Specifying the PREMO Synchronization Objects

D.J. Duke[1]**, D.A. Duce**[2]**, I. Herman**[3]**, G. Faconti**[4]

[1]Dept of Computer Science, University of York, Heslington, York, YO1 5DD, UK

[2]Rutherford Appleton Laboratory, Chilton, Didcot, OX11 0QX, UK

[3]Centrum voor Wiskunde en Informatica (CWI), Kruislaan 413, 1098 SJ Amsterdam, The Netherlands

[4]CNR - Instituto CNUCE, Via S.Maria 36, 56126 Pisa, Italy

Abstract

This paper describes the formal specification of object types for managing inter-media synchronisation and control within PREMO, an emerging ISO/IEC standard for multimedia systems. Object-Z, an object-oriented extension to the Z specification language, is used for this purpose. Some aspects of PREMO are non-trivial to express in Object-Z, and the paper outlines the reasons for choosing this specific language, and sets out recommendations for further research in the use of formal languages in this area. The work reported here has been carried out by members of the ISO/SC24 committee involved in producing the PREMO standard, and has been informed by a number of workshops sponsored by the ERCIM Computer Graphics Network.

1 Introduction

The authors of this paper have been engaged for some years now in the development of an ISO/IEC standard for multimedia programming called PREMO (PResentation Environments for Multimedia Objects). In 1993 a Special Rapporteur Group on Formal Description Techniques was established in ISO/IEC SC24 to consider the role of formal description techniques in SC24 standards in general and in PREMO in particular. The group reported in May 1994 [17]. Their recommendations included a recommendation that one of the main uses of formal description should be during the development stages of a standard and that PREMO was a prime candidate for this. Some preliminary specification work accompanied the report. The recommendations were accepted by SC24 and since that time three of the authors (Duce, Duke and Herman) have been engaged with an international group of experts in the development of PREMO and have been largely responsible (along with G.J. Reynolds, now at CSIRO, Australia) for the specification work that has been undertaken. The work has focused on three main areas: the PREMO object model, the specification of inter-media synchronization and 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).

This paper reflects on the approach we have taken to specifying the PREMO objects which deal with inter-media synchronization and the difficulties encountered. The specification itself is not included in this paper: for a more detailed presentation of the specification on which this paper is based, see the ERCIM Technical Report [6].

Our aim in undertaking this work was to capture the key features of the inter-media synchronization functionality as it was being developed. At each stage in the formal ISO development process, nations participating in the project are invited to comment and vote on the functionality presented. The use of formal description techniques to model and track the development of the emerging functionality proved an invaluable adjunct to accumulated experience and craft skills in identifying deficiencies and errors in the standards documents themselves. Though this is not a new insight, our experience adds weight to the view that formal description has an important role to play in the development of ISO/IEC standards. This is particularly true in the case of a standard such as PREMO for which the resources to develop a prototype or reference implementation in parallel with the development of the design did not exist. The formal description as a mathematical model of the design can be exercised in various ways to build up confidence that the behaviour of the design as presented in the standards document does indeed accord with the intentions of the designers.

The formal notation used is Object-Z [2, 7, 8], an object-oriented extension to the Z notation [18]. Use of Object-Z to specify PREMO was one of the recommendations of a Special Rapporteur's Report on Formal Description Techniques in PREMO [17] and the specification of aspects of PREMO using the Z and Object-Z notations have appeared as [5] and [4]. An overview of Z and Object-Z is beyond the scope of this paper, and the reader is directed to [11, 18] (Z) and [2, 7, 8] (Object-Z) for details. One of us (G. Faconti) has explored the use of LOTOS and LOTOS tools in this endeavour in work reported elsewhere in this workshop.

The next section introduces the PREMO system and the problem we are studying.

1.1 A short overview of PREMO

This section gives a very short overview of PREMO; for a more detailed presentation the interested reader should consult [12] or [13].

Today's application developers needing to realize high-level multimedia applications which go beyond the level of multimedia authoring do not have an easy task. There are only a few programming tools that allow an application developer the freedom to create multimedia effects based on a more general model than multimedia document paradigms, and these tools are usually platform specific. In any case, there is currently no available ISO/IEC standard encompassing these requirements. A standard in this area should focus primarily on the presentation aspects of multimedia, and much less on the coding, transfer, or hypermedia document aspects, which are covered by a number of other ISO/IEC or de-facto standards (for example, MHEG[14]). It should also concentrate on the programming tool side, and less on, e.g., the (multimedia) document format side. These are exactly the main concerns of PREMO.

