited multiple times. It is useful in the specification to distinguish between 'visits' to a given coordinate. To achieve this a type is defined that combines a position in the space with a visit number,

$$Location == C \times \mathbb{N}_\infty$$

and we define a total order over this type.

$$
\begin{array}{l}
\_\, prec \,\_ : Location \leftrightarrow Location \\
\hline
\forall\, c_1, c_2 : C;\ n_1, n_2 : \mathbb{N} \bullet \\
\quad (c_1, n_1)\ prec\ (c_2, n_2) \\
\quad \Leftrightarrow \\
\quad n_1 < n_2 \lor (n_1 = n_2 \land c_1 < c_2)
\end{array}
$$

No invariant is given at this point to link the coordinates visited during traversal with the parameters that determine traversal behaviour. Operations defined later in this object type can update these parameters, and it

simplifies the specification if the relationship between these variables is captured as part of a 'framing' schema that is then used to define the effect of such operations.

Variables introduced in the next cluster define how progress is made during play mode. They include the state component, *stepping*, that indicates when an object is moving from one presentation location to the next, *refpoints*, which defines the synchronization elements that have been associated with specific points on the co-ordination space, and *loopStart*, which is the coordinate that progression will start from initially. The locations that remain to be traversed when the object is in play mode define the *span*, while the relation $\prec$ defines the order in which these locations will be traversed.

$$
\begin{array}{ll}
stepping : \mathbb{B} & \text{[true while moving from current to new]} \\
requiredPosition : Location & \text{[new (target) position]} \\
point : Location & \text{[location during span traversal]} \\
refpoints : C \pfun SyncElement & \text{[the sync. points]} \\
loopStart : C & \text{[the starting coord for loops]} \\
span : \mathbb{P}\, Location & \text{[locations to be traversed]} \\
\_ \prec \_ : Location \rel Location & \text{[order of traversal]} \\
\hline
\dom refpoints \subseteq minimumPosition \upto maximumPosition & \\
currentDirection = forward & \\
\quad \Rightarrow loopStart = startPosition \land (\prec) \subseteq prec & \\
currentDirection = backward & \\
\quad \Rightarrow loopStart = endPosition \land (\prec) \subseteq prec^{-1} &
\end{array}
$$

Two operation descriptions illustrate the economy of definition that is supported by the model given above. First, the *progressPosition* operation calculates the next location to be visited; it is expected that it will be specialised by subclasses to address behaviour specific to various types of media. The point in the coordinate space that will be visited next is returned as an output.

progressPosition

$$
\begin{array}{l}
\Delta(requiredPosition, stepping) \\
newPosition! : C \\
\hline
currentState = PLAY \land \neg\, stepping \\
\exists\, count : \mathbb{N} \mid count < nloop \bullet \\
\quad (newPosition!, count) \in span \\
\quad requiredPosition' = (newPosition!, count) \\
stepping'
\end{array}
$$

Once a new location has been calculated, the object is placed into a 'stepping' mode. Once in this mode, the next point in the span to be visited must be calculated, bearing in mind requirements related to synchronization elements. A succinct description of this operation is given below.

$\Phi step$

$$
\begin{array}{l}
\Delta(point, span, loopCounter) \\
\hline
stepping \land point \neq requiredPosition \\
point \prec point' \\
\neg\, requiredPosition \prec point' \\
\textbf{let}\, skipped == \{loc : span \mid loc \prec point'\} \bullet \\
fst(\!| skipped |\!) \cap \dom refpoints = \varnothing \\
loopCounter' = loopCounter \\
\quad + |snd(point') \ominus snd(point)| \\
span' = span \setminus skipped
\end{array}
$$

Stepping mode is an artefact of the specification introduced to model the sequence of operations that are assumed to take place internally. The decision to use this approach raises an issue that goes beyond ones' taste in specification language, and which has implications for other standards and systems, for example the VRML 2.0 standard [48] under development in SC24. In developing a standard, particularly in an area such as graphics where performance is a non-trivial concern, there may be implicit assumptions about the execution model that will be used to realise the system. In the case of PREMO for example, the specification of *progressPosition* given above, and the semantics of the internal *stepping* mode, involve a level of operational detail that normally one would associate more with a design or implementation. The problem is that a more abstract description of the intended behaviour may be rather more difficult for committee members or even implementors to understand; state machines are after all a well understood engineering concept, a fact that has been borne out by the experience of others in designing languages to document requirements and specifications [49].

Three further 'internal' schemas are used in [36] to define the semantics of stepping mode; other operations within the *Synchronizable* object type define moding behaviour, access and modification of the attributes that define the span, and control over the placement of synchronization elements. Two of the main operations for controlling mode are presented below, as they will be revisited in the next section and also illustrate a further point about the application of formal methods in PREMO. The first pair of operations allow the object to be placed into the state 'PLAY' or 'STOPPED'. The former can only be achieved when the media object is stopped; if this condition is not met, an error is raised. A media object can however be stopped when it is in any state. Stopping an object causes its position and loop counter to be reset to their initial values, and therefore requires 'internal' variables to be updated.

play

$\Delta(currentState)$
$\Phi DoAction$
$\Phi UpdateSpan$

$currentState \in \{PLAY, STOPPED\}$
$\quad \Leftrightarrow\!\!\!\rightarrow$ $\boxed{\text{exc}}$ WrongState

---

$currentState' = PLAY$
WrongState $\Leftrightarrow\!\!\!\rightarrow$ $currentState' = currentState$

stop

$\Delta(currentState, loopCounter, currentPosition,$
$\quad stepping)$
$\Phi UpdateSpan$
$\Phi DoAction$

$currentState' = STOPPED$
$loopCounter' = 0$
$currentPosition' = loopStart$
$\neg\ stepping'$

The operation *play* uses some new notation for specifying exceptions. The notation is described in [33], but is not important for the purposes of this paper.

If the object is PAUSED or WAITING, then it can only react to a very restricted set of operation requests: the attributes of the object may be retrieved (but not set) and the resume, pause or stop operations may be invoked, which may result in a change in state. The difference between PAUSED and WAITING is that, in the latter case, the object returns to the place where it had been suspended by a *Wait* flag, whereas, in the former case, a complete new processing stage begins. The differentiation between these two states, i.e., the usage of the *Wait* flag, is essential; this mechanism ensures the timely control over the behaviour of the object at a synchronization point. If the object could only be stopped by another object via a pause call, an unwanted race

condition could occur.

pause

$\Delta(currentState, stepping)$
$\Phi DoAction$

$currentState \neq STOPPED$
$\quad \wedge\ currentState' = PAUSED$
$\quad \wedge\ \neg\ stepping'$
$\vee$
$currentState = STOPPED$
$\quad \wedge\ currentState' = currentState$

resume

$\Delta(currentState)$
$\Phi DoAction$

$currentState \in \{PLAY, PAUSED, WAITING\}$
$\quad \Leftrightarrow\!\!\rightarrow$ exc WrongState

$currentState' = PLAY$
WrongState $\Leftrightarrow\!\!\rightarrow$ $currentState' = currentState$

Examples of how the general specification presented here could be specialized to address specific media are considered in Section 5. Some appreciation for the effect that formally specifying these facilities has had on the PREMO standard can be gained by comparing the specification presented in [36] with the specification developed from the original proposal for the synchronization facilities [50].

Timers and TimeSynchronizable Objects PREMO defines a clock object type which provides an interface to whatever notion of time is supported by its environment. The object type supports an operation, *inquireTick*, which returns the number of ticks that have elapsed since the start of the PREMO era (defined in the standard). In the formal description, an 'internal' operation $\Phi Tick$ is introduced to represent the progression of time by requiring that the number of ticks elapsed after the operation occurs is greater than the number before. The specification does not currently relate invocation of this operation to the passage of time in the environment. Object-Z is not based on a real-time model, and so the approach taken to specifying aspects of PREMO related to time is perhaps closest to that advocated by Abadi and Lamport [51]. Time and time units are introduced as given types;

$[Time]$

$[TimeUnit]$

a 'clock' object type is then defined with a state variable that explicitly captures the current time. This specification is not complete, in that we have not defined the relation '$>$' used in $\Phi Tick$. However, at the level of abstraction that seems suitable for describing PREMO operations, there seems little more that can usefully be added.

