Formal Methods in the Development of PREMO

D.A.Duce, D.J.Duke, P.J.W.ten Hagen, I. Herman, G.J. Reynolds

Computer Science/Department of Interactive Systems

CS-R9465 1994

Formal Methods in the Development of PREMO

D.A. Duce

Informatics Department, DRAL
Chilton, Didcot, Oxon OX11 0QX, United Kingdom
dad@inf.rl.ac.uk

D.J. Duke

Department of Computer Science, University of York
Heslington, York, YO1 5DD, United Kingdom
duke@minster.york.ac.uk

P.J.W. ten Hagen, I. Herman, G.J. Reynolds

Department of Interactive Systems, CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
{paulh,ivan,reynolds}@cwi.nl

Abstract

ISO/IEC JTC1/SC24 are developing a standard for the presentation of multimedia objects, called PREMO (Presentation Environments for Multimedia Objects). PREMO is a multipart standard, the most well-defined parts of which, at the time of writing, are at the stage of Committee Draft.

This paper describes how formal description techniques are being used in the development of the PREMO standard, shadowing the development of the standard itself. The approach taken uses a combination of Z and Object-Z. The motivation and merits of this approach are discussed, and illustrated with a description of some fundamental concepts of the PREMO object model.

AMS Subject Classification (1991): 68N15
CR Subject Classification (1991): D.1.5,D.2.1,D.3.1,F.3.2,H.5.1,I.m,K.1
Keywords & Phrases: PREMO, formal methods, multimedia, object models, active objects, Z, Object-Z.
Note: This paper has been offered for publication in the journal "Computer Standards and Interfaces".

1. Introduction

PREMO - Presentation Environments for Multimedia Objects - is the name of a standard under development within ISO/IEC JTC1/SC24, the ISO/IEC committee responsible for standardization in the area of computer graphics and image processing. The project was approved during 1994. PREMO is a multipart standard, which currently consists of four parts:

- Part 1: Fundamentals of PREMO
- Part 2: Foundation component
- Part 3: Modelling, rendering and interaction component
- Part 4: Multimedia systems services component

Further parts may be added in the future. At the time of writing (October 1994), Parts 1 and 2 had the status of Committee Draft and Part 3 is a Working Draft. Part 4 is at an earlier stage within ISO/IEC. An overview of PREMO is given in Herman et al [13].

Recognizing the importance of formal description techniques, SC24 appointed a Special Rapporteur for Formal Description Languages (G.J. Reynolds) in July 1993 and invited him to call an ad hoc meeting of experts and provide an initial report on the applicability of formal description techniques to SC24 standards, with particular regard to formally specifying object behaviour and interfaces, by May 1994. The authors of the report are included amongst the authors of this present paper. The report was duly delivered [23] and the following resolutions were passed at the SC24 Plenary in June 1994.

1. **Encouragement to use Formal Description Techniques:** ISO/IEC JTC1/SC24 actively endorses and encourages the use of formal description techniques during the development of SC24 standards;

2. **Endorsement to use Formal Description Languages in the Development of PREMO:** ISO/IEC JTC1/SC24 endorses the use of the formal description language, Object-Z, in the development of PREMO.

3. **Publication of Formal Descriptions of SC24 Standards**: ISO/IEC JTC1/SC24 encourages the publication of formal descriptions of SC24 standards according to the guidelines in Annex F of JTC1 Directives, Second Edition, 1992. This includes publication as an ISO Technical Report, if the Formal Description is not published with the standard.

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 [6, 7, 4, 2, 24, 10, 21, 8, 1, 11, 26, 12, 20, 22]. A book by Kilov and Ross of Bellcore[19] is an indication of the uptake of formal methods (in this case Object-Z) in industry; further examples are contained in a forthcoming book by Hinchey and Bowen [15].

The choice of Z [25] and Object-Z [9] for the work described here was motivated by three considerations.

1. PREMO is a state-based system and there is a natural affinity to a state-based formal description technique, such as Z. 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 has the status of International Standard is LOTOS [18]. LOTOS is a language based on process algebra, though it is loosely coupled to an algebraic specification language, ACTONE. The present PREMO work is concerned with the description of the PREMO object model, including intra and inter object communication. LOTOS was considered inappropriate for this work, at least initially, because it is not state-based and not object-oriented. However, some work is planned to look in more detail at the appropriateness of LOTOS for describing the communications aspects of PREMO, building on the object model definition described here.

3. There was expertise in the group in both Z and Object-Z, more so than in process based notations. The existence of appropriate expertise in formal description techniques is a significant hurdle to overcome in gaining acceptance for formalism. 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.

The next section illustrates how Z and Object-Z are being used in the development of PREMO in particular to describe the PREMO object model. The description is based on the description of PREMO in the Committee Drafts of Parts 1 and 2 of the standard [16, 17]. The version of Object-Z used in this paper is that described in the University of Queensland Technical Report [9].

2. The Premo Object Model

2.1 Overview

The Premo standard poses some particularly interesting issues for formal description in that the Premo definition follows an object-oriented style and is designed to be extensible in the tradition of object-oriented systems. The key concept in Premo in this respect is the object model. The object model describes the object type structure and subtyping relationships, references to objects, non-object types, operations on objects, subtyping and inheritance, operation dispatching, the semantics of servicing requests for operations to be performed on an object, the object life-cycle, and the event model.

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 would in principle be possible to use an object-oriented formal description technique for which the semantics of the Premo object model were included in the semantics of the formal description technique itself. In practice, such an approach would mean defining a new formal description technique.

Instead the approach being taken is to have two levels of specification. The first describes the object model itself. This specification is given in Z. The second describes the behaviour of Premo objects themselves and for this the Object-Z notation is used. Certain aspects of the Object-Z specification are interpreted with respect to the semantics of the Z object model specification; the link between the two levels of the specification defines how this is done. This situation is similar to the difficulty encountered in building 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 [14], 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 explicitly code some aspects of the Made object model in C++, rather than rely on mechanisms intrinsic to C++.