It is quite natural that the initiative for a standardization activity aiming at such a specification came from the group which has traditionally concentrated on presentation aspects over the past 15 years, namely ISO/IEC JTC1/SC24 (Computer Graphics). Indeed, this is the ISO subcommittee whose charter has been the development of computer graphics and image processing standards in the past. The Graphical Kernel System was the first standard for computer graphics published in this area; it was followed by a series of complementary standards, addressing different areas of computer graphics and image processing. Perhaps the best known of the application program interface (API) standards are PHIGS, PHIGS PLUS, and IPS (see, e.g., Arnold and Duce[1] for an overview of all these Standards). The subcommittee has now turned its attention to presentation media in general as a way of augmenting traditional graphics applications with continuous media such as audio, video, or still image facilities, in an integrated manner. The need for a new generation of standards for computer graphics emerged in the past 4-5 years to answer the challenges raised by new graphics techniques and programming environments and it is extremely fortunate that the review process to develop this new generation of presentation environments coincided with the emergence of multimedia. In consequence, a synergistic effect can be capitalized on.

The JTC1 SC24 subcommittee recognised the need to develop such a new line of standards. It also recognised that any new presentation environment should include more general multimedia effects to encompass the needs of various application areas. To this end, the PREMO project was started in ISO/IEC JTC1 SC24 WG6. PREMO reached the status of Draft International Standard in January 1997.

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

- PREMO is a Presentation Environment. PREMO, as well as the SC24 standards cited above, aims at providing a standard "programming" environment in a very general sense. The aim is to offer a 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.

- PREMO is aimed at a Multimedia presentation, whereas earlier SC24 standards concentrated either on synthetic graphics or image processing systems. Multimedia is considered here in a very general sense; high-level virtual reality environments, which mix real-time 3D rendering techniques with sound, video, or even tactile feedback, and their effects, are, for example, within the scope of PREMO.

- PREMO is Object Oriented. This means that, through standard object-oriented techniques, a PREMO implementation becomes extensible and configurable. Object-oriented technology also provides a framework to describe distribution in a consistent manner.

A precise object model constitutes a major part of PREMO. The object model is fairly traditional, and is based on the concepts of subtyping and inheritance. It is also very pragmatic in the sense that it includes, for efficiency reasons, the notion of non-object (data) types, as is the case with a number of object-oriented languages, such as C++ or Java, and in contrast to 'pure' object-oriented models, such as SmallTalk. The PREMO object model originates from the object model developed by the OMG consortium for distributed objects, but some aspects of the OMG model have been adapted to the needs of PREMO. A strong emphasis is placed in the model on the ability of objects to be active. That is, PREMO uses objects that have, conceptually, their own thread of control. However, if every object contains a thread of control, objects become 'heavyweight' and unsuitable for use as record-like structures. Having to develop a separate (non-object) data type for structures would be unfortunate, since treating structures as objects bring benefits in the form of inheritance, subtyping etc. The PREMO object model reconciles the tension between the desire for active objects on the one hand, and efficient structure objects on the other, by splitting the object type hierarchy into separate branches for 'simple' objects that can be used efficiently as structures, and 'enhanced' objects that have their own thread of control. The top level of the PREMO object type hierarchy is shown in Figure 1.

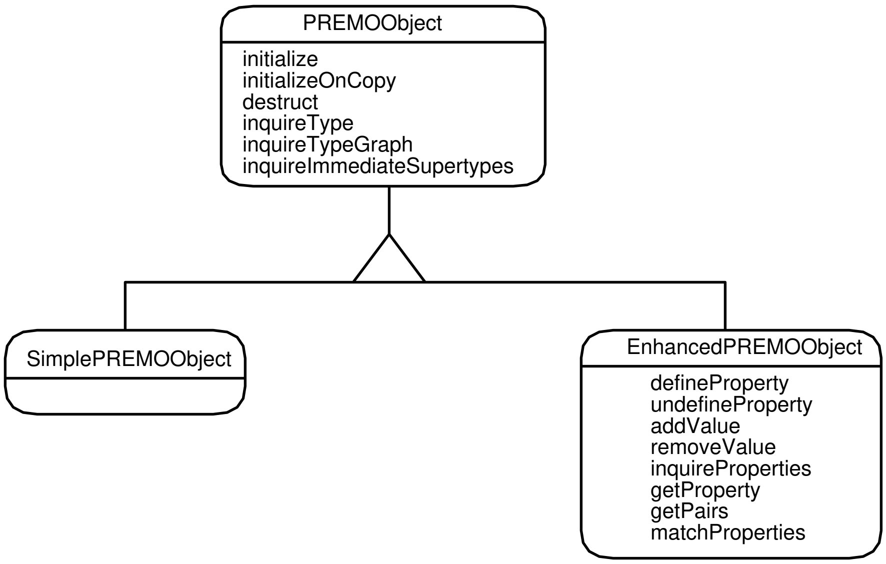

Figure 1: The top of the PREMO Object Type Hierarchy.