Clock

$tickUnit : TimeUnit$
$accuracyUnit : TimeUnit$
$accuracy : Time$ [read-only]

$ticks : Time$

$\Phi Tick$

$\Delta(ticks)$

$ticks' > ticks$

inquireTick

$ticks! : Time$

$ticks! = ticks$

The approach taken here, which does not include mention of real-time, may seem fundamentally inadequate given the blatantly central role of time in the definition of continuous media and in reasoning about the behaviour of systems that process and integrate media; the whole issue of quality of service for example has lead to significant work on logics to address the temporal aspect of media [52, 53]. However, in the context of this specification, use of a real-time logic would be inappropriate. First, with the exception of the small number of object types that utilise time, the descriptive power of such logics would not be called on. However, as PREMO utilises inheritance to construct new object types, object types with a real-time behaviour may be extended, and combined with other object types; for example PREMO defines a *Timer* object type as a combination of *Clock* and a state machine with RUNNING, STOPPED and PAUSED states. This creates two options for the specifier: either **all** object types that might conceivably be combined with an object type with a real-time basis are described in the common real-time logic, or take an approach similar to that adopted with the object model and adopt conventions for linking specifications written in different formalisms. The first option imposes an inappropriate level of complexity and rigour on the whole specification, while the second seems to contradict the desire for greater formality that lead to the adoption of a real time logic in the first place. In other words, although the current specification does not have a real-time foundation, it represents a compromise between ease of description and level of rigour that is appropriate to the task.

3.4 A LOTOS Specification of the Synchronizable Object Type

To this point, the use of formal methods described in this paper has not involved tool support beyond that of mathematical typesetting. It is certainly true that using, for example, a Z type checker, would have improved the quality of the specification by identifying certain errors. Had we been in a position to derive the normative text from the specification, this approach would have proven attractive. However, it should be clear from the account of the work given thus far that the nature of the standards process, and the resources available to the specification group, made a lightweight approach inevitable. Further, to improve the clarity of the specifications, we have adopted conventions that are not supported by software tools. We will review the issue of cost-benefit trade-off in the conclusions, but one particular aspect of the standard that seemed to require a more thorough analysis than that afforded by the Object-Z specification alone was the behaviour of the synchronizable objects described in the previous subsection, and in particular the moding behaviour arising from the explicit operations such as *play*, and the internal transitions arising from synchronization points.

Two of the authors (GF and MM) have considerable experience with the formal description technique LOTOS [54, 29]. LOTOS is particularly well supported by tool environments, which include the verification environment LITE [55], the action based model checker (X)AMC [56], and the Autograph tool [57, 58] that can generate a graphical representation of the automaton described by a LOTOS specification. These tools were used in combination (see figure 2) to model and analyse the behaviour of PREMO synchronizable objects. The process and results of this analysis are reported in this subsection, which is followed by a comparison between the Object-Z and LOTOS approaches.

Modelling the PREMO Synchronizable Object In order to maintain a clear, although not formal, relation with the Object-Z specification of the PREMO synchronizable object we chose to follow a constraint oriented style of specification [59]. In this style behaviour is modelled as constraints on the temporal ordering of actions. We let each value of each control variable correspond to one process. For example the control variable "current mode" (the variable *currentState* in the Object-Z specification) can assume four different values: *PLAY*, *STOPPED*, *PAUSED* and *WAITING*. This resulted in four different processes, one for each value. The names of the Object-Z schemas which represent the methods that can be invoked to change the mode (*play*, *stop*, *pause*, *resume*) have been modelled as actions (*doPLAY*, *doSTOP*, *doPAUSE*, *doRESUME*). The action *doWAIT* is

introduced in the LOTOS specification to represent the internal, i.e. non-observable, transition to the *WAITING* state. Similarly an action *donePlay* is introduced to model completion of the *play* operation which causes a transition to the *STOPPED* state.

Each LOTOS process is structured as a non-deterministic choice of possible actions that can occur in the state represented by that process. For example only the actions *doSTOP* and *doPLAY* can be invoked in *STOPPED* mode, and thus the process representing this mode was modelled as a choice of a *doSTOP* and a *doPLAY* action, after which a process is invoked that corresponds to the mode entered by that action. So, after *doPLAY* the process *PLAY* is invoked. Further, every process is enabled to signal an exception that models the case of method invocation from a 'wrong' state.

process *STOPPED* [*doSTOP*, *doPLAY*, *doPAUSE*,
doRESUME, *doWAIT*, *donePlay*, *exc*] :
noexit :=
doPLAY; *PLAY*[...]
[]
doSTOP; *STOPPED*[...]
[]
exc; *STOPPED*[...]
endproc

Specifications of the processes corresponding to the other modes are given in [60].

Modelling the mode transitions alone gives a first core specification of the synchronizable object. The main body of the core specification consists essentially of initiating the process modelling the *STOPPED* mode since that is the initial state of the synchronizable object. The hide-operator specifies that the actions *doWAIT* and *donePlay* are internal actions within the process *modeTransitions*.

process *SynchronizableObject*
[*doSTOP*, *doPLAY*, *doPAUSE*, *doRESUME*, *exc*] :
noexit :=
hide *doWAIT*, *donePlay* **in**
modeTransitions[...]
endproc

process *modeTransitions*
[*doSTOP*, *doPLAY*, *doPAUSE*, *doRESUME*, *doWAIT*,
donePlay, *exc*] :
noexit :=
STOPPED[...]
endproc

This core specification captures the state transition aspects of the Object-Z specification.

The core specification is extended with a process called *progressPosition* which models the internal progression through the coordinate space of the synchronizable object. The approach taken is to model the events that trigger a change in the behaviour of the object, rather than the precise way in which the object progresses through the coordinate space. The specification thus models the fact that certain events *can* occur, rather than exactly *when* they will occur. In the Object-Z specification, when an object is in *PLAY* mode, the *progressPosition* operation computes the next location in the coordinate space to be visited. This is called the required position. In the LOTOS specification, the computation of the required position is denoted by the action *target*. Progress towards a target is achieved in a sequence of steps called a stage. The action *doneStage* models termination of progression when the required position is reached. During a stage two kinds of action may occur, *doSignal* which models the signalling of a reference point and *doStep* which models the progression

of the current position to the next stepping point. Processes *NOTSTEPPING* and *STEPPING* correspond to the two possible values of the variable *stepping* in the Object-Z specification.

The process *progressPosition* is added to the core specification in a constraint oriented way using parallel composition with synchronisation over those actions on which this process is imposing extra behavioural constraints.