The Premo object model defines the semantics of object types and object interactions.

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

The object model makes a 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. An object reference that refers to no object has a distinguishable value.

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 request is held by the callee. Subsequent requests will overwrite any pending request.

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. Examples in PREMO are integer and real numbers. PREMO defines a hierarchy of non-object data types, containing both simple (basic) data types and constructed data types. This can be modelled by a free-type definition in Z, an extract of which follows:

$$\begin{array}{lll} \textit{non-obj} ::= & \textit{int\_value}\langle\langle \mathbb{Z} \rangle\rangle & \text{– Integers} \\ \mid & \textit{real\_value}\langle\langle \Re \rangle\rangle & \text{– Real numbers} \\ \mid & \textit{char\_value}\langle\langle \textit{Char} \rangle\rangle & \text{– Characters} \\ \mid & \textit{seq\_value}\langle\langle \mathrm{seq}\ \textit{non-obj} \rangle\rangle & \text{– Sequence} \\ \mid & \ldots & \end{array}$$

Further consideration of the description of non-object types is beyond the scope of this paper.

2.2 The PREMO *Object System*
This section presents an initial specification of the PREMO object system using a combination of Z and Object-Z. Since PREMO is very much an evolving standard this description is not intended to be definitive in any sense. Rather, the aim is to demonstrate the feasibility and utility of using an object-oriented formal description technique to obtain a precise description of the standard.

It is convenient to factor the formal account into three:

- The first is concerned with the PREMO object model and the way in which the collection of objects that make up a running PREMO system can interact with each other. This is developed in this section.
- The second focuses on the structure and behaviour of individual PREMO objects. Section 2.3 demonstrates the approach adopted for PREMO.
- PREMO defines the concept of a component as a way of organising a collection of objects that together implement certain services into a single abstraction. This aspect of PREMO is described in Section 4. Initial attempts to formalise the notion of component revealed the need for further clarification of the concepts.

The Object Model. As the PREMO standard makes a clear commitment to a particular view of object references and object types it is useful to capture these ideas explicitly. Two "given" sets are introduced. Each names a set of values that are of interest in the specification, but whose structure is unimportant. The state of an object is not an explicit part of this model. An object type defines the behaviour common to a set of objects. This behaviour includes changes to the state brought about by operations.

$[\textit{object}]$ - object identities
$[\textit{objtype}]$ - object types

An object reference is a value that reliably denotes a particular object. An object reference that refers to no specific object is given a distinguished null value, *NULLobject*.

$[objref]$ - object references: these are used to refer to objects

$NULLobject : objref$

Each object type has an interface, consisting of the set of operations that an object of that type can be requested to perform. It is important to distinguish between an operation as something that can be performed to achieve an effect, and an operation *name* as something that can be passed around as a value. The former concept is denoted by the type *operation* and the latter by *opname*.

$[operation]$ - operations effect state change
$[opname]$ - set of operation names

The structure of PREMO object types is captured by the schema given below. An object type introduces a collection of operation names and definitions which may build on the structure of other object types through inheritance. In order to define the meaning of inheritance within PREMO we need to distinguish between the definitions given explicitly within an object type, and the larger set of operations made available to the type through inheritance. This process is simplified by defining a *framing schema*, $\Phi interfaces$, to represent the structures common to the internal and external interface of an object type. The name of the schema is prefixed with the symbol 'Φ' to indicate that this is an auxiliary definition, and not part of the system being defined.

The *interfaces* of object types are described by two functions. One function gives the set of operation names. The second takes an object type to a mapping from operation names to operation definitions, that is, for each object type it assigns an implementation to each operation name defined for that type.

$\Phi interfaces$

$$names : objtype \leftrightarrow\!\!\!\!\!\!+ \;\mathbb{P}\, opname$$
$$defns : objtype \leftrightarrow\!\!\!\!\!\!+ \;(opname \leftrightarrow\!\!\!\!\!\!+ \; operation)$$

$$\forall\, t : objtype;\; o : opname \bullet names(t) = \operatorname{dom}(defns\ t)$$

An object type is defined by introducing the interface twice. The internal interface represents the operations defined explicitly by an object type. The external interface represents all of the operations available, both from the type and its supertypes (those types from which it inherits). Variables are tagged with either the subscript 'int' or 'ext' to indicate which interface they refer to. A new variable, *inherits-from*, is introduced to represent the type inheritance hierarchy.

One of the difficulties in defining the meaning of inheritance is accounting for the possibility of name clashes caused by multiple inheritance. The solution described in this paper is to map each operation o and object type t to a definition graph ($defgraph$) that represents the inheritance relationship between supertypes of t that define the operation. For each operation defined within t, there must be a unique 'minimum' supertype of t (possibly t itself) in which the operation is defined. The function that determines the minimal elements of a relation is generic, and is given by:

$[X]$

$$min : (X \leftrightarrow X) \rightarrow \mathbb{P}\, X$$

$$\begin{aligned}
&\forall\, R : X \leftrightarrow X \bullet \textbf{let}\; items == \operatorname{dom} R \cup \operatorname{ran} R \bullet\\
&\qquad \forall\, v : items \bullet v \in min(R)\\
&\qquad \Leftrightarrow\\
&\qquad \forall\, v' : items \bullet v' \neq v \Rightarrow (v, v') \in R
\end{aligned}$$