Enhanced PREMO objects can communicate with one another through messages, i.e., through the operations defined on the object types. Objects can become suspended either by waiting for an operation invocation to return, or by waiting on the arrival of an operation request. Consequently, operations on objects serve as a vehicle to synchronize various activities (note that this concept of object synchronization is not the same as media synchronization although, of course, the concepts are related). Whether the concurrent activity of active objects is realized through separate hardware processors, through distribution over a network, or through some multithreaded operating system service, is oblivious to PREMO and is considered to be an implementation choice.

The emphasis on the activity of objects stems primarily from the need for synchronization in multimedia environments and forms the basis of the synchronization model described in this paper. Using concurrency to achieve synchronization in multimedia systems is not specific to PREMO. Other models and systems have taken a similar approach (see, for example [3], and PREMO, whose task is to provide a synthesis for standardization, has obviously been influenced by these approaches.

2 An Approach to Specifying Synchronizable Objects

2.1 Introduction

As described above, 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. This is the event-based synchronization approach, which forms the basic layer of synchronization in PREMO. 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 [13].

In line with the object-oriented approach of PREMO, the synchronization model defines abstract object types that capture the essential features of synchronization. For the event-based synchronization scheme two major object types are defined:

- synchronizable objects, which form the supertypes of, e.g., various media object types;
- synchronization points, which may be used to manage complex synchronization patterns among synchronizable objects.

These objects and the approaches we have taken to their specification are described in somewhat more detail below.

2.2 Progression Space

Synchronizable objects in PREMO are autonomous objects, which have an internal progression along an internal one dimensional coordinate space. This space can be:

- 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" to the real and integer numbers, respectively. 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 unbounded. (The symbol "C" is used in this section to denote either an extended real, integer or time.) 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. Subtypes of synchronizable objects may add a semantic meaning to this coordinate space. For example, media objects may represent time, or video frame numbers along this space. Attributes that define the extent of the progression space can be set through operations defined on these objects.

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. The trade-off here would be the loss of direct contact between the definition of the standard and its specification.

2.3 Reference Points and Synchronization Elements

A synchronizable object is a finite state machine that controls the position and progress through an ordered collection of coordinates, some of which contain *synchronization elements* that can be used to organise the behaviour within a system based on such objects. The intention is that object types representing different kinds of media (video, sound etc) will inherit from this class and specialise the coordinate system and state machine in an appropriate way. For example, a video might use frame numbers as coordinates.

The positions within a synchronizable object at which some action needs to take place are marked by synchronization elements. Synchronization is defined in terms of the PREMO event system. The basic unit of this is the event structure which encapsulates information about events: the name of the event, data associated with the event (specified as a sequence of key value pairs) and a reference to the object generating the event, the event source. When a Synchronization Element is attached to the coordinate space it defines a reference point. In addition to the event structure, the synchronization element also defines a boolean Wait flag. When a reference point is reached, the synchronizable object makes a message call to the object stored in the reference point, using the operation reference to identify which operation it has to call, and using the event instance 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. Operations are defined on synchronizable objects to add and delete reference points, and to add and delete synchronization elements associated with reference points.

A synchronization element is described by a structure whose main components are a reference to an event handler, plus the event that should be signalled to that handler when the element is activated. The specification of event structures is straightforward, apart from object references. The approach taken in the PREMO documents is to introduce explicit types to describe object references, so, for example, the type of a reference to an $EnhancedPREMOObject$ is $RefEnhancedPREMOObject$. In Object-Z, an instance of an object type is implicitly a reference to an object of that type, i.e. references are part of the semantic model underlying Object-Z, and are not mentioned explicitly in the text of a specification.

This highlights a general issue concerning the mapping between the informal text of the ISO/IEC PREMO documents and the formal description and highlights the need to pay particular attention to areas where the semantics of the notation and object model used in the PREMO documents might differ from the semantics of the formal description notation. This point is developed further in an earlier paper [4].

2.4 The 'Synchronizable' Object Type

A synchronizable object is a finite state machine that controls progression of the object through its coordinate space. In the formal description, the object $Synchronizable$ is defined as an object type in Object-Z. The state of a $Synchronizable$ object includes attributes such as the start and end positions, the current direction of play, current mode, current position, repeat flag, number of loops to be executed, synchronization points defined, etc.