An outline of the specification is given below; much of the detail of the *STEPPING* process in particular has been omitted. For details the reader is invited to consult [60] where the full specification is described.

process *SynchronizableObject*
[*doSTOP*, *doPLAY*, *doPAUSE*, *doRESUME*, *exc*, *doSignal*] :
noexit :=
hide *doWAIT*, *donePlay* **in**
modeTransitions [*doSTOP*, *doPLAY*, *doPAUSE*, *doRESUME*, *doWAIT*, *donePlay*, *exc*]
|[*doSTOP*, *doPLAY*, *doPAUSE*, *doRESUME*, *doWAIT*, *donePlay*]|
progressPosition [*doSTOP*, *doPLAY*, *doPAUSE*, *doRESUME*, *doWAIT*, *donePlay*, *doSignal*]
endproc

process *progressPosition*
[*doSTOP*, *doPLAY*, *doPAUSE*, *doRESUME*, *doWAIT*, *donePlay*, *doSignal*] :
noexit :=
hide *doStep*, *doneStage* **in**
NOTSTEPPING [*doSTOP*, *doPLAY*, *doPAUSE*, *doRESUME*, *doWAIT*, *donePlay*, *doSignal*, *doStep*, *doneStage*]
endproc

process *NOTSTEPPING*
[*doSTOP*, *doPLAY*, *doPAUSE*, *doRESUME*, *doWAIT*, *donePlay*, *doSignal*, *doStep*, *doneStage*] :
noexit :=
doPLAY; *NEWTARGET*[...]
[]
doRESUME; *NEWTARGET*[...]
[]
doSTOP; *NOTSTEPPING*[...]
[]
doPAUSE; *NOTSTEPPING*[...]
endproc

```
process NEWTARGET
    [doSTOP, doPLAY, doPAUSE, doRESUME, doWAIT,
     donePlay, doSignal, doStep, doneStage] :
noexit :=
    hide target in
    target;
    STEPPING [doSTOP, doPLAY, doPAUSE,
              doRESUME, doWAIT, donePlay,
              doSignal, doStep, doneStage]
endproc

process STEPPING
    [doSTOP, doPLAY, doPAUSE, doRESUME, doWAIT,
     donePlay, doSignal, doStep, doStage] :
noexit :=
    doStep;
        (STEPPING [doSTOP, doPLAY, doPAUSE,
                   doRESUME, doWAIT, donePlay,
                   doSignal, doStep, doneStage]
        []
        doSignal; ...
        )
    []
    doneStage;
        (NEWTARGET[...]
        []
        NOTSTEPPING[...])
    []
    donePlay; NOTSTEPPING[...]
endproc
```

Tool support for verification The textual basic LOTOS specification can be presented in different ways. The full Lite environment provides both tools for graphical representation and tools for simulation and for the generation of Extended Finite State Machines (EFSM). The simulator can be used to derive an action tree (where each path in the tree represents one of the traces that can possibly be generated) and to produce an EFSM which is strong bisimulation equivalent to the original behaviour expression. Extended Finite State Machines can be translated automatically into an automaton in a format (FC2) suitable for input to the JACK [61, 56] toolset. This set comprises (M)AUTO, Autograph and the (X)AMC model checker. In particular, we can automatically generate graphical representations of the automata with the Autograph tool such as the one presented in figure 1.

More important, we can use the (X)AMC model checker to automatically verify properties of a specification that are written as ACTL formulas. ACTL is an action based temporal logic. Whenever a formula does not hold, the model checker can produce a trace of actions that shows the violation of the formula. This provides useful information on how to improve the specification if errors are found. Figure 2 outlines the various transformations of the LOTOS specification and the verification tools we used to analyse and develop the specification of part of the PREMO synchronizable object.

We have checked some of the properties described in the standard. One of them, for example, characterizes the difference between resuming to *PLAY* from *PAUSED* mode, versus from *WAITING* mode. If the synchronizable object makes a transition to *PLAY* from *PAUSED*, it has to compute immediately a new required position before it continues progressing through the coordinate space. However, if the synchronizable object makes a transition to *PLAY* mode from *WAITING*, there should still be a required position to be reached, and

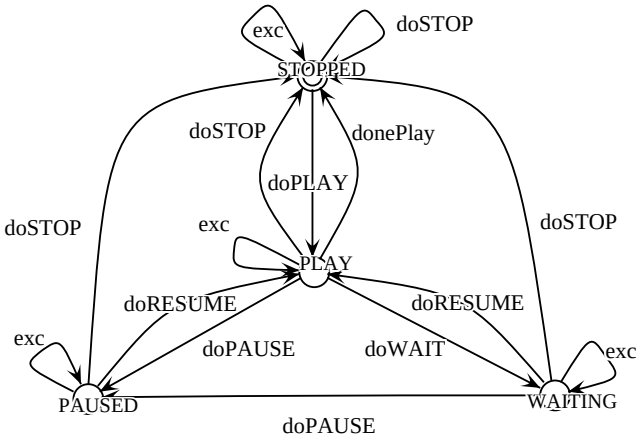

Figure 1: Mode transition diagram.

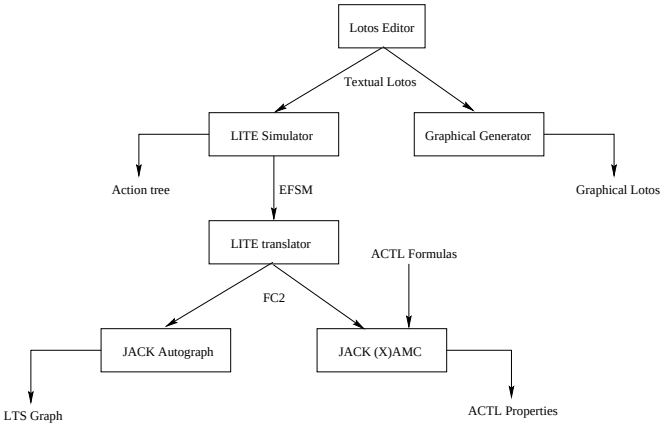

Figure 2: Specification and Verification Tools.

the process should continue progression directly, *without* first computing a new required position. The first situation can be characterized informally as *"after a doRESUME from the pausing state, always a new target is computed before stepping is continued"*. The second situation is *"after a doRESUME from the waiting state it is* not *the case that first a new target is computed before stepping is continued"*.

In ACTL we have to rephrase these statements in terms of actions that can be observed during all possible execution paths of the specification seen as a transition graph. This may sometimes be a rather complicated task. For example, to state the first requirement, we first observe that "continuing stepping" means, in terms of actions, that we can observe either a *doStep* action, indicating real stepping, or the actions that indicate the end of the stepping process which are *doneStage* and *donePlay*. The computation of a new required position is modelled by the action called *target*. We now have to express that for all execution paths in all states for all next states that can be reached by a *doPAUSE* there does not exist a path (from that state on) for which we *cannot* observe a *target* action until we can observe a *doStep*, *donePlay* or *doneStage* indicating the continuation of stepping. Formally in ACTL:

$$
\begin{aligned}
&AG[doPAUSE][doRESUME] \\
&\quad \sim E[\, true \\
&\qquad \{\sim target\} U \{doStep \mid donePlay \mid doneStage\} \\
&\qquad true]
\end{aligned}
$$

The model checker can check automatically whether this property holds for the specification. Although the formulation of ACTL expressions can be rather complex, it is still much less work than any attempt to prove the property by paper and pencil.

Model checking may not only be helpful in the final verification of the properties of a specification but also during the development of the specification. In this context we used model checking to obtain insight into the consequences of adding a jump operation to the set of operations; informally, *jump* changes the current position of a synchronizable object in its coordinate space to a value specified as an argument. We investigated the situations in which the invocation of a *jump* can be allowed without causing unexpected side effects. A possibly dangerous situation can occur when the *currentPosition* pointer is changed after a new required position has been computed and before the *currentPosition* pointer has been updated to a stable value, either by reaching the required position, or by encountering the end of the coordinate space (modelled by *donePlay*) or by a *doSTOP* or a *doPAUSE* action which may both occur when the synchronizable object is in *WAITING* mode. This particular situation can be characterized by an ACTL formula which consequently can be checked against different "prototypes" of the specification.

The different prototypes are obtained by adding a different process that models different conditions in which *jump* can be invoked. For example, in one variant, *jump* could be made available always, and in another it could be made available only during *STOPPED* and *PAUSED* modes. By means of such tuning the least restrictive condition was found that prevents occurrence of the dangerous situation described above. In this particular example this condition was that a jump operation can be allowed whenever the process is *not* in the stepping stage, i.e. not between a target action and a consequent *doneStage*, *donePlay*, *doSTOP* or *doPAUSE* action. For details the reader is invited to consult [60].

This kind of analysis may help in avoiding the formulation of too restrictive requirements in a standard, and makes it possible to analyse different options in a rigorous way before a final design decision is made.