The structure of PREMO object types is then represented by the following schema:

$$\begin{array}{|l}
\textit{types} \\
\hline
\Phi \textit{interfaces}_{int} \\
\Phi \textit{interfaces}_{ext} \\
\_ \ \textit{inherits-from} \ \_ : \textit{objtype} \leftrightarrow \textit{objtype} \\
\textit{defgraph} : (\textit{opname} \times \textit{objtype}) \rightarrow (\textit{objtype} \leftrightarrow \textit{objtype}) \\
\hline
\forall o : \textit{opname};\ t : \textit{objtype} \bullet \qquad [1] \\
\quad \textit{defgraph}(o, b) = \{ s, t : \textit{objtype} \mid b \ \textit{inherits-from}^* \ s \wedge o \in \textit{names}_{int}(s) \\
\qquad\qquad \wedge\ o \in \textit{names}_{int}(t) \wedge s \ \textit{inherits-from}^* \ t \} \\
\\
\forall t : \textit{objtype};\ o : \textit{opname} \bullet \qquad [2] \\
\quad o \in \textit{name}_{ext}(t) \Leftrightarrow \left(\begin{array}{l} o \in \textit{names}_{int}(t) \\ \vee \\ \exists s : \textit{objtype} \bullet t \ \textit{inherits-from} \ s \wedge o \in \textit{names}_{ext}(s) \end{array} \right) \\
\\
\forall o : \textit{opname};\ t : \textit{objtype} \bullet o \in \textit{names}_{ext}(t) \Rightarrow \# \textit{min}(\textit{defgraph}(o, t)) = 1 \qquad [3] \\
\\
\forall t : \textit{objtype} \bullet \textit{defns}_{ext}(t) = \\
\quad \{ o : \textit{opname};\ s : \textit{objtype} \mid o \in \textit{names}_{ext}(t) \wedge \textit{min}(\textit{defgraph}(o, t)) = \{s\} \} \qquad [4] \\
\hline
\end{array}$$

The four key predicates have been numbered. [1] defines the structure of the definition graph for each operation. For each operation o, an object type s is related to another, t, provided that both define o and that s inherits from t. Intuitively, this means that the definition of o given in s is more specific than t, and should override it when the corresponding operation is invoked on an instance of s.

[2] states that an operation name is part of the external interface of a type if it is either defined by the type or is part of the external interface of some inherited type. Predicate [3] requires that for each operation available in the external interface of a type there is a unique 'closest' type that defines the operation. The purpose of this constraint is to ensure that any operation name in the external interface of an object type has a unique implementation. This is explicated by the fourth predicate, which states that the external definition of a type t maps each operation name o in the external interface of t to the definition of o that dominates. As consequence of this, if s defines an operation o to have implementation op_s, and t inherits from s but overrides the definition of o with op_t, then any client of t that requests o will invoke op_t.

PREMO also defines a subtype relationship between object types. In the current draft this is equated with the inheritance hierarchy. However, inheritance as implemented in object-oriented systems is not generally a reflexive relation (an object type does not inherit from itself), whereas the intuition underlying subtyping suggests that any object type is a subtype of itself. Consequently the specification has been weakened to require only that s is a subtype of t if s inherits from t.

$$\begin{array}{|l}
\textit{subtypes} \\
\hline
\textit{types} \\
\_ \ \textit{is-subtype-of} \ \_ : \textit{objtype} \leftrightarrow \textit{objtype} \\
\hline
\forall s, t, u : \textit{objtype} \bullet \\
\quad s \ \textit{inherits-from} \ t \Rightarrow s \ \textit{is-subtype-of} \ t \\
\quad s \ \textit{is-subtype-of} \ s \\
\quad s \ \textit{is-subtype-of} \ t \wedge t \ \textit{is-subtype-of} \ s \Rightarrow s = t \\
\quad s \ \textit{is-subtype-of} \ t \wedge t \ \textit{is-subtype-of} \ u \Rightarrow s \ \textit{is-subtype-of} \ u \\
\hline
\end{array}$$

Separating subtyping and inheritance in the formal definition has shown a need to clarify the conditions in PREMO which are sufficient to guarantee the existence of subtypes.

Object types act as templates for the construction of object instances within a Premo system. These instances are accessed through object references, which identify both the object instance and a 'reference type' which may differ from the 'immediate type' of the object (the type from which it was created). In general, an object's reference type can be any super type of the immediate type, a situation that is illustrated in Figure 0.1. In the figure, the object type scrollable-window has been created by inheriting from 'window' and 'scroll-bar'; the two operations taken from window have been overridden. An object of type scrollable-window has been created, and is accessible through two references, ref-A and ref-B. Sending the message 'move' to the object via ref-A will result in the operation *op-m*1 being invoked; sending the same message through ref-B will invoke *op-m*, the implementation of *move* defined in the supertype.

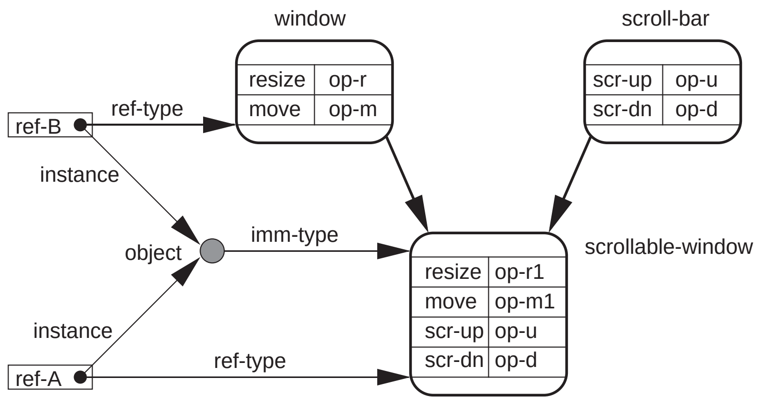

Figure 0.1: Object types, instances, and references.