There are some differences between the concepts in PREMO for accessing objects and the Object-Z access concepts which cause some difficulties. PREMO uses concepts of read-only attributes, read/write attributes, and internal (protected) operations. The latter is derived in part from the access control facilities provided by C++ and Java [10]. Although Object-Z does have a concept of (externally visible) attribute, there is no direct counterpart to protected operations. The PREMO standard defines a number of conventions for 'decorating' operation names to indicate the accessibility of the operation. These are not used in the specification, although the Z convention of naming an operation schema ΦS is used for naming operations that are not final operations but 'frames' that captures a common pattern of processing that will be used as the basis for defining subsequent operations.

The attributes which define the start and end position of a synchronizable object are read/write attributes in PREMO. Setting the values of these attributes may cause exceptions, for example if an attempt to set a start position that is after the stop position currently set. In Object-Z, the set and get methods for attributes are not an explicit part of the text and so the exceptions are difficult to document. This is a good example of one of the trade-offs involved in specification, in this case between explicitness and readability. It could be argued 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.

In order to describe the behaviour in the standard, it is necessary to introduce more structure into the Object-Z specification than is contained in the standard. This is necessary in order to describe how an object progresses along its coordinate space in PLAY mode, how operating mode changes occur, how reference points are treated during progression and the actions that occur when a reference point is encountered. More will be said about this shortly.

Initialization of objects is also handled differently in PREMO and Object-Z. The PREMO object model provides explicit initialization operations, including $initializeOnCopy$. Initialization in our specification is treated in the normal Object-Z way, i.e. by defining an initial state rather than an initialization operation. The $initializeOnCopy$ operation would be problematic to specify without making the details of the underlying machine environment visible in the specification.

The meaning of progression through the coordinate space is quite interesting to define. When an object is in the PLAY state it carries out its internal processing in a loop of processing stages, each of which consists of the following steps:

1. The value of the current position is advanced using the operation $progressPosition$ which returns the required next position. This operation is 'protected' meaning that the operation is not accessible to clients of the object via its interface, but can be modified within object types that inherit from this class.

2. This required position is compared with the current position and the end position, and the following actions are performed:

   (a) If there are reference points lying between the current position and the newly calculated position, then any associated synchronization actions are performed (in the order in which they are defined on the progression space). This means:

      - perform data presentation for any data identified by the points on the progression space between the current position or the previous reference point and the next reference point or the end point;
      - invoke the operation, whose description is stored in the reference point, on the object whose reference is stored at the reference point, using the stored event as an argument;
      - if the Wait flag stored in the synchronization element belonging to the reference point is set to TRUE, the object's state is changed to WAITING. This internal transition is the only means by which the WAITING state can be entered. If the state of the object is set back, eventually, to PLAY, the stage continues at this point.

(b) If the required position is smaller than the end position, then this becomes the local position and the processing stage is finished.

Two aspects of this processing cycle are left unspecified in the definition of $Synchronizable$:

- what 'data presentation' means, and
- the detailed semantics of the $progressPosition$ operation.

Both these aspects should be specified in the appropriate subtypes of $Synchronizable$. The abstract specification of a synchronizable object is such that no media specific semantics are directly attached to it. Subtypes, realizing specific media control should, through specialization, attach semantics to the object through their choice of the type of the internal coordinate system, through a proper specification of what data presentation means, and through a proper specification of the $progressPosition$ operation. The latter defines what it really means to "advance" along the internal coordinate system. For example, this progression may mean the generation of the next animation frame, decoding the next video frame, advance in time, etc.

> The following operation calculates the next location required; it is expected that it will be specialised by subclasses to address behaviour specific to various types of media. It is a 'protected' operation in the sense that clients of a $Synchronizable$ object are not supposed to invoke this. The point in the coordinate space that will be visited next is returned as an output.
>
> $progressPosition$
>
> $$
> \begin{array}{l}
> \Delta(requiredPosition, stepping) \\
> newPosition! : C \\
> \hline
> currentState = PLAY \wedge \neg\ stepping \\
> \exists\, count : \mathbb{N} \mid count < nloop \bullet \\
> \quad (newPosition!, count) \in span \\
> \quad requiredPosition' = (newPosition!, count) \\
> stepping'
> \end{array}
> $$

The specification, following the PREMO documents, places few constraints on how the required position is chosen.