3.5 Comparison

As with any design representation, selecting a formal notation involves trade offs between, for example, ease of expression of particular aspects of behaviour, and clarity versus ease of analytic power. No single formal method is suitable for the description and analysis of every aspect of a complex system. The Object-Z specification language has been advocated for the description of standards [62, 10] because of its accessible specifications combined with natural language explanations, the expressiveness of the underlying set theory, and its use of object orientation. A drawback of the language is that it is (relatively) difficult to develop tools for the behavioural analysis of specifications. Syntax and type-checkers, developed for Z [27], are not sufficient for a rigorous check of behavioural properties.

A combination of specification formalisms and supporting tools may, in certain circumstances, be helpful in the development of a specification. Certainly, this was our experience in the context of PREMO. Of course, such an approach brings about questions on the formal relationship between the combined formalisms, and indeed, Clarke and Wing [42] identify this area as one of the key challenges for formal methods at the present. Although much work remains to be done here, the combined use of methods may have many advantages even if the relationship between the notations involved is not (yet) formally defined. Different formalizations of the same problem offer different insights, simply because the different notations, methods and tools involved cause the specifier to view and express the problem in a variety of forms. As our experience with PREMO has illustrated, the application of methods with analytic leverage (software tools, for example) to critical aspects of a system can offer a payback in terms of revealing unexpected subtleties. Such methods can be 'expensive' in that the level of formality needed for automated analysis requires more time to be spent in the development of the model than for 'lightweight' approaches. As we have described however, an initial specification, even in a lightweight style, can be useful in directing this more expensive effort to those aspects of the system where it will be most effective.

4. Multimedia System Services

No attempt has yet been made to give a formal description of any aspects of the third part of PREMO, Multimedia System Services (MSS). The reason for including this section in the paper is to look at some of the difficulties that would arise were such an attempt to be made, and also to look at the difficulties that arise in trying to integrate a document from a different source (in this case the Interactive Multimedia Association) in an emerging standard. This issue is important in the context of the current paper, as it is increasingly common for initial drafts of standards or components of standards to originate from outside the ISO/IEC Committee responsible for the standard. In this section we describe the key provisions of this component of PREMO and discuss the issues that they raise for formal description.

4.1 Background

The essential goal of the MSS is to establish a model for a dataflow-like processing network of abstract entities, which together build up a complete multimedia processing environment. It would 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 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., they 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 as a whole. Using this mechanism, the client has the possibility to set up specialised processing networks, adapted to the task at hand and the resources available.

Media streams flow among virtual devices; this flow is controlled by separate constituent objects, called *StreamControl*. These objects act as a controlling point for an event-based synchronization mechanism, aspects of which have been mentioned in previous sections (for example, the *Synchronizable* and *EventHandler* object types).

There are other objects defined in the MSS which provide other means of controlling a processing network. For example, virtual connections act as an abstraction to set up specific networks; the *Group* object type provides a single entry point for a cluster of virtual devices. In a PREMO application, all of these objects can be spread over a real network, i.e., they can form the basis for a really distributed multimedia environment.

It is the 'history' of this part of PREMO rather than the content which is of particular interest from the point of view of this paper. As already noted, the original specification was developed and 'donated' to ISO by the IMA (International Multimedia Association), which is a large, industrial association of multimedia system and application vendors. When the specification arrived at ISO, all of the major concepts were present, in a mature form, and had also undergone an experimental implementation, at least for some portion of the specification. However, the specification text itself was far from complete; there were a number of missing or ambiguous

provisions, non documented features based on implicit “common understanding”, reliance on external, albeit incomplete specifications (e.g., some object service specification of OMG, which were not yet fully available at that time), etc. Also, the IMA version of MSS relied on one single environment (namely the OMG Object Services and CORBA specifications) and thus lacked the generality required by an ISO standard.

Consequently, converting the original IMA specification into a full-blown ISO Part required a significant amount of work. A number of details had to be defined, and the general object model of PREMO, which was already available at that time, formed a solid basis for this work. It became very clear that the time and energy spent on the development of a precise object model, together with the formal specification, for event synchronization, etc, paid off very well, and helped to turn an incomplete, albeit technically very mature and exciting working document into a solid, and precise specification. In fact, by doing so, the original IMA document was enriched with new features, derived from the general facilities of PREMO, and unsuspected even by the original designers of the IMA version of the Multimedia Systems Services.

This exercise was also beneficial for the PREMO document. The review process for the MSS component forced the designers of PREMO to add new features to the general PREMO facilities, which proved to be extremely helpful for the full PREMO model, and it provided the designers of PREMO with some new tools and concepts which became fundamentally important for other parts of the PREMO document. Some of these features have since received a more formal specification, too, and there are plans to complete these specifications in the future. The following subsections cite some examples of the concepts taken over from the IMA document.

4.2 Interfaces and types

MSS laid great emphasis on the difference between interface and type; a distinction that was blurred by C++ (except for very well-informed users), though the widespread usage of Java is making the distinction better-known.

The conceptual separation between interface and type was already present in the PREMO object model, but it was MSS which forced the PREMO designers to make effective use of this conceptual difference. 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, to be precise, include all possible operations for all types. In other words, if an implementation aims at being compliant with PREMO, but would find it necessary to add new operations to a specific object, this could only be done 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. The question that this raises is one of conformance testing. Preliminary experiments in implementing parts of the Standard have raised a number of issues concerning what it means for a program in an object-oriented language to implement a Standard that contains its own object model, particularly where the model used by the language differs from that of the standard. Further discussion of the issues involved is beyond the scope of the present paper.

4.3 Property Management

Properties are used to store values with an object; they differ from attributes or variables in that they may be created or removed dynamically, and are thus not subject to static type checking. A property is a pair, consisting of a key (i.e., a string) and a sequence of values which can include object references. Conceptually, properties are 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 property 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 the negotiation mechanism in PREMO. 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 use properties? The fundamental reason lies, in fact, in the conservative nature of the PREMO object model. In PREMO, operations on a type are defined statically, when defining ("declaring") the object. The PREMO framework does not require an environment to support run-time changes to object or type structure, such as adding or removing an operation.

On the other hand, it has been advocated elsewhere that more dynamic object models should be used for graphics or multimedia (see, [32, 63]). Indeed, the use of delegation or, on a more "modes" level, a more dynamic view of objects like, for example, the approach adopted in Python [64] (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 [32] have also shown that implementing such features on 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. Clearly the implementation of properties does not pose significant problems. The experience with the specification has also shown that the dynamicity offered by properties seem to be quite appropriate for PREMO. Consequently, properties play a somewhat less elegant, but very useful role in PREMO in increasing the dynamic nature of object instances.

Although it is of course possible to specify a dynamic object model, it is quite another matter to develop (object oriented) specifications in which the underlying object model is dynamic, in the sense of allowing the structure of objects modelled in the notation to vary. Object oriented systems are increasingly offering dynamic capabilities, based on reflective capabilities (e.g. the reflection package of Java), meta-object protocols, and the use of delegation and prototypes. While it could be argued that these systems support a design culture far removed from that of formal specification and rigorous development, systems such as PREMO illustrate the potential for these to come into contact. The development of specification techniques and design methods to safely and effectively utilise dynamic object models is an open problem.

4.4 Formal Specification of Dataflow-like Environments

Media processing elements in PREMO 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. To this end, the MSS component of PREMO defines object types to represent resources, devices, ports and streams, but at an abstract rather than a concrete level. That is, the standard defines a minimal interface needed to build a network of devices without prescribing, e.g., the operational details of how a port operates or how a device processes data. Such details are left to implementation-specific subtypes; the intention is that the standard provides just enough information about the way that devices are connected in such a network to enable a network to be configured by a negotiation mechanism that utilises properties of the object types involved. As a result of this approach, a PREMO system resembles a dataflow network, with devices and their subtypes as the processing elements, connected by data streams that incorporate the synchronization mechanisms discussed in earlier sections of this paper. This "dataflow" approach is not new to PREMO, it appears in published approaches to multimedia systems (for example, [43]), 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 - a so-called 'plug and play' framework.