The relationship between objects and types is captured formally by the schema given below. Names in the schema are consistent with Figure 0.1.

$$\begin{array}{|l}
\textit{objects} \\
\hline
\textit{subtypes} \\
\textit{instance} \; : \textit{objref} \nrightarrow \textit{object} \\
\textit{imm-type} : \textit{object} \nrightarrow \textit{objtype} \\
\textit{ref-type} \; : \textit{objref} \nrightarrow \textit{objtype} \\
\hline
\forall\, r : \textit{objref} \bullet \textit{imm-type}(\textit{instance}(r)) \;\textit{is-subtype-of}\; \textit{ref-type}(r) \\
\operatorname{ran} \textit{instance} = \operatorname{dom} \textit{imm-type} \\
\operatorname{dom} \textit{instance} = \operatorname{dom} \textit{ref-type} \\
\hline
\end{array}$$

It is unclear from the current draft what axioms, if any, can be asserted to hold over the object system. If the specification was extended to model the storage allocation of objects, then an obvious requirement would seem to be no 'dangling references', that is every reference either identifies some object in the store, or is null. Unfortunately this requirement may not be easy to enforce within some Premo implementations.

The object model may also support query operations, such as inquiring the immediate supertypes of a specific object type. The result is a sequence whose order is not defined. The foundation component must provide services used by Premo objects to inquire type structures, for example:

ImmediateST

$\Xi objects$
$t? : objtype$
$ts! : \mathrm{seq}\ objtype$

$\mathbf{let}\ tset == \{s : objtype \mid t?\ subtype\ s\} \bullet \#ts! = \#tset \land \mathrm{ran}\ ts! = tset$

The interface to this operation is contained in the description of PREMOObject in Section 3.1.

Object Life Cycle. The notion of *component* allows the PREMO standard to be structured in terms of the services provided. Underlying all PREMO components is a *foundation component* (which consists of a collection of foundation objects) which provides functionality necessary for all PREMO components. The foundation objects include the life-cycle manager object. This provides object life-cycle services which include the creation of new objects, destruction of objects and object references and management of object references. It is possible to have more than one life-cycle manager in a PREMO system and to distinguish between different life-cycle managers. A type is introduced to represent the identity of life-cycle managers.

$[LMid]$ - Identify given life cycle managers

A *lifecycle* schema introduces variables to represent the creator of each object reference (and hence implicitly of each object), and the pool of references that can be used by the manager. The predicate requires that the references allocated by each manager are distinct, and that every object instance has a creator.

lifecycle

$objects$
$creator : objref \rightarrowtail\!\!\!\!\!+ LMid$
$pool : LMid \rightarrowtail\!\!\!\!\!+ \mathbb{P}\, objref$

$\mathrm{dom}\ pool \subseteq \mathrm{ran}\ creator$
$\forall P, Q : LMid \bullet P \neq Q \Rightarrow pool(P) \cap pool(Q) = \varnothing$
$\mathrm{dom}\ creator = \mathrm{dom}\ instance$

Life cycle manager operations need to take an additional parameter, which is the life cycle manager identifier. Life cycle manager (LCM) objects can be accommodated by viewing an operation such as $create(lcm, type)$ as operation invocations within Object-Z, i.e. $lcm.create(objtype)$. The problem of connecting Z and Object-Z descriptions is discussed in Section 2.3.

create

$$
\begin{array}{|l}
\Delta \mathit{lifecycle} \qquad\qquad\qquad\qquad \text{[this operation changes some of the LCM state]} \\
\Xi \mathit{subtypes} \qquad\qquad\qquad\qquad \text{[but leaves the type structure unchanged]} \\
\mathit{lcm?} : \mathit{LMid} \\
\mathit{type?} : \mathit{objtype} \\
\mathit{ref!} : \mathit{objref} \\
\hline
\mathit{lcm?} \in \operatorname{dom} \mathit{pool} \\
\textbf{let } \mathit{lcm\text{-}pool} == \mathit{pool}(\mathit{lcm?}) \bullet \\
\quad \mathit{ref!} \in \mathit{lcm\text{-}pool} \\
\quad \mathit{pool}' = \mathit{pool} \oplus \{\mathit{lcm?} \mapsto \mathit{lcm\text{-}pool} \setminus \{\mathit{ref!}\}\} \\
\quad \mathit{creator}' = \mathit{creator} \oplus \{\mathit{ref!} \mapsto \mathit{lcm?}\} \\
\quad \exists\, \mathit{new} : \mathit{object} \bullet \\
\quad\quad \mathit{new} \notin \operatorname{ran} \mathit{instance} \\
\quad\quad \mathit{instance}' = \mathit{instance} \oplus \{\mathit{ref!} \mapsto \mathit{new}\} \\
\quad\quad \mathit{ref\text{-}type}' = \mathit{ref\text{-}type} \oplus \{\mathit{ref!} \mapsto \mathit{type?}\} \\
\quad\quad \mathit{imm\text{-}type}' = \mathit{imm\text{-}type} \oplus \{\mathit{new} \mapsto \mathit{type?}\} \\
\hline
\end{array}
$$

As a second example, the *cast* operation creates a new object reference to a given object, such that the type of the reference is some supertype of the object's immediate type. Note that the LCM again appears as a parameter, which is used in the precondition. Exceptions could be defined for trying to cast a reference to a non-supertype or asking the wrong LCM to operate on a reference.