Having chosen a new position, the specification then has to deal with any reference points and associated synchronization actions lying between the current position and the newly calculated position and termination as a result of reaching the end position after the required number of loops. The detail of the specification is quite involved, but the basic idea behind the approach taken is the introduction of an additional mode, 'stepping mode' which is used to model the sequence of operations which are assumed to take place internally. Stepping mode is not used in the PREMO documents; there the style is more declarative, saying what the behaviour should be rather than how it should be accomplished and not relating this in any direct way to the state of the object. While it could be argued that such detail is better captured in a process oriented language, a better alternative (in principle) would be to find a more declarative means of specifying the behaviour of the object, perhaps through invariants over behaviours. This is an area for further work.

This raises an important issue about PREMO, which has implications for other standards and systems, for example VRML [15]. 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 specification of $progressPosition$ given above, and the semantics of the internal $stepping$ mode that will presented shortly, 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 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 [16].

Four conceptual operations are introduced to describe the aspects of the stepping mode processing cycle, which are subsequently composed to describe the behaviour of the cycle. The first operation, $\Phi step$, represents the selection of a new location between the current position and required position for which the related data will be presented. Some of the locations between the current point and this new point may be skipped, though no location which corresponds to a reference point may be skipped. This approach certainly has a discrete flavour, whereas PREMO itself describes an ideal situation where the progression space may be continuous. However, the specification of $\Phi step$ still does not define how the intermediate position is chosen, it merely narrows down the conditions that the intermediate position must satisfy. Refining the specification of $\Phi step$ into a subtype describing a particular media type, say a sampled media type, could choose the intermediate position to coincide with the limit of the range of the next sample in the underlying coordinate system. One aspect of the specification presented here that has not yet been explored is specialization through subtyping, and in particular a check that the behaviour of concrete media types can in fact be described in the same specification structure as used here for the abstract PREMO components.

The second operation, $\Phi signal$, describes the signalling that takes place when a reference point is reached. The final two operations deal with termination of stepping mode. $\Phi doneStepping$ deals with the case that the value of the intermediate point reaches the $requiredPosition$ or while in WAITING mode the object undergoes a $stop$ or $pause$ operation. The case that the object has completed playing is dealt with by $\Phi donePlay$. Each cycle of processing is then defined by the progression to a new position followed by a sequence of step and signal operations. As a consequence of the moding induced by the $stepping$ flag, this processing is forced to continue until the required position has been reached.

$$\Phi behaviour \mathrel{\widehat=} (\Phi step \land \Phi signal) \lor \Phi doneStepping \lor \Phi donePlay$$

Specification of most of the operations on $SynchronizableObject$ is straightforward. The operation to set a synchronization element, $setSyncElement$ will serve as an illustration. A synchronization element can be set at a given coordinate within the extent of a media object. This cannot be done while the object is in PLAY state, and the specified coordinate must be valid for that object, i.e. between the minimum and maximum bounds. The specification is given below. The notation for describing exceptions is discussed in a later section.

> A synchronization element cannot be set while the object is in PLAY state, and the specified coordinate must be valid for that object, i.e. between the minimum and maximum bounds.
>
> **setSyncElement**
>
> $\Delta(refpoints)$
> $refpoint? : C$
> $syncData? : SyncElement$
>
> ---
>
> $currentState \in \{PAUSED, STOPPED\}$
> $\quad \longrightarrow \boxed{\text{exc}}$ WrongState
> $refpoint? \in minimumPosition \ldots maximumPosition$
> $\quad \longrightarrow \boxed{\text{exc}}$ WrongValue
>
> ---
>
> $refpoints' = refpoints \oplus \{refpoint? \mapsto syncData?\}$
> WrongState $\vee$ WrongValue $\longrightarrow$ $refpoints' = refpoints$

PREMO also allows periodic synchronization events to be defined. Periodic events are specified by two coordinates, one giving the time of the first event, the other giving the time that should elapse between events (the periodicity). The common synchronization element that should be invoked on each occurrence of an event is also given as an argument to the operation.

> **setPeriodicSyncElement**
>
> $\Delta(refpoints)$
> $startRefPoint? : C$
> $periodicity? : C$
> $syncData? : SyncElement$
>
> ---
>
> $currentState \in \{PAUSED, STOPPED\}$
> $\quad \longrightarrow \boxed{\text{exc}}$ WrongState
> $startRefPoint? \in minimumPosition \ldots maximumPosition$
> $\quad \longrightarrow \boxed{\text{exc}}$ WrongValue
>
> ---
>
> **let** $points == \{n : \mathbb{N} \bullet (startRefPoint? + n \times periodicity?)\} \bullet$
> $$refpoints' = refpoints \oplus \left(\begin{array}{l} (minimumPosition \ldots maximumPosition) \\ \lhd \\ \{p : points \bullet p \mapsto syncData?\} \end{array} \right)$$
> WrongState $\vee$ WrongValue $\longrightarrow$ $refpoints' = refpoints$

To describe the operation, we first construct the (infinite) set of all the time points that are related by the start point and periodicity. Each point in this set is mapped to the common synchronization element, and this mapping, restricted to the bounds of the object's coordinates, is written into the reference points. Any existing synchronization point that happens to occur at the same time as a periodic event is thus replaced by the new event.