The dataflow model underlying the MSS framework presents two distinct challenges to formal description techniques. First, the interface and properties of MSS object types are not sufficient in themselves to characterise the behaviour of these devices, and it is unclear at present what style of specification would be most suitable. Recall that in the case of *Synchronizable* objects, it was necessary to introduce a rich 'specification' state to model the behaviour set out informally in the normative standard. However, the MSS component is rather less prescriptive, and it seems that there is little that can be said about the behaviour of, for example, a virtual device, that does not involve over-commitment. An alternative approach is to view the MSS component,

not so much as a set of object types, but as a software architecture that provides sets of building blocks, and connectors through which those blocks can be assembled into systems. This approach to system models has gained much ground from the work of Shaw and Garlan and [65]. Recent work on formal foundations for architectures [66] may support a quite different approach to modelling systems like PREMO.

Both of the options described above rely on descent into, or at least to the level of, the PREMO object model. In order to understand the behaviour of a PREMO system in terms of the dataflow paradigm that, at least conceptually, is offered by MSS, alternative approaches to specification are needed. Two lines of work seem potential candidates for this. The first is the work of Milner on calculi for mobile processes [37] which has lead more recently to the concept of action system and control structure in which it seems possible to understand both the concept of communication channel (for message passing) and dataflow. The second line of work is that on coordination languages and their models. These languages, such as Linda [67] and Manifold [68] have the property that they abstract communication and sharing of data from the problem of processing that data, and this is essentially what the MSS component sets out to achieve for media data. Approaches to the specification of coordination have been explored in [69].

Progress towards the formal description of MSS has importance beyond the PREMO standard. We believe that specification is most effective when the formalism allows the phenomena of interest to be expressed without introducing unnecessary distinctions and artefacts from complex encodings. For example, the π calculus succeeds because it directly captures fundamental properties of communication and concurrency; we would like something that achieves the same for MSS, integrating the description of dataflow with aspects such as negotiation, quality of service, and where necessary the underlying object model of PREMO (for example, event handling). This may just be a matter of finding the right synthesis of existing theories and methods; the work of Blair et al [53] for example may provide an appropriate foundation of some of these concerns. However, certain pieces of this jigsaw may still be missing, or in need of refinement. In particular, describing and understanding the behaviour of a distributed multimedia application will at some point require discussion on media content and presentation. This piece of the puzzle is the subject of the next section.

5. Media Content

PREMO includes a framework for media primitives that provides a basis for the derivation of object types for representing specific media content. The framework is abstract, in that it does not itself support the representation of specific media data. However, any implementation or domain-specific component developed on top of PREMO is likely to introduce media content, and so the question arises as to how this content could be incorporated into the specification. Work on the specification of primitives for graphics standards has been reported (for example, [23]), but has not had to address issues such as time, which is fundamental to the data and presentations that PREMO is expected to process. Indeed, following on from the discussion on dataflow within MSS, it would be desirable to find a general specification framework for describing temporal media, rather than developing ad-hoc solutions.

This section describes our initial work towards such a framework for a limited set of media object types, but which encompasses continuous, discrete and digital media. The approach is described in detail in [35], and was developed using a class hierarchy derived from a multi-media system called MADE [32]. The class names have been retained from the original paper.

The object type that forms the basis of the media hierarchy is called *MMedia*. This provides a model of a 'content space' that is similar to the progression space of the PREMO *Synchronizable* object. However, *MMedia* goes further, and associates content with certain locations in the space. The structure of media content is not of concern here, and content is represented by a generic type *P*; the specification, which is taken directly from [35] also extends a type *MSyncObject* that effectively defines a simple model of synchronization elements that is not needed here.

Although presentations such as video recordings or animations can be understood in terms of discrete units (frames) it is useful to remain unbiased towards any concrete model of progression space. For this reason, a generic type T is used as the domain over which the contents of a media object are defined. We require, however, that a class define some successor relation $\preceq$ over T. It is assumed that $\preceq$ is a well-ordering in the sense that any subset S of T has least and greatest elements denoted $inf(S)$ and $sup(S)$; it is not necessary

that $\{inf, sup\}(S) \in S$. Shortly we will describe how T can be realized as a linear sequence or how it can be replaced by a continuous model.

MMedia$[T, P]$

MSyncObject $[T]$

$rendering : P$ [visual presentation of the object]
$content : T \pfun P$ [temporal extent]
$mode : status$ [control mode]
$posn : T$ [position within media object]
$begin, end : T$ [locators]
$\_ \preceq \_ : T \rel T$ [successor relation over T]

$$\forall i, j, k : T \bullet \begin{cases} i \preceq i \\ i \preceq j \land j \preceq i \Leftrightarrow i = j \\ i \preceq j \land j \preceq k \Rightarrow i \preceq k \end{cases}$$
$begin, end \in \mathrm{dom}\, content$
$begin \preceq posn \preceq end$
$mode = play \Rightarrow rendering = content(posn)$

INIT

$begin = inf(\mathrm{dom}\, content)$
$end = sup(\mathrm{dom}\, content)$ [infimums and supremums]
$mode = stop$
$posn = begin$

play

$\Delta(rendering)$
$actions! : \mathbb{F}\, EVENT$

$mode = play$
$posn \preceq end$
$posn \preceq posn'$
$actions! = synchro(posn')$

stop

$\Delta(mode, rendering, posn)$

$mode = play \lor posn = end$
$posn' \preceq posn$
$mode' = stop$

$\Diamond \Box\, \mathrm{pre}\, play \Rightarrow \Box \Diamond\, \mathbf{op} = play$

The state invariants define the successor relation over T, locate the begin and end points of the presentation of an object within its content, and define the rendering of the object to be the contents of the object at the particular position reached whilst the object is playing. These are very general properties of a multimedia object, independent of specific details of the media type. Only the *play* operation need to be considered to illustrate the approach. This operation advances the state of the presentation (using the successor relation on the progression domain) to the next synchronization point. Some comments are in order:

- The successor relation over the progression domain T is defined to be a partial order. This allows the

description of systems that combine aspects of hyper-media and multimedia, for example, games or simulations where the audio and visual presentation is determined by the actions of the user.

- The temporal predicate is a fairness requirement; it indicates that if *play* is eventually always enabled then it is always eventually performed.

The basic multimedia class can be specialized to describe both discrete and continuous presentations. The continuous case is introduced first, and is based on the timed history model of [70] in which time is represented by the positive real numbers; $TIME == \mathbb{R}^+$. The content of a continuous presentation is then modelled as a continuous function of values drawn from the presentation type P over time.

ContinMMedia[P]

MMedia[$TIME, P$]

$$content \in TIME \twoheadrightarrow\!\!\!\!\!\circ\;\; P$$

The symbol '$\twoheadrightarrow\!\!\!\!\!\circ$' denotes the set of topologically continuous functions; if P itself is continuous (say $P = \mathbb{R}$, representing some signal level) then *content* may also be total. If P is discrete then *content* will be a step function which maintains values drawn from P over finite intervals and which is undefined between such stable intervals. See [70] for the definition of topological continuity and continuous functions in Z.

A discrete media object can be defined by instantiating T to be the natural numbers and $\preceq$ to be the usual $\leq$ relation. In Z, a sequence is a function whose domain is some initial prefix of the natural numbers and it is useful to require that the *content* function now represents a sequence of presentations. This means that the post condition on *play* can be strengthened to state that the next position in the content is selected.