cast

$$
\begin{array}{|l}
\Delta \mathit{lifecycle} \\
\Xi \mathit{subtypes} \\
\mathit{lcm?} : \mathit{LMid} \\
\mathit{old?} : \mathit{objref} \\
\mathit{req?} : \mathit{objtype} \\
\mathit{new!} : \mathit{objref} \\
\hline
\mathit{creator}(\mathit{old?}) = \mathit{lcm?} \\
\mathit{imm\text{-}type}(\mathit{instance}(\mathit{old?}))\ \mathit{is\text{-}subtype\text{-}of}\ \mathit{req?} \\
\textbf{let } \mathit{lcm\text{-}pool} == \mathit{pool}(\mathit{lcm?}) \bullet \\
\quad \mathit{new!} \in \mathit{lcm\text{-}pool} \\
\quad \mathit{pool}' = \mathit{pool} \oplus \{\mathit{lcm?} \mapsto \mathit{lcm\text{-}pool} \setminus \{\mathit{new!}\}\} \\
\quad \mathit{instance}' = \mathit{instance} \oplus \{\mathit{new!} \mapsto \mathit{instance}(\mathit{old?})\} \\
\quad \mathit{imm\text{-}type}' = \mathit{imm\text{-}type} \\
\quad \mathit{ref\text{-}type}' = \mathit{ref\text{-}type} \oplus \{\mathit{new!} \mapsto \mathit{req?}\} \\
\hline
\end{array}
$$

Operation Dispatching. Objects in PREMO communicate by sending messages which cause the receiver to perform a specified operation using given arguments. As a result, the state of the receiving object may change and the operation may return results to the original sender. Although this 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 and are at a level of detail beyond that usually captured by specification techniques. For PREMO, the issues of *what* behaviour is expected (i.e. the specification of the object model) partly encompasses issues of *how* that behaviour is to be provided (e.g. operational details of the object model).

Operation invocation is subject to three different semantics, depending upon the mode of the operation receptor. Three request modes are described in the standard, and these are represented as values in the following free type definition:

$$\begin{array}{lll} opmode ::= & async & \text{– Asynchronous: no suspension, no results} \\ \quad\;\; | & sync & \text{– Synchronous: caller suspended, results returned} \\ \quad\;\; | & sampled & \text{– Sampled: consecutive calls overwrite earlier requests} \end{array}$$

To assist in defining operation request semantics we introduce two type synonyms. The parameters to an operation are a sequence of non-object values. A request is a pair consisting of an operation and the object that will perform the operation.

$$\begin{array}{ll} params & == \text{seq}\, non\text{-}obj \\ request & == object \times operation \end{array}$$

We need to extend both the object and type model to accommodate information about objects, operations, and results. It is unclear whether this information should be added to the model as part of operation definitions, or earlier, as part of the definition of objects and types, as there is some overlap in discussing these points within the PREMO document. Here we follow the former approach.

$$\begin{array}{|l} \textit{dispatching} \\ \hline lifecycle \\ mode : operation \mathrel{+\!\!\!\rightarrow} opmode \\ result : operation \times object \times params \mathrel{+\!\!\!\rightarrow} non\text{-}obj \\ selectable : object \mathrel{+\!\!\!\rightarrow} \mathbb{P}\, operation \\ \hline \forall\, op : operation, obj : object, p : params \mid (op, obj, p) \in \text{dom}\, result \bullet \\ \quad mode(op) = sync \\ \quad \exists\, nm : opname \bullet nm \in names_{ext}(imm\text{-}type(obj)) \land \\ \quad\quad\quad op = defns_{ext}(imm\text{-}type(obj))(nm) \\ \forall\, obj : object \bullet \\ \quad selectable(obj) \\ \quad \subseteq \\ \quad \{\, nm : opname \mid nm \in names_{ext}(imm\text{-}type(obj)) \bullet defns_{ext}(obj)(nm) \,\} \\ \hline \end{array}$$

Selectable operations are those that a particular object is willing to perform at a point in time; how this set is set or changed is not within the scope of the PREMO standard, but rather is a property of the language(s) within which a particular system is instantiated. In order to model operation request semantics we need to consider the runtime environment of a PREMO system, and this is captured by the following schema definition:

$$\begin{array}{|l} \textit{runtime} \\ \hline dispatching \\ suspended : \mathbb{P}\, object \\ pending : request \rightarrow \text{bag}(params \times object) \\ \hline \forall\, o : object \bullet \forall\, p : operation \bullet \\ \quad mode(p) = sampled \Rightarrow count(pending(o, p)) \leq 1 \\ \hline \end{array}$$

An object that invokes a synchronous operation is suspended until the operation has been performed. Sending a request to an object has the effect of making the request pending; the called object and operation is associated with a bag (multi-set) of invocations for that operation. In the case of sampled mode operations, that bag can contain at most a single item.

The first part of operation performance is called *selection*, and involves adding an operation request to the bag of pending requests. If the invoked operation is in sampled mode, 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.

$$
\begin{array}{|l}
\textit{select} \\
\hline
\Delta \textit{runtime} \\
\Xi \textit{dispatching} \\
\textit{sender?} : \textit{objref} \\
\textit{receiver?} : \textit{objref} \\
\textit{mesg?} : \textit{opname} \\
\textit{args?} : \textit{params} \\
\hline
\textbf{let} \left[\begin{array}{ll}
\textit{caller} & == \textit{instance}(\textit{sender?}) \\
\textit{callee} & == \textit{instance}(\textit{receiver?}) \\
\textit{ans} & == (\textit{args?}, \textit{caller}) \\
\textit{opn} & == \textit{defns}_{\textit{ext}}(\textit{ref-type}(\textit{receiver?}))(\textit{mesg?}) \\
\textit{call} & == (\textit{callee}, \textit{opn}) \\
\textit{held} & == \textit{pending}(\textit{call})
\end{array} \right] \bullet \\
\quad \textit{caller} \notin \textit{suspended} \\
\quad \textit{mode}(\textit{opn}) \neq \textit{sampled} \Rightarrow \textit{pending}' = \textit{pending} \oplus \{\textit{call} \mapsto \textit{held} \uplus [\![\textit{ans}]\!]\} \\
\quad \textit{mode}(\textit{opn}) = \textit{sampled} \Rightarrow \textit{pending}' = \textit{pending} \oplus \{\textit{call} \mapsto [\![\textit{ans}]\!]\} \\
\quad \textit{mode}(\textit{opn}) = \textit{sync} \Rightarrow \textit{suspended}' = \textit{suspended} \cup \{\textit{caller}\} \\
\quad \textit{mode}(\textit{opn}) \neq \textit{sync} \Rightarrow \textit{suspended}' = \textit{suspended} \\
\hline
\end{array}
$$