2.5 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 operator $inquireTick$ 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 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. In part this mirrors the looseness of the requirements placed on time in PREMO.

Timers are defined as a combination of $Clock$ and a state machine with RUNNING, STOPPED and PAUSED states.

$TimeSynchronizable$ objects are $Synchronizable$ objects enriched with an timer interface and an attribute called $speed$ which relates progress through the coordinate space inherited from $Synchronizable$ with the progression of time as measured by the $Timer$. In other words, speed defines the number of coordinate space units that an instance of the object type will progress through in one tick.

The formal description of $TimeSynchronizable$ in Object-Z is straightforward except that the $TimeSynchronizable$ object type overloads a number of operations inherited from $Synchronizable$ to allow time values to be used as alternatives to positions within the coordinate space. Object-Z does not allow such overloading. Again there is a trade-off here between 'bending' the formal description technique to align with the description in the PREMO document or using the formal description technique without enrichment at the cost of a more complex mapping between the formal and the informal descriptions.

2.6 Event Handlers

The $EventHandler$ object type provides basic support for allowing objects to register interest in particular events, and for those objects to be notified, via the handler, when such events occur. Some preliminary definitions are required. When an object registers interest in an event, the registration is given a unique $EventId$ that can be used subsequently to unregister interest. Also, a sequence of constraints on the key-value properties of events can be provided as an additional filter. A constraint matching mode ($AndOr$) determines whether all constraints, or only some, must match for success.

Rather than represent all the details of such constraints, we make use of the abstraction provided by specification languages and simply indicate that there is a satisfaction relationship (sat) between constraints and key-value pairs such as those that appear as event data.

The operations for registering and unregistering interest in an event are quite straightforward.

Dispatching an event to the event handler invokes the callback operation of all objects that have registered interest in the event and for which the associated constraint is satisfied by the event instance.

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

The universal quantifier in the predicate part of this operation effectively suggests that the notifications are performed in parallel. In practice, notification may well be sequential, and as a result of execution time and propagation delays (particularly for remote objects), race conditions are a possibility. For example, a callback to one object may trigger processing that sends an 'unregister' request back to the event handler for an object that has not (yet) been notified of the current event. It should also be noted that the description of this and other operations in which an operation is invoked on some object reference relies critically on the semantics of object identity developed recently for Object-Z.

The callback routine of an event handler is just the dispatch event operation, with an appropriate renaming of the input variables.

2.7 Synchronization Points

A synchronization point is an event handler, specialised so that a set of objects can be associated with each event name. Subclasses of this type can specialise the behaviour of event dispatching so that conditions of this set are checked before the operation associated with the event is invoked.

A particular form of synchronization, representing a common need in multimedia applications, is defined by a subtype of $SynchronizationPoint$ called $ANDSynchronizationPoint$. Instances of this object type wait until all objects that have registered as event sources have signalled the event, before the object then invokes the callback operation on objects that have registered interest in being notified on the event.

The 'dispatch event' operation in this type checks whether the object that signalled the latest event completes the set of objects that have registered interest in that event. If so, the $received$ register is reset so that the objects can again synchronize on the event, and the $callback$ action is invoked on objects that have registered interest in this event, subject to the same filtering mechanism as described in the version of this operation defined in $EventHandler$. Note again that this aspect of the behaviour of this object type is not clear from the Standard. If all required objects have not yet signalled the event, the latest event notification is added to the $received$ register.

3 Exception Handling

The PREMO Standard does define an explicit model of exceptions, and indicates the circumstances in which particular exceptions should be raised. A preliminary approach to modelling exceptions within Object-Z specifications appeared in an internal report by members of the PREMO Rapporteur Group [9]. The approach taken in this paper is a development of that first attempt.