DiscrMMedia[P]

MMedia[$\mathbb{N}, P$]

$$content \in \operatorname{seq} P$$
$$(\preceq) = \leq$$

play

$$posn' = posn + 1$$

It is also possible to model the discrete presentation of a continuous medium; the issue here is how the *content* sequence should be related to the continuous model. The approach we take is to posit a mapping *frame* from points in the sequence to open intervals of time (see Figure 3) such that:

- *frame* is monotonic, that is, as we progress through the sequence we also make progress in time,
- the supremum of any interval is the infimum of the next,
- the infimum of the first interval is *begin* and the supremum of the last interval is *end*,
- the presentation at any discrete point is some function of the presentation over the corresponding interval. This function may simply be some sampling of the contents at some point in the interval or it may involve some 'averaging' of the presentation.

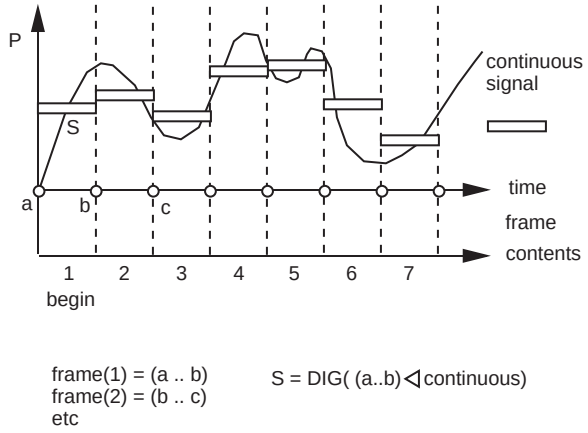

Figure 3: Digital Presentation of Continuous Media.

DigMMedia[*P*]

DiscrMMedia[*P*] [Inherit discrete multimedia base class]

$$
\begin{array}{l}
continuous : TIME \twoheadrightarrow P \\
frame : \mathbb{N} \rightarrow \Im TIME \\
sample : (TIME \twoheadrightarrow P) \rightarrow P \\
\hline
\mathrm{dom}\, frame = \mathrm{dom}\, content \\
frame \neq \langle\,\rangle \Rightarrow \\
\quad inf(frame(head\, frame)) = begin \\
\quad sup(frame(last\, frame)) = end \\
\quad \forall i : \mathrm{dom}\, frame \mid i > 1 \bullet \\
\qquad inf(frame(i)) = sup(frame(i\text{-}1)) \\
\forall i : \mathrm{dom}\, frame \bullet \\
content(i) = sample(frame(i) \lhd continuous)
\end{array}
$$

For any frame *i*, the presentation of *i* is a sample of the continuous data recorded for the interval over which *i* is defined.

The group of models presented in this section encapsulate a first attempt at responding to Zave's challenge for mathematical models of the application domain (here, media data). Such a model can clearly be related to PREMO-like facilities such as synchronization; it remains to be seen however whether the model captures important aspects of media data, and how it will work in concert with, for example, models of media dataflow.

6. Conclusions and Open Questions

The introduction to this paper began with quotations from four experts, each of which identified a significant factor in the use of formal methods. Shortly we will revisit these factors in the context of the work that we have reported on the specification of PREMO, but first, we will summarise the main points arising from of our experience:

- We found it helpful to utilise a variety of techniques, both state-based and model-based, rather than adopting a single formalism - even though this limited the scope for tool support.
- Through the process of building a specification, we were able to identify deficiencies in PREMO, to put forward convincing arguments to the standards committee on the nature of the deficiency, and to

propose solutions. Being required to 'stand back' and understand the relationships within the standard was particularly valuable in three areas:

  - in understanding of object model and role it plays in PREMO;
  - in abstracting out the notion of progression space that underlies the synchronizable objects; and
  - in identifying commonality in behaviour between proposed object types (particularly within the hierarchy beneath *Synchronizable*), and factoring this out by modifying the hierarchy accordingly.

- The descriptive style in the PREMO document drew on the Object-Z notation, and the informal comments needed to document the formal specification helped improve the presentation of the normative text.
- We did not produce a full specification of the system, as:
  - we had neither the time or the personnel to complete the task;
  - the usefulness of the specification was constrained by the number of people able to comment on and critique, as opposed to just read, the formal material; and
  - there were a number of difficult technical problems in the specification itself, not the least being uncertainty initially about what notation to adopt, and then later concerns over the integration of the various models.
- We have proposed and experimented with techniques for describing multimedia data and systems.

Let us now relate these points to the factors identified by Jones, Hall, Zave and Dill.

1. Jones emphasised the importance of choosing an appropriate level of rigour and formality. Our experience has much in common with his report on the usefulness of abstract descriptions of state; for both the object model and synchronizable objects, we found that, once we had understood the state space model, we had a much clearer understanding of how the operations over that state should behave. In all honesty however, we should also point out that our lightweight use of formal methods was also because we still do not know how best to tackle the description of PREMO in a fully formal way, at least within the constraints imposed by the nature of standardization and the ISO process.
2. Hall raised the need to ask the 'right' questions about the use of formal methods, and in particular, how formal methods can improve the quality and decrease the cost of a system. Our initial view was that formal methods would provide a way of clarifying our understanding of a complex system, and of documenting the behaviour of components of that system in a way that would allow us to argue about the choices made in the design of such components. The latter was particularly important, as the resource limitations that have already been mentioned also made it impractical to develop a reference implementation in parallel with the text. It is also important to note that, from the outset, there was no aim of carrying out formal proof or refinement. Our claim is that the use of formal methods did indeed improve the quality and reduce the cost (at least, in terms of person-effort) in the development of PREMO.
3. Zave identified the importance of mathematical models of the application domain to underpin the use of formal methods, and indeed this is one of the key problems that we encountered when beginning the specification process. Quite simply, we were fortunate that the most troublesome aspects of PREMO (the object model and synchronization facilities) were largely independent of media data, and could therefore be tackled in terms that were familiar to the participants (models for the semantics of object-oriented systems, and state machines, respectively). As we have discussed however, there are significant areas in which a well defined mathematical model is still required, not the least being models of dataflow architecture and of media content. Indeed, it should also be stressed that the linkage between the object model and remainder of the specification is at best inelegant, and a better understanding of the mathematical underpinnings of object oriented systems may lead to radically better approaches.

4. Dill and Rushby noted the importance of effective tools to support the use of formal methods in practice. Our tools for Z and Object-Z were limited to text editors and typesetters, and thus the benefits from those specifications came from the process of developing the specification. However, the ability to carry out automated analysis of LOTOS specifications, even for a limited part of the system, produced further insight. It was never a case of Object-Z **or** LOTOS; while we lacked tool support for Object-Z, its expressiveness for modelling state spaces is something that LOTOS was never intended to support.

Following on from the last point, a number of statements have been made in the paper about the difficulties encountered in using particular notations to describe aspects of PREMO. These should not be taken as criticisms of the language. There is a tension in specification language design between providing an expressive language while maintaining a simple underlying semantic model. Languages that attempt to handle all aspects of systems, for example concurrency, synchronization, real time, and error handling are likely to become difficult to understand and to use. We believe that progress in taming the intellectual complexity of systems like PREMO is likely to come, not from new and more complex specification logics, but from developing approaches that support the integration of partial specifications that capture particular aspects of a system in an appropriate representation. A basis for relating these representations may then be found by examining the underlying mathematical structures. The idea of formal specification, is after all, to utilise the simplicity, elegance and richness of mathematics to understand the behaviour of systems such as PREMO. Specification languages are simply one means to this end.

It remains for us to outline the open questions that we believe have been raised by the specification activities within PREMO.

1. How can suitable description techniques be identified and utilised in the development of complex systems? Although PREMO is not as complex a system as, for example, some of the reported industrial applications of formal methods, the breadth of the standard, spanning object model to media data, seems to challenge representation in a single formalism, at least if the result is to be produced and understood within the timescale and resources available to a standards committee. One question that may well be relevant is how the dependence of system components on the underlying software architecture can be modelled formally, particularly if the architecture itself is layered, as in PREMO where dataflow and constraint management is built on top of the object-oriented core.