The second stage is evaluation of a pending request. A variable (r') is used to choose some request for which there is a non-empty bag of calls. Since we are not fully modelling object state we cannot explicitly describe the effect of evaluating an operation, but instead define an 'interface' to Object-Z in the form of the appropriate method invocation.

$$
\begin{array}{|l}
\textit{evaluate} \\
\hline
\Delta \textit{runtime} \\
\Xi \textit{subtypes} \\
r' : \textit{request} \\
\textit{res!} : \textit{non-obj} \\
\hline
\textit{count}(\textit{pending}(r)) > 0 \\
\textbf{let } r' == (\textit{opn}, \textit{callee}) \bullet \exists \textit{ps} : \textit{params};\ \textit{caller} : \textit{object} \bullet \\
\quad (\textit{args}, \textit{caller}) \sqsubseteq \textit{pending}(r') \bullet \\
\quad \textit{callee} \notin \textit{suspended} \\
\quad \textit{opn} \in \textit{selectable}(\textit{ob}) \\
\quad \textit{res!} = \textit{result}(\textit{opn}, \textit{callee}, \textit{args}) \\
\hline
\end{array}
$$

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

$\Delta runtime$
$\Xi dispatching$
$r : request$

$count(pending(r)) > 0$
$\mathbf{let}\ r == (opn, callee) \bullet \mathbf{let}\ pending(r) == (args, caller) \bullet$
$\quad pending' = pending \oplus \{r \mapsto pending(r) \uplus [\![(args, caller)]\!]\}$
$\quad mode(op) = sync \Rightarrow suspended' = suspended \setminus \{caller\}$
$\quad mode(op) \neq sync \Rightarrow suspended' = suspended$

A complete picture of operation invocation can be obtained by using the Z schema calculus to combine the three stages by forward composition. The semantics of $S \mathbin{⨾} T$ is that the final state achieved by S becomes the initial state acted on by T (just as in ordinary composition of program statements via ';').

$$perform \mathrel{\widehat{=}} select \mathbin{⨾} evaluate \mathbin{⨾} return$$

Intermediate states are implicitly hidden, so the variables r' and r introduced to pass the selected request from performance to return are not visible outside the semantics of invocation. The intention is that *perform* captures the semantics of operation invocation within Object-Z type definitions. Informally, the Object-Z expression $obj.opn(args)$ should be understood as applying the *perform* operation to the state of the object model.

3. Foundation Objects

3.1 PREMO *Object*

The previous section has developed a formal model of the PREMO object system, covering Part 1 of the standard. It is now possible to consider the behaviour of the PREMO foundation object types defined in Part 2. PREMO defines a collection of foundation object types which can be formed into a type hierarchy. PREMOObject is the root of this hierarchy. The operations on the PREMOObject type fall into two categories.

- Operations which control the creation and destruction of objects. These operations are used by life cycle management objects to create and destroy object instances.
- Operations which return information on the object type structure and where the object type fits into the type hierarchy.

The specification of PREMOObject is given by an Object-Z class definition, part of which is illustrated below.

PREMOObject

...

INIT

...

inquireImmediateSupertypes

$\Omega ImmediateST$

$t? = \mathit{ref\text{-}type}(\text{this})$

...

The *inquireImmediateSupertypes* operation makes use of the type information in the object model. The effect of this operation is defined in terms of the *immediateST* operation in the object model; see Section 2.2. We adopt a convention that references to object model structures are prefixed with an 'Ω' symbol.

An Object-Z *class* consists of an outer box within which appear the definitions of a state (in an unnamed box), a schema INIT describing valid initial states, and a collection of operations that can modify the state. Each operation has a *Delta*-list that mentions those variables changed by the operation; all other variables are unchanged, so for example moving a mouse does not affect the position of its button. A class can be defined as an extension to another using inheritance; the base class is named at the start of the definition, and each state or operation of the base is then part of the new class. Additional variables or invariants can be added to inherited structures.

3.2 Events and Event Handling

PREMO events provide a general mechanism for synchronisation between separate object instances in a running PREMO system. Within this paper we concentrate on the object type that provides general services for handling events. This will be used by types defined in subsequent components, for example Multi-Media Services, to implement synchronisation between separate elements in a running system.

An earlier paper [5] developed an alternative approach to event handling. The approach described here treats event handling in terms of an EventHandler object type as defined in Part 2 of the PREMO standard. A set of event names is introduced,

$$[event\text{-}name]$$

and then used within an Object-Z class definition. The handling of events draws on facilities specified within the object model.

EventHandler

PREMOObject

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

INIT

$register = \varnothing$

register

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

$\mathbf{let}\ T == \Omega ref\text{-}type(obj?) \bullet$
$\quad opn? \in \Omega names_{ext}(T)$
$\quad \Omega mode(defns_{ext}(T)) \neq sync$
$register' = register \cup \{e? \mapsto (opn?, obj?)\}$

unregister

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

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

$$
\begin{array}{|l}
\textit{send} \\
\hline
e? : \textit{event-name} \\
\hline
\forall\, obj : objref;\ opn : opname \bullet \\
\quad e? \mapsto (opn, obj) \in register \\
\quad \Rightarrow \\
\quad [\ \Omega perform \mid sender? = \mathrm{this} \wedge receiver? = obj \wedge mesg? = opn \wedge args? = \langle e? \rangle\ ] \\
\hline
\end{array}
$$

Of the operations defined by the event manager, both *register* and *unregister* are straightforward, though they involve some checking of operation names and modes. Further development of this point is beyond the scope of this paper. The third 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 *perform* operation holds with the input parameters bound to appropriate values.