We introduce the following type to define the exception values that are discussed in the formal specification. The list given below is not complete, but it should be clear that it could be enumerated if required.

$$
\begin{array}{lcll}
Exception & ::= & \mathrm{Okay} & \text{– Used in the specification to indicate no exception} \\
& | & \mathrm{WrongState} & \\
& | & \mathrm{WrongValue} & \\
& | & \mathrm{EndPosition} & \\
& | & \ldots & \text{– to be completed}
\end{array}
$$

A common problem with dealing with exceptions in any representation is that the details of exceptional cases, and the corresponding actions, can easily obscure the normative behaviour of an operation. However, one of the values of a formal specification is that it helps to identify possible causes for failure, so it is highly desirable to document both normal behaviour and the behaviour that results when an operation is invoked inappropriately. To overcome the problem of detail, we introduce notation to hide some of the detail. For example, the full specification of the $setSyncElement$ operation from the class $Synchronizable$ appears below. What is of concern here is the amount of material needed to describe the two exceptions.

$$
\begin{array}{l}
\textit{setSyncElement} \\
\hline
\Delta(\mathit{refpoints}) \\
\mathit{refpoint}? : C \\
\mathit{syncData}? : \mathit{SyncElement} \\
\mathit{exceptions}! : \mathit{Exception} \\
\hline
\mathit{currentState} \in \{\mathit{PAUSED}, \mathit{STOPPED}\} \\
\mathit{refpoint}? \in \mathit{minimumPosition} \ldots \mathit{maximumPosition} \\
\mathit{refpoints}' = \mathit{refpoints} \oplus \{\mathit{refpoint}? \mapsto \mathit{syncData}?\} \\
\mathit{exceptions}! = \mathit{Okay} \\
\vee \\
\mathit{currentState} \notin \{\mathit{PAUSED}, \mathit{STOPPED}\} \wedge \mathit{exceptions}! = \mathit{WrongState} \\
\quad \wedge\ \mathit{refpoints}' = \mathit{refpoints} \\
\vee \\
\mathit{refpoint}? \notin \mathit{minimumPosition} \ldots \mathit{maximumPosition} \\
\mathit{exceptions}! = \mathit{WrongValue} \wedge \mathit{refpoints}' = \mathit{refpoints}
\end{array}
$$

Another specification of the same operation is given below, this time using some conventions to structure the description of exceptions and related behaviour.

$$
\begin{array}{l}
\textit{setSyncElement} \\
\hline
\Delta(\mathit{refpoints}) \\
\mathit{refpoint}? : C \\
\mathit{syncData}? : \mathit{SyncElement} \\
\hline
\mathit{currentState} \in \{\mathit{PAUSED}, \mathit{STOPPED}\} \\
\quad \longrightarrow \boxed{\text{exc}}\ \text{WrongState} \\
\mathit{refpoint}? \in \mathit{minimumPosition} \ldots \mathit{maximumPosition} \\
\quad \longrightarrow \boxed{\text{exc}}\ \text{WrongValue} \\
\hline
\mathit{refpoints}' = \mathit{refpoints} \oplus \{\mathit{refpoint}? \mapsto \mathit{syncData}?\} \\
\text{WrongState} \vee \text{WrongValue} \longrightarrow \mathit{refpoints}' = \mathit{refpoints}
\end{array}
$$

The points to note are that:

1. Preconditions whose failure should raise an exception are gathered into a special section of the operation, and each such condition is tagged by the name of the exception (e.g. ‘WrongState’) that should be raised if that precondition is not satisfied when the operation is invoked.

2. We assume that all exceptions are returned through a variable $exceptions!$ of type $Exception$, which is implicitly declared in any operation that defines exceptions.

3. The ‘normal processing’ of an operation is written as usual, with the return of an ‘okay’ value through $exceptions!$ being implicit.

4. A gap is used to separate the 'normal' behaviour of an operation from predicates that describe required behaviour in exceptional cases. In this section of the operation, exception names are related to the predicate that should hold in the operation if that exception is raised.

4 Conclusion

A substantial part of this document was written at the same time as the material that was entered into the Working and Committee Drafts of the PREMO documents. The result was that a number of issues, both minor and non-trivial, were detected before they became embedded into a Standards document and therefore the subject of formal commenting. In this way, the use of formal specification has been of significant help in supporting the development of the PREMO Standard. It was fortunate in that at the time the Working Draft was produced, both the editor of the relevant part of the PREMO document (PREMO Part 2) and the first author of this paper were at the same institution, and it was possible to develop both documents in parallel, with the two authors sitting in a room and exchanging comments directly. Following further meetings of the PREMO Rapporteur Group, the Time and Synchronization aspects of PREMO underwent several changes, and the original specification became inconsistent with the base document. In preparing this paper, the two have been realigned, though in the process we have discovered a significant number of issues and questions concerning the technical content of PREMO that have had to be resolved within the Rapporteur Group. In summary, the value of a document such as this is twofold. First, it contains a precise statement of the expected behaviour of a portion of the PREMO standard. Second, the process of writing the document has, we believe, contributed to improving the overall quality of that Standard.