2. What is the trade-off between producing mathematically elegant abstractions, versus models that capture detailed design decisions? It is usually assumed that a formal specification of a system is a prelude to design, yet a number of details in PREMO represent quite low-level decisions. One which is particularly noticeable is the operational details of the *Synchronizable* object type. This issue has implications for other standards and systems, for example VRML [48]. In developing a standard, particularly in an area where performance is a non-trivial concern, there may be implicit assumptions about the execution model that will be used to realise the system. In the case of PREMO for example, the semantics of the internal *stepping* mode involves a level of operational detail that normally one would associate more with a design or implementation. The problem is that a more abstract description of the intended behaviour would have been more difficult to relate to the standard and communicate to the committee; state machines are, after all, a well understood engineering concept, a fact that has been borne out by the experience of others in designing languages to document requirements and specifications [49].

3. Do we have a model of the application domain? In the case of PREMO, a number of different application domains could be identified, including dataflow processing and media content. One of the difficulties in answering this is that the adequacy of a model depends in part on the questions that one wishes to ask of the model. The approach to media content representation presented here seems useful in relating different kinds of presentation, but is, at least superficially, different from that assumed in [53] to represent quality of service requirements.

ACKNOWLEDGEMENTS

The work presented in this paper was carried out under the auspices of the ERCIM Computer Graphics Network, funded under the CEC Human Capital and Mobility Programme (contract CHRX-CT93-0085). The authors also thank Rebecca Lane for her help with the typesetting of this paper.

References

1. H. Saiedian, "An Invitation to Formal Methods," *IEEE Computer*, vol. 29, no. 4, pp. 16 – 17, 1996.
2. C.B. Jones, "A Rigorous Approach to Formal Methods," *IEEE Computer*, vol. 29, no. 4, pp. 20–21, 1996.
3. A. Hall, "What is the Formal Methods Debate About?," *IEEE Computer*, vol. 29, no. 4, pp. 22–23, 1996.
4. P. Zave, "Formal Methods are Research, not Development," *IEEE Computer*, vol. 29, no. 4, pp. 26–27, 1996.
5. D.L. Dill and J. Rushby, "Acceptance of Formal Methods, Lessons from Hardware Design," *IEEE Computer*, vol. 29, no. 4, pp. 23–24, 1996.
6. ISO/IEC 13522-1, International Organisation for Standardisation, *Information processing systems, Information technology, Coding of Multimedia and Hypermedia Information (MHEG)*, 1994.
7. T. Meyer-Boudnik and W. Effelsberg, "MHEG Explained," *IEEE MultiMedia*, pp. 26–38, 1995.
8. J.L. Mitchell, D. Le Gall, and C. Fogg, *MPEG Video Compression Standard*, Chapman and Hall, 1996.
9. ISO, http://www.iso.ch
10. G. J. Reynolds, D. A. Duce, and D. J. Duke, "Report of the ISO/IEC JTC1/SC24 Special Rapporteur Group on Formal Description Techniques," Tech. Rep. ISO/IEC JTC1/SC24 N1152, International Organization for Standardization, 1994, ftp://ftp.cwi.nl/pub/premo/Miscellaneous/sc24-fdt-report.ps
11. D.A. Duce and E.V.C. Fielding, "Towards a Formal Specification of the GKS Output Primitives," in *Eurographics '86*, A.A.G. Requicha, Ed. 1988, pp. 307–323, North Holland.
12. D.A. Duce, P.J.W. ten Hagen, and R. van Liere, "An Approach to Hierarchical Input Devices," *Computer Graphics Forum*, vol. 9, no. 1, pp. 15–26, 1990.
13. D.A. Duce and L.B. Damnjanovic, "Formal Specification in the Revision of GKS: An Illustrative Example," *Computer Graphics Forum*, vol. 11, no. 1, pp. 17–30, 1992.
14. D.B. Arnold, D.A. Duce, and G.J. Reynolds, "An Approach to the Formal Specification of Configurable Models of Graphics Systems," in *Proceedings of Eurographics '87*, G. Maréchal, Ed. 1987, North-Holland.
15. D. Soede, F. Arbab, I. Herman, and P. J. W. ten Hagen, "The GKS Input Model in MANIFOLD," *Computer Graphics Forum*, vol. 10, no. 3, pp. 209–224, 1991.
16. G. P. Faconti and F. Paternó, "An Approach to the Formal Specification of the Components of an Interaction," in *Proceedings of Eurographics '90*, C. E. Vandoni and D. A. Duce, Eds. 1990, North-Holland.

17. F. Paternó and G. F. Faconti, "On the Use of LOTOS to Describe Graphical Interaction," in *People and Computers VII: HCI'92 Conference*, D. Diaper A. Monk and M.D. Harrison, Eds. 1992, BCS HCI Specialist Group, Cambridge University Press.
18. D.J. Duke and M.D. Harrison, "Abstract Interaction Objects," *Computer Graphics Forum*, vol. 12, no. 3, pp. C–25 – C–36, 1993.
19. G. D. Abowd, *Formal Aspects of Human-Computer Interaction*, Ph.D. thesis, University of Oxford Computing Laboratory: Programming Research Group, 1991, Available as Technical Monograph PRG-97.
20. M.D. Harrison and A. Dix, "A state model of direct manipulation," in *Formal Methods in Human Computer Interaction*, M.D. Harrison and H.W. Thimbleby, Eds. 1990, pp. 129–151, Cambridge University Press.
21. B. Sufrin and J. He, "Specification, refinement, and analysis of interactive processes," in *Formal Methods in Human Computer Interaction*, M.D. Harrison and H.W. Thimbleby, Eds. 1990, pp. 153–200, Cambridge University Press.
22. M.D. Harrison and D.J. Duke, "A review of formalisms for describing interactive behaviour," in *Proc Intl. Workshop on Software Engineering and Human-Computer Interaction*. 1995, vol. 896 of *Lecture Notes in Computer Science*, pp. 49–75, Springer-Verlag.
23. P.W. Nehlig and D.A. Duce, "GKS-9X: The Design Output Primitive, an Approach to a Specification," *Computer Graphics Forum*, vol. 13, no. 3, pp. 381–392, 1994, Conference Issue: Proc. Eurographics'94.
24. G.J. Reynolds, *Configurable Graphics Systems: Modelling and Specification*, Ph.D. thesis, PhD thesis, Univeristy of East Anglia, Norwich, United Kingdom, September 1991.
25. H. Kilov and J. Ross, *Information modeling: an object-oriented approach*, Prentice-Hall, 1994.
26. M.J. Hinchey and J.P. Bowen, *Applications of Formal Methods*, Prentice Hall International, 1995.
27. J.M. Spivey, *The Z Notation: A Reference Manual*, Prentice Hall International, second edition, 1992.
28. R.W. Duke, P. King, G. Rose, and G. Smith, "The Object-Z specification language: Version 1," Tech. Rep. 91-1, Software Verification Research Centre, University of Queensland, 1991.
29. ISO/IS 8807, International Organisation for Standardisation, *Information processing systems - Open Systems Interconnection - LOTOS - A Formal Description Technique Based on Temporal Ordering of Observational Behaviour*, 1988.
30. ISO/IEC 13817-1, International Organisation for Standardisation, *Information technology - Programming languages, their environments and system software interfaces - Vienna Development Method - Specification Language - Part 1: Base language*, 1996.
31. I.S.C. Houston and M.B. Josephs, "A formal description of the OMG's Core Object Model and the meaning of compatible extension," *Computer Standards & Interfaces*, vol. 17, pp. 553–558, 1995.
32. I. Herman, G.J. Reynolds, and J. Davy, "MADE: A Multimedia Application Development Environment," in *Proceedings of the IEEE International Conference on Multimedia Computing Systems (ICMCS'94)*. 1994, IEEE CS Press.
33. D. A. Duce, D. J. Duke, P. J. W. ten Hagen, I. Herman, and G. J. Reynolds, "Formal methods in the development of PREMO," *Computer Standards & Interfaces*, vol. 17, no. 5-6, pp. 491 – 509, 1995, Special Issue on Formal Methods and Standards.
34. J. Gosling, B. Joy, and G. Steele, *The Java Language*, Addison-Wesley, 1997.
35. D. A. Duce, D. J. Duke, P. J. W. ten Hagen, and G. J. Reynolds, "PREMO - an initial approach to a formal definition," *Computer Graphics Forum*, vol. 13, no. 3, pp. C–393 – C–406, 1994.
36. D.J. Duke, D.A. Duce, I. Herman, and G. Faconti, "Specifying the PREMO synchronization objects," Tech. Rep. 02/97-R048, European Research Consortium for Informatics and Mathematics (ERCIM), 1997, ftp://ftp.inria.fr/associations/ERCIM/research_reports/pdf/0297R048.pdf
37. R. Milner, J. Parrow, and D. Walker, "Mobile logics for mobile processes," *Information and Computation*,

vol. 100, pp. 1–77, 1992.