4. Components

Objects and their associated operations provide basic units of functional behaviour on which a PREMO application can draw within its implementation. In general an application will be interested in more than just the isolated behaviour obtained from a single operation. Instead, the PREMO standard envisages that a collection of objects whose operations together provide various services can be packaged into larger structures call *components*.

This section gives a flavour of an approach to an abstract description of components and in particular the ways in which one component can depend upon another. This functionality is still at an early stage of development within PREMO. Three new types are introduced;

$[compnm]$	- the set of component names
$[servnm]$	- the set of services (names)

$$deptype ::= subtype \mid service$$

the two given sets are used to name components and the services that can be provided, while *deptype* serves to introduce the two types of dependencies that can exist.

The structure of a component is quite simple: it has a name, and it defines a set of types which it uses to implement a set of services. Finally, it may depend on the existence of other components, either because it inherits from another type (subtype dependency) or because it uses services provided by the other. Indeed, these possibilities are not necessarily exclusive.

$$
\begin{array}{|l}
\textit{Component} \\
\hline
name : compnm \\
defines : \mathbb{F}\, objtype \\
provides : \mathbb{F}\, servnm \\
uses : servnm \mathrel{+\!\!\!\!\twoheadrightarrow} \mathbb{F}\, opname \\
depends\_on : compnm \leftrightarrow deptype \\
\hline
name \notin \mathrm{dom}\ depends\_on \\
\hline
\end{array}
$$

The only formal axiom is that a component does not depend on itself. A component A in PREMO may depend on another component B in two ways:

1. there are objects in A whose types inherit from the object types defined in B.

2. there are objects in A whose behaviour depends on the services offered by B.

The consistency of a collection of components is expressed by the following schema.

$$
\begin{array}{|ll}
\multicolumn{2}{l}{\mathit{Components}} \\
\mathit{objects} & \text{[make use of the object model structure]} \\
\mathit{units} : \mathit{compnm} \pfun \mathit{Component} & \\
\hline
\forall\, \mathit{comp} : \operatorname{ran} \mathit{units} \bullet \operatorname{dom} \mathit{comp.depends\_on} \subseteq \operatorname{dom} \mathit{units} & [1] \\
\forall\, c : \operatorname{dom} \mathit{units} \bullet \forall\, d : \mathit{compnm};\ t : \mathit{deptype} \mid (d, t) \in \mathit{c.depends\_on} \bullet & [2] \\
\quad t = \mathit{subtype} \Rightarrow \exists\, ot_c : \mathit{c.defines};\ ot_d : \mathit{d.defines} \bullet & [2a] \\
\qquad\qquad\qquad ot_c\ \mathit{inherits\text{-}from}\ ot_d & \\
\quad t = \mathit{service} \Rightarrow \exists\, \mathit{sv} : \mathit{c.provides} \bullet & [2b] \\
\qquad \mathit{uses}(\mathit{sv}) \not\subseteq \bigcup \{\, \mathit{ot} : \mathit{c.defines} \bullet \mathit{names}_{\mathit{ext}}(\mathit{ot}) \,\} & \\
\qquad \mathit{uses}(\mathit{sv}) \subseteq \bigcup \{\, \mathit{ot} : \mathit{d.defines} \bullet \mathit{names}_{\mathit{ext}}(\mathit{ot}) \,\} & \\
\hline
\end{array}
$$

Predicate [1] says that the collection of components is closed with respect to the dependency relations within each component.

Predicate [2] expresses consistency with respect to conditions (1) and (2) above. Predicate [2a] expresses consistency with respect to inheritance and predicate [2b] with respect to services.

5. Multimedia Objects

In Premo emphasis is placed on the ability of objects to be active. This stems from the need to have development environments where different media may be presented in an integrated way, and which allow for the various medium-specific presentation techniques to coexist within the same system. Conceptually, different media (e.g. a video sequence and a corresponding sound track) may be considered as parallel activities that have to reach specific milestones at distinct synchronization points.

An approach to the specification of multimedia objects has been described in a paper by Duce et al.[5] at the Eurographics '94 conference. It is felt that the ideas in this paper will provide a good starting point for a formal description of Part 3 of Premo as this becomes more concrete.

6. Conclusions

This paper set out to demonstrate the approaches being taken to the formal description of the emerging Premo standard. The formal description work is undertaken by a subset of the experts engaged in the Premo work and aims to track the development of the standard. The major benefit obtained at this stage is insight into the emerging design. The development of the specification is a very useful forcing function for clarifying the fundamental concepts of Premo and their interrelationships. The specification exercise reported here on the Committee Drafts of Parts 1 and 2 has revealed a number of ambiguities, conflicts and issues requiring resolution, which are being fed into the next stage of processing.

The realization that a two-level specification would be required in order to separate the object model and runtime behaviour from object behaviour is interesting. At the present time there is no object model which has the status of ISO/IEC International Standard, and as object technology is still an evolving subject, there is good reason to accept that different application domains will evolve different object models until common understanding of requirements is attained. In consequence, it is unlikely that the object model in a new area such as multimedia systems, will directly match the object models in existing object-oriented specification languages. Since object models are primarily concerned with implementation issues rather than program semantics, it is unlikely that this position will change.

A single notation can be used to describe both the object model and higher level structures, if both are encoded in the same way. An example of this approach is contained in Sufrin and He's paper [26].

The difficulty with this approach is that key ideas are encoded into and obscured by the formalism. The main benefit for formal methods in standards is most likely to be as a way of identifying and defining precisely what the key concepts are and how they interrelate, rather than as a vehicle for proving assertions about a standard.

There may be ways to make specification languages more flexible in terms of capturing both object and meta-level concepts in the one model. One of us [DJD] has done some exploratory work with order-sorted algebras which shows promise, but at the present time such notations are less familiar than model-based specification languages and there is still much work to be done to demonstrate the hypothesis.