With respect to the technical content of the specification, we have found that Object-Z provides a good starting point for building a description of PREMO in a way that mirrors the structure of the ISO document. This is important, since not all members of the Rapporteur Group are experts in formal specification or Object-Z, and the close mapping simplifies the task of explaining consequences and issues identified from the formal model in terms of the material as presented in the normative document. The process of mapping the PREMO object types into Object-Z classes is not however straightforward. In an earlier paper [4] we reported on issues related to differences in the object models of PREMO and Object-Z. The concerns identified here are wide ranging, but a general theme has been that the state-operation style of Object-Z can necessitate encoding aspects of PREMO object type behaviour in a rather operational style. This is perhaps best illustrated by the model of internal progression within the $Synchronizable$ object type.

The concerns raised in this paper are not criticisms of Object-Z. 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.

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 are grateful to Dr Mieke Massink for valuable comments on the work reported here.

References

[1] D.B. Arnold and D.A. Duce. *ISO Standards for Computer Graphics: The First Generation*. Butterworths, 1990.

[2] D.A. Carrington, D.J. Duke, R.W. Duke, P. King, G.A. Rose, and G. Smith. Object-Z: An object-oriented extension to Z. In S. Vuong, editor, *Formal Description Techniques (FORTE'89)*. North Holland, 1990.

[3] R.B. Dannenberg, T. Neuendorffer, J. Newcomer, D. Rubine, and D. Anderson. Tactus: Toolkit-level support for synchronized interactive multimedia. *Multimedia Systems Journal*, 1(2):77–86, 1993.

[4] 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 and Interfaces*, 17(5-6):491–509, 1995. Special Issue on Formal Methods and Standards.

[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.J. Duke, D.A. Duce, I. Herman, and G. Faconti. Specifying the PREMO Synchronization Objects. Technical Report 02/97-R048, ERCIM Research Reports, 1997. Available from http://www-ercim.inria.fr/.

[7] R. Duke, P. King, G. Rose, and G. Smith. The Object-Z specification language: Version 1. Technical Report 91–1, Software Verification Research Centre, University of Queensland, 1991.

[8] R. Duke, G. Rose, and G. Smith. Object-Z: A specification language advocated for the description of standards. *Computer Standards and Interfaces*, 17(5-6):511–533, 1995. Special Issue on Formal Methods and Standards.

[9] D.J. Duke (editor). Time and synchronisation in PREMO: A formal specification of the NNI proposal. Technical Report Technical Report OME-116, ISO/IEC JTC1 SC24/WG6, 1995. Available at ftp://ftp.cwi.nl/pub/Premo.

[10] A. Freeman and D. Ince. *Active Java: Object-Oriented Programming for the World Wide Web*. Addison-Wesley, 1996.

[11] I.J. Hayes. *Specification Case Studies*. Series in Computer Science. Prentice Hall International, second edition, 1992.

[12] I. Herman, G.S. Carson, J. Davy, P.J.W. ten Hagen, D.A. Duce, W.T. Hewitt, K. Kansy, B.J. Lurvey, R. Puk, Reynolds. G.J., and H. Stenzel. Premo: An ISO standard for a presentation environment for multimedia objects. In D. Ferrari, editor, *Proceedings of the Second ACM International Conference on Multimedia (MM'94)*. ACM Press, 1994.

[13] I. Herman, G.J. Reynolds, and J. Van Loo. PREMO: An emerging standard for multimedia presentation. *IEEE Multimedia*, 1996. To appear.

[14] ISO/IEC 13522-1. *Information processing systems, information technology, coding of multimedia and hypermedia information (mheg)*. Geneva ISO Central Secretariat, 1994.

[15] ISO/IEC 14722. *Information processing systems, computer graphics, virtual reality modelling language (VRML)*. Geneva ISO Central Secretariat, 1996.

[16] N.G. Leveson, M.P.E. Heimdahl, H. Hildreth, and J.D. Reese. Requirements specification for process control systems. *IEEE Transactions on Software Engineering*, 20(9):684–707, 1994.

[17] G.J. Reynolds, D.A. Duce, and D.J. Duke. Report of the ISO/IEC JTC1/SC24 special rapporteur group on formal description techniques. Technical Report ISO/IEC JTC1/SC24 N1152, ISO/IEC, 1994.

[18] J.M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International, 1992. Second edition.