38. C.B. Jones, "An object-based design method for concurrent programs," Tech. Rep. UMCS-92-12-1, Department of Computer Science, University of Manchester, 1992.
39. D.J. Duke, *Object-Oriented Formal Specification*, Ph.D. thesis, Department of Computer Science, University of Queensland, 1992, ftp://ftp.it.uq.edu.au/pub/Thesis/david_duke.ps.Z.
40. D.J. Duke and R.W. Duke, "Towards a semantics for Object-Z," in *VDM'90: VDM and Z!*, D. Bjørner, C.A.R. Hoare, and H. Langmaack, Eds. April 1990, vol. 428 of *Lecture Notes in Computer Science*, pp. 242–262, Springer-Verlag.
41. P. Zave and M. Jackson, "Conjunction as composition," *ACM Transactions on Software Engineering and Methodology*, vol. 2, no. 4, pp. 379–411, 1993.
42. E.M. Clarke and J.M. Wing, "Formal methods: State of the art and future directions," *ACM Computing Surveys*, vol. 28, no. 4, pp. 626–643, 1996.
43. S.J. Gibbs and D.C. Tsichritzis, *Multimedia Programming*, ACM Press/Addison-Wesley, 1995.
44. J.F. Koegel Buford, "Architecture and issues for distributed multimedia systems," in *Multimedia Systems*. ACM Press/Addison-Wesley, 1994.
45. I. Herman, G.J. Reynolds, and J. Van Loo, "PREMO: An emerging standard for multimedia. Part I: Overview and framework," *IEEE MultiMedia*, vol. 3, pp. 83–89, 1996.
46. 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," *Multimedia Systems*, 1997, To appear.
47. A. Freeman and D. Ince, *Active Java: Object-Oriented Programming for the World Wide Web*, Addison-Wesley, 1996.
48. ISO/IEC 14722, International Organization for Standardization, *Information processing systems - Computer graphics and image processing - Virtual Reality Modelling Language (VRML)*, 1996, Draft International Standard.
49. N.G. Leveson, M.P.E. Heimdahl, H. Hildreth, and J.D. Reese, "Requirements specification for process control systems," *IEEE Transactions on Software Engineering*, vol. 20, no. 9, pp. 684–707, 1994.
50. D.J. Duke, "Time and synchronisation in PREMO: A formal specification of the NNI proposal," Tech. Rep. OME-116, ISO/IEC JTC1 SC24/WG6, 1995, ftp://ftp.cwi.nl/pub/premo/RapporteurGroup/Miscellaneous/OME-116.ps.gz
51. M. Abadi and L. Lamport, "An old-fashioned recipe for real time," Tech. Rep. 91, DEC Systems Research Center, Oct. 1992, http://gatekeeper.dec.com/pub/DEC/SRC/research-reports/abstracts/src-rr-091.html.
52. L. Blair, G. Blair, H. Bowman, and A. Chetwynd, "Formal specification and verification of multimedia systems in distributed open processing," *Computer Standards & Interfaces*, vol. 17, pp. 413–436, 1995.
53. G. Blair, L. Blair, and J.B. Stefani, "A specification architecture for multimedia systems in Open Distributed Processing," *Computer Networks and ISDN Systems*, vol. 29, pp. 473–500, 1997.
54. T. Bolognesi and E. Brinksma, "Introduction to ISO specification language LOTOS," *Computer Networks and ISDN Systems*, vol. 14, no. 25–59, 1987.
55. P. van Eijk, "The Lotosphere Integrated Tool Envirnoment LITE," in *Proceedings of the 4th International Conference on Formal Description Techniques*. 1991, North-Holland.
56. R. De Nicola, A. Fantechi, S. Gnesi, and G. Ristori, "Verifying Hardware Components within JACK," in *Proceedings of CHARME'95*. 1995, vol. 987 of *Lecture Notes in Computer Science*, pp. 246 – 260, Springer-Verlag.
57. V. Roy and R. De Simone, "AUTO and Autograph," in *Computer Aided Verification*. 1991, vol. 531 of *Lecture Notes in Computer Science*, pp. 65 – 75, Springer-Verlag.

58. E. Madeleine and A. Pnueli, "A Verification tool for Distributed Systems using reduction of Finite Automata Networks," in *Formal Description Techniques II*, S.T. Vuong, Ed., 1990, pp. 61–66.

59. C.A. Vissers, G. Scollo, M. van Sinderen, and E. Brinksma, "Specification Styles in Distributed Systems Design and Verification," *Theoretical Computer Science*, vol. 89, 1991.

60. G.P. Faconti and M. Massink, "Using LOTOS for the evaluation of design options in the PREMO standard," in *Proceedings of the 2nd BCS-FACS Northern Formal Methods Workshop, Ilkley*. 1997, Electronic Workshops in Computing, Springer-Verlag, To appear.

61. A. Bouali, S. Gnesi, and S Larosa, "The Integration Project for the JACK Environment," *Bulletin of the EATCS*, vol. 54, no. 207–223, 1994.

62. R. Duke, G. Rose, and G. Smith, "Object-Z: A Specification Language Advocated for the Description of Standards," *Computer Standards and Interfaces*, vol. 17, no. 5-6, pp. 511–533, 1995, Special Issue on Formal Methods and Standards.

63. D. Brookshire Conner and A. van Dam, "Sharing between graphical objects using delegation," in *Object-Oriented Programming for Graphics*, V. de May C. Laffra, E.H. Blake and X. Pintado, Eds. 1995, Focus on Computer Graphics, Springer-Verlag.

64. A. Watters, G. van Rossum, and J.C. Ahlstrom, *Internet Programming in Python*, M&T Books, 1996.

65. M. Shaw and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall International, 1996.

66. R. Allan and D. Garlan, "A formal basis for architectural connection," *ACM Transactions on Software Engineering and Methodology*, July 1997.

67. D. Gelernter, "Generative communication in Linda," *ACM Transactions on Programming Languages and Systems*, vol. 7, no. 1, pp. 80–112, 1985.

68. F. Arhab, "The IWIM model for coordination of concurrent activities," in *Coordination Languages and Models*. 1996, vol. 1061 of *Lecture Notes in Computer Science*, pp. 34–56, Springer-Verlag.

69. P. Ciancarini, K.K. Jensen, and D. Yankelevich, "On the operational semantics of a coordination language," in *Object-Based Models and Languages for Concurrent Systems*, P. Ciancarini, O. Nierstrasz, and A. Yonezawa, Eds. 1995, vol. 924 of *Lecture Notes in Computer Science*, pp. 77–106, Springer-Verlag.

70. B. Mahoney and I. Hayes, "A case-study in timed refinement: A mine pump," *IEEE Transactions on Software Engineering*, vol. SE-18, no. 9, pp. 817–826, 1992.