A problem with the two-level approach is that the connection between the object and runtime model is necessarily informal. The best that we can do is to link equivalent parts of the two models using common naming conventions. Such linkage between models is not necessarily bad – for example Barwise [3] notes that multiple models are routinely used in engineering. However, one of the benefits of formal description techniques is that they support formal reasoning about specifications, and this cannot be carried out where informal conventions are used to bind together a heterogeneous model; the best that can be done is to reason formally within each model and use rigorous argument to reason between models.

Acknowledgements
The authors of this paper have been brought together under the auspices of the ISO/IEC JTC1/SC24 Study Group on Formal Description Techniques. This has been enabled through participation in the ERCIM Computer Graphics Network funded under the CEC Human Capital and Mobility Programme.

References

1. G. D. Abowd. *Formal Aspects of Human-Computer Interaction.* PhD thesis, University of Oxford Computing Laboratory: Programming Research Group, 1991. Available as Technical Monograph PRG-97.
2. D. B. Arnold, D. A. Duce, and G. J. Reynolds. An Approach to the Formal Specification of Configurable Models of Graphics Systems. In G. Maréchal, editor, *Proceedings of Eurographics '87.* North-Holland, 1987.
3. J. Barwise. Heterogenous reasoning. In *Conceptual Graphs for Knowledge Representation: First Intl. Conf. on Conceptual Structures, number 699 in Lecture Notes in Computer Science.* Springer-Verlag, 1993.
4. D. A. Duce and L. B. Damnjanovic. Formal Specification in the Revision of GKS: An Illustrative Example. *Computer Graphics Forum*, 11(1):17 – 30, 1992.
5. 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*, 13(3):C–393 – C–406, 1994.
6. D. A. Duce and E. V. C. Fielding. Towards a Formal Specification of the GKS Output Primitives. In A.A.G. Requicha, editor, *Proceedings of Eurographics '86.* North-Holland, 1986.
7. D. A. Duce, R. van Liere, and P. J. W. ten Hagen. An Approach to Hierarchical Input Devices. *Computer Graphics Forum*, 1990.
8. D. J. Duke and M. D. Harrison. Abstract Interaction Objects. *Computer Graphics Forum*, 12(3):C–25 – C–36313, 1993.
9. R. Duke, P. King, G. Rose, and Smith G. The Object-Z Specification Language Version 1. Technical Report 91-1, Software Verification Research Centre, University of Queensland, Australia, 1991.
10. G. P. Faconti and F. Paternó. An Approach to the Formal Specification of the Components of an Interaction. In C. E. Vandoni and D. A. Duce, editors, *Proceedings of Eurographics '90.*

North-Holland, 1990.

11. M. D. Harrison and A. Dix. A state model of direct manipulation. In M. D. Harrison and H. W. Thimbleby, editors, *Formal Methods in Human Computer Interaction*, pages 129 – 151. Cambridge University Press, 1990.

12. M. D. Harrison and D. J. Duke. A review of formalisms for describing interactive behaviour. Technical report, Department of Computer Science, University of York, 1994. (To be presented at the workshop on Research Issues in the Intersection of Software Engineering and Human-Computer Interaction in conjunction with ICSE'94, Sorrento.).

13. I. Herman et al. PREMO: A ISO Standard for a Presentation Environment for Multimedia Objects. In D. Ferrari, editor, *Proceedings of the Second ACM International Conference on Multimedia*. ACM Press, 1994.

14. I. Herman, G.J. Reynolds, and J. Davy. MADE: A Multimedia Application Development Environment. In L.A. Belady, S.M. Stevens, and R. Steinmetz, editors, *Proceedings of the IEEE International Conference on Multimedia Computing Systems (ICMCS'94)*. IEEE CS Press, 1994.

15. M. Hinchey and J. Bowen. *Applications of Formal Methods*. Prentice Hall International Series in Computer Science, 1995. (to appear).

16. International Organisation for Standardisation. *Information processing systems - Computer graphics and image processing - Presentation Environments for Multimedia Objects (PREMO - Part 1: Fundamentals of PREMO. Document ISO/IEC JTC1/SC24 N1190*, 1994.

17. International Organisation for Standardisation. *Information processing systems - Computer graphics and image processing - Presentation Environments for Multimedia Objects (PREMO - Part 2: Foundation component. Document ISO/IEC JTC1/SC24 N1191*, 1994.

18. 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.

19. Haim Kilov and James Ross. *Information modeling: an object-oriented approach*. Prentice-Hall, 1994.

20. P. Nehlig and D. A. Duce. GKS-9x: The Design Output Primitive an Approach to a Specification. *Computer Graphics Forum*, 13(3):C–381 – C–392, 1994.

21. F. Paternó and G. F. Faconti. On the Use of LOTOS to Describe Graphical Interaction. In D. Diaper A. Monk and M.D. Harrison, editors, *People and Computers VII: HCI'92 Conference*. BCS HCI Speicalist Group, Cambridge University Press, 1992.

22. G.J. Reynolds. *Configurable Graphics Systems: Modelling and Specification*. PhD thesis, School of Information Systems, University of East Anglia, September 1991.

23. G.J. Reynolds, D.A. Duce, and D.J. Duke. *Report of the ISO/IEC JTC1/SC24 Special Rapporteur Group on Formal Description Techniques. Document ISO/IEC JTC1/SC24 N1152*. International Organisation for Standardisation, 1994.

24. D. Soede et al. The GKS Input Model in Manifold. *Computer Graphics Forum*, 10(3):209 – 224, 1991.

25. J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall, second edition, 1992.

26. B. Sufrin and J. He. Specification, refinement and analysis of interactive processes. In M. D. Harrison and H. W. Thimbleby, editors, *Formal Methods in Human Computer Interaction*, pages 153 – 200. Cambridge University Press, 1